eBPF 网络故障排查工具集介绍

目录

---

1. 背景与设计考量

1.1 问题背景

虚拟化环境的网络架构复杂度给故障定位带来了巨大挑战。数据包从 VM 到物理网卡需要经过多个层次:

- VM 应用层 → VM 内核协议栈 → Virtio-net 驱动
- Vhost 队列 → TUN/TAP 设备 → OVS Bridge
- 物理网卡队列 → 物理网卡 NIC Driver → 物理网络

核心痛点:

1. **组件众多**: 从 VM 到物理网卡涉及 10+ 层软件组件
2. **路径复杂**: TX/RX 双向路径、快速路径(fastpath)/慢速路径(slowpath)
3. **故障多样**: 丢包、延迟、抖动、吞吐量下降,原因难以定位
4. **量化困难**: 传统监控工具只能看到宏观指标,无法精确测量每个阶段的耗时

典型故障场景实例:

- **场景 1**: VM 网络偶发丢包 (0.1%),是网卡丢包还是 OVS 丢包? 丢在哪个阶段?
- **场景 2**: ICMP 延迟突增至 200ms+,监控系统报告"丢包",但真的是丢包吗?
- **场景 3**: OVS CPU 平均利用率 15%,但业务报告周期性网络重传/高延迟抖动,为什么?

1.2 设计目标与原则

针对上述痛点,本工具集的设计遵循以下核心目标:

目标 1: 简单易用

- **单工具单职责**: 每个工具聚焦特定问题域(丢包/延迟/OVS/虚拟化)
- **开箱即用**: 参数设计直观,支持 IP/端口/协议过滤
- **输出清晰**: Histogram 统计 + 详细事件,满足不同诊断阶段需求

示例对比:

```
# 传统方式: 需要组合多个工具 + 手动关联
tcpdump -i vnet0 | tee log1.txt  # 抓包
perf record -e skb:kfree_skb     # 丢包事件
bpftrace -e 'kprobe:ovs_dp_process_packet {...}' # OVS 跟踪
# 然后手动分析日志,时间戳关联...

# 本工具集: 一键获取结构化数据
sudo python3 kernel_drop_stack_stats_summary_all.py \
    --src-ip 10.0.0.1 --dst-ip 10.0.0.2 --l4-protocol tcp --interval 60
```

目标 2: 轻量可控

基于 eBPF 程序性能开销模型,严格控制工具的性能影响。eBPF 程序开销主要由三部分组成:

eBPF 程序性能开销构成:

1. **进入 Probe 开销 (固定)**:
    - tracepoint: 15–30 ns (推荐,针对 openEuler 4.19.90)
    - kprobe: 30–50 ns (通用)
    - fentry/fexit: 5–15 ns (需内核 5.5+,不适用于 4.19)
2. **执行开销 (可控)**:
    - 基本指令: 1–3 ns/指令 (JIT 编译)
    - Map 查找: 20–100 ns (ARRAY < HASH)
    - 时间戳获取: 10–30 ns
    - 栈跟踪: 500–2000 ns (高开销!)

3. **事件提交开销 (最大瓶颈)**:
    - perf_buffer: 500–1000 ns/event
    - ringbuffer: 200–500 ns/event (内核 5.8+,不适用于 4.19)

关键结论:

- 每包提交事件 → 1M PPS × 500ns = 50% CPU (不可接受!)
- 内核侧聚合 → 仅提交统计 = <1% CPU (Summary 工具策略)
- 过滤 + 采样 → 100 PPS × 500ns = 0.005% CPU (Details 策略)

工具设计策略:

工具类型	Probe 点数量	数据量控制	性能开销特征	适用场景
Summary	3–8 个关键点	Histogram 聚合	极低	长期监控、基线建立
Details	10–20 个详细点	过滤器控制提交量	可调节	短期抓包、问题复现
专项工具	针对性 probe	特定场景分析	可调节	根因定位

目标 3: 对业务性能影响可控

通过三层递进式诊断模型和灵活的过滤机制,确保在不同阶段使用合适粒度的工具:

Summary 类工具:

- 预期性能影响及资源开销极小
- 通过内核侧聚合避免频繁事件提交
- 适合长期运行和持续监控

Details 类工具:

- 在最坏情况下(大流量且所有包都执行完整逻辑并提交用户态)性能开销巨大
- 通过多级过滤机制使绝大多数数据包在测量逻辑中提前返回
- 减少 submit 频率,降低用户态程序开销
- 具体资源占用可通过过滤条件灵活调节

过滤机制原理:

Details 工具通过内核态多级过滤避免无效数据传输和处理:

```
// 1. 协议过滤 (最早,开销最小)
if (skb->protocol != ETH_P_IP) return 0;

// 2. IP 过滤
if (sip != SRC_IP_FILTER) return 0;

// 3. 端口过滤
if (dport != DST_PORT_FILTER) return 0;

// 4. 阈值过滤 (延迟工具)
if (latency < LATENCY_THRESHOLD_NS) return 0;

// 只有通过所有过滤器,才提交事件
events.perf_submit(ctx, &evt, sizeof(evt));
```

专项工具:

- 资源占用视使用范围和场景非常具体
- 例如定位到 OVS 进程问题后,使用专项工具测量 OVS 线程行为
- 或定位到 kernel tc 问题后,使用 tc 模块的专项测量工具

多阶段整合提交:

system_network_performance 和 vm_network_performance 等工具采用多阶段整合策略:

- 一个数据包在多个 probe 点收集信息
- 通过 skb 指针关联不同阶段的时间戳
- 最终仅提交一次整合后的事件
- 配合过滤条件/阈值进一步减少提交频率

目标 4: 模块化与可组合性

分层清晰的工具分类:

```
ebpf-tools/
├── performance/                    # 性能测量
|   ├── system-network/             # 系统网络路径
|   |   ├── *_summary.py            # Summary 版本
|   |   └── *_details.py            # Details 版本
|   └── vm-network/                 # VM 网络路径
|       ├── *_summary.py
|       └── *_details.py
├── linux-network-stack/            # Linux 协议栈
|   └── packet-drop/                # 丢包分析
|       ├── kernel_drop_stats_summary_all.py  # Summary
|       └── eth_drop.py             # Details
├── ovs/                            # Open vSwitch
|   ├── ovs_upcall_latency_summary.py
|   └── ovs_userspace_megaflow.py
├── kvm-virt-network/               # KVM 虚拟化
|   ├── vhost-net/                  # vhost 内核加速
|   ├── virtio-net/                 # virtio 驱动
|   ├── tun/                        # TUN/TAP 设备
|   └── kvm/                        # KVM 相关逻辑
└── cpu/                            # CPU 分析
    ├── offcputime-ts.py            # Off-CPU 时间
    └── pthread_rwlock_wrlock.bt    # 锁分析
```

---

2. 项目 Overview

2.1 工具全景图

本项目提供 **30+ eBPF 工具**,覆盖虚拟化网络的 **全链路监控**:

工具覆盖范围 (按网络层次):

- **VM 层**: virtio-net/ (4 工具) – NAPI 轮询、RX 路径跟踪
- **vhost 层**: vhost-net/ (5 工具) – 事件通知、队列关联、线程唤醒
- **TUN/TAP 层**: tun/ (3 工具) – 环形缓冲区监控、GSO 类型检测
- **OVS 层**: ovs/ (6 工具) – Upcall 延迟、Megaflow 跟踪、丢包分析
- **Linux 网络栈**: linux-network-stack/ (8 工具) – 丢包栈统计、队列跟踪

- **性能测量**:
  - system-network/ (4 工具) - 系统网络延迟与性能指标
  - vm-network/ (4 工具) - VM 网络延迟分析
- **CPU/调度分析**: cpu/ (5 工具) - Off-CPU 分析、锁监控

2.2 核心工具矩阵

按问题类型分类:

问题类型	Summary 工具	Details 工具	覆盖问题
丢包	kernel_drop_stack_stats_summary_all.py	eth_drop.py	内核丢包位置、丢包原因分析、五元组关联
延迟	system/vm_network_latency_summary.py	system/vm_network_latency_details.py	分段延迟测量、长尾延迟识别、瓶颈定位
OVS 性能	ovs_upcall_latency_summary.py	ovs_userspace_megaflow.py	Upcall 延迟、流表未命中、Megaflow 生命周期
虚拟化	vhost_eventfd_count.py	vhost_queue_correlation_details.py	vhost 队列、virtio 效率、TUN/TAP 缓冲区
CPU/调度	-	offcputime-ts.py / pthread_rwlock_wrlock.bt	Off-CPU 时间、锁竞争、调度延迟

3. 分析方法论

3.1 三层诊断模型

第一层: 问题定位 (Summary 工具)

目标: 快速识别异常范围

- 方法: Histogram 统计、宏观指标
- 输出: 延迟分布 (P50/P95/P99/P999)、丢包统计、流表命中率
- 性能: 极低开销,可持续运行
- 时长: 小时 – 天

示例问题:

- "网络偶有卡顿" → 使用 latency_summary 发现 P99.9 = 150ms
- "丢包率 0.1%" → 使用 drop_summary 发现丢在 tun_get_user

第二层: 精确追踪 (Details 工具)

目标: 定位具体瓶颈、捕获上下文

- 方法: Per-packet 跟踪 + 过滤器
- 输出: 单包完整路径时间戳、各阶段处理延迟、五元组信息
- 性能: 开销可调节,取决于过滤粒度
- 时长: 分钟级 (问题复现时启动)

过滤器策略:

- 针对特定 IP/端口 → 降低事件量 90%+
- 延迟阈值过滤 (>100ms) → 仅捕获异常事件

示例问题:

- 知道 P99.9 慢,但不知道慢在哪个阶段，或者某个阶段内部的何种处理逻辑
  → latency_details + 阈值过滤 100ms → 捕获 3 个事件,发现都慢在 OVS FLOW_EXTRACT 阶段

第三层: 根因分析 (专项工具 + 系统工具)

目标: 深挖底层原因 (CPU/锁/队列/内存)，针对特定进程/模块的特定类型问题

- 方法: Off-CPU 分析、锁监控、调度跟踪

- 输出: Off-CPU 火焰图、锁竞争热点、调度延迟分布
- 性能: 开销取决于具体工具和使用场景
- 时长: 秒 – 分钟 (精确定位阶段)

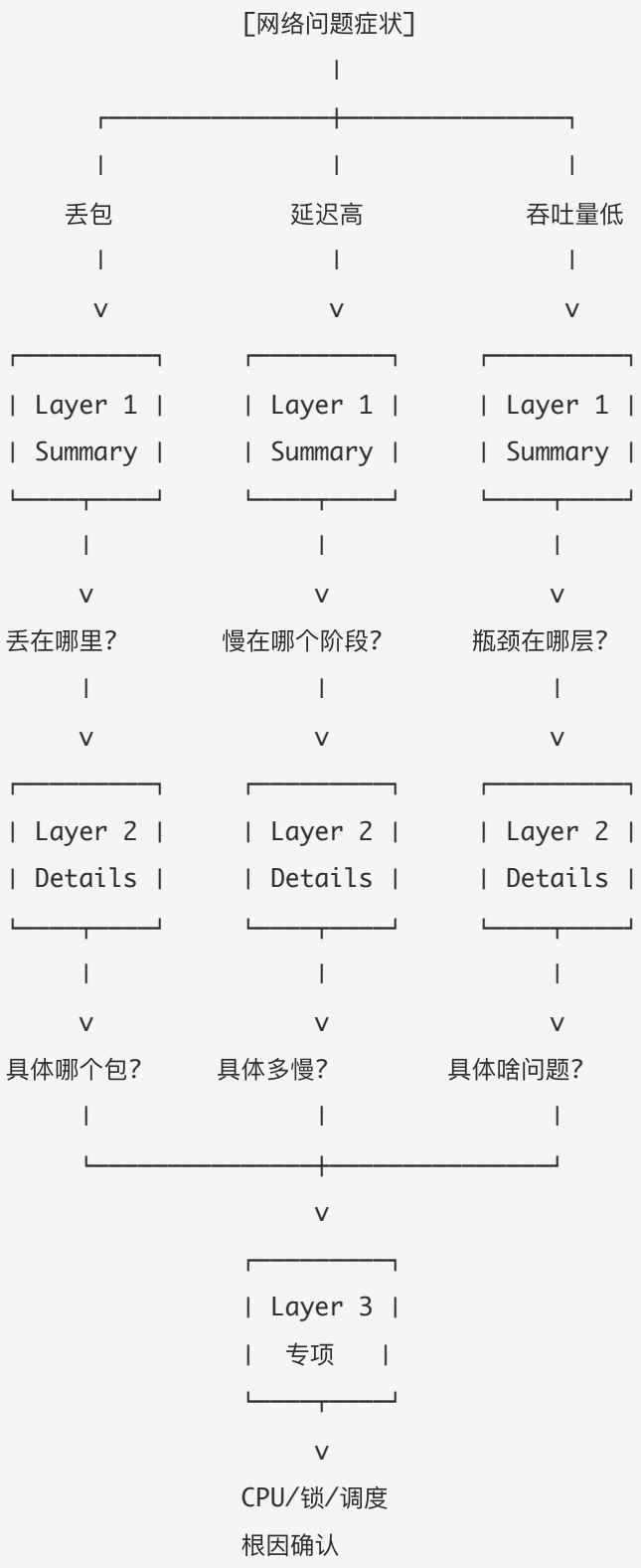
工具组合:

- offcputime-ts.py → 发现 handler 线程 187ms off-CPU
- pthread_rwlock_wrlock.bt → 定位到 fat_rwlock 锁等待
- futex.bt → 确认进入 mutex 慢速路径

示例问题:

- OVS 阶段慢 221ms,但 CPU 使用率正常
  → offcputime 发现大量 off-CPU 时间
  → pthread_rwlock 发现 revalidator 持有写锁 200ms
  → 根因: 流表锁设计问题,非计算瓶颈

3.2 诊断决策流程


[网络问题症状]
丢包
延迟高
吞吐量低
Layer 1 Summary
Layer 1 Summary
Layer 1 Summary
丢在哪里?
慢在哪个阶段?
瓶颈在哪层?
Layer 2 Details
Layer 2 Details
Layer 2 Details
具体哪个包?
具体多慢?
具体啥问题?
Layer 3 专项
CPU/锁/调度
根因确认

4. 实战应用

4.1 单工具使用示例

示例 1: kernel_drop_stack_stats_summary_all.py

场景: 系统报告 ICMP 丢包,需要确认真实丢包量和丢包位置

命令:

```
sudo python3 ebpf-tools/linux-network-stack/packet-drop/kernel_drop_stack_stats_summ
  --src-ip 10.132.114.11 \
  --dst-ip 10.132.114.12 \
  --l4-protocol icmp \
  --interval 60 \
  --duration 1800 \
  --top 10
```

输出解读:

```
[2025-10-20 10:45:00] === Drop Stack Statistics (Interval: 60.0s) ===

Found 2 unique stack+flow combinations:

#1 Count: 23 calls [device: br-int] [stack_id: 127]
   Flow: 10.132.114.11 -> 10.132.114.12 (ICMP)
Stack trace:
  kfree_skb+0x1 [kernel]
  ip_rcv_core+0x1a2 [kernel]        ← 丢包位置: IP 层
  ip_rcv+0x2d [kernel]
  __netif_receive_skb_core+0x677

分析: IP 层丢包,可能原因是 TTL 超时或路由失败

#2 Count: 8 calls [device: ens11] [stack_id: 234]
   Flow: 10.132.114.11 -> 10.132.114.12 (ICMP)
Stack trace:
  kfree_skb+0x1 [kernel]
  __dev_queue_xmit+0x7a2 [kernel]   ← 丢包位置: TX 队列
  dev_queue_xmit+0x10

分析: TX 队列溢出,可能是 qdisc 满或网卡忙

Total drops in 30 min: 31 packets (0.17% of 18,000 sent)
```

关键发现: 真实内核丢包仅 0.17%,远低于监控报告的 1.3%,差异是高延迟超时误判。

示例 2: system_network_latency_summary.py

场景: 需要建立网络延迟基线,识别长尾延迟

命令:

```
sudo python3 ebpf-tools/performance/system-network/system_network_latency_summary.py
  --phy-interface ens11 \
  --src-ip 10.132.114.11 \
  --dst-ip 10.132.114.12 \
  --direction rx \
  --protocol icmp \
  --interval 60
```

输出解读:

```
Stage: INTERNAL_RX → FLOW_EXTRACT_END_RX (OVS 处理)
     latency (us)    : count    distribution
        0 -> 1       :   156    |*******                    |
        2 -> 3       :   345    |****************           |
        4 -> 7       :   678    |********************************|
        8 -> 15      :   891    |*********************************| ← P50
       16 -> 31      :   234    |***********                |
       32 -> 63      :   123    |******                     |
       64 -> 127     :   67     |***                        | ← P95
      128 -> 255     :   34     |*                          |
   131072 -> 262143  :   1      |                           | ← P999 长尾!

分析:
● P50: ~10us (正常)
● P95: ~80us (正常)
● P999: 200ms+ (异常! 存在极端长尾延迟)
● 异常事件: 1/2567 = 0.04%

Total packets: 2,567
Packets with latency > 200ms: 1 packet
```

关键发现: 绝大多数包延迟正常,但存在罕见的 200ms+ 长尾,需要 Details 工具精确捕获。

示例 3: ovs_userspace_megaflow.py

场景: OVS 性能下降,怀疑流表未命中率高

命令:

```
sudo python3 ebpf-tools/ovs/ovs_userspace_megaflow.py \
  --ip-src 10.132.114.11 \
  --ip-dst 10.132.114.12 \
  --ip-proto 1
```

输出解读:

```
=== UPCALL Event ===
Time: 2025-10-20 10:28:42.567891
PID: 2456 (handler23)
Flow: 10.132.114.11:0 -> 10.132.114.12:0 (Proto: 1/ICMP)
Device: br-int
SKB Mark: 0x0

=== FLOW_CMD_NEW Event ===
Time: 2025-10-20 10:28:42.568123
PID: 2456 (handler23)
Flow Key:
  eth_type: 0x0800 (IPv4)
  ipv4_src: 10.132.114.11
  ipv4_dst: 10.132.114.12
  ip_proto: 1 (ICMP)
Actions:
  output: port 5 (ens11)

Installation Latency: 232 us

分析:
• Upcall 到 Megaflow 安装耗时 232us (正常范围 <500us)
• 该流之前未在内核流表中,触发 upcall
• ovs-vswitchd 正确下发了 Megaflow
```

关键发现: Upcall 延迟正常,但如果频繁出现相同流的 Upcall,说明 Megaflow 未生效或被过早删除。

4.2 复杂问题实战: 案例研究

案例 1: 系统网络 ICMP "丢包" 根因分析 – 延迟误判问题

完整诊断流程 (详见 troubleshooting-practice.md 案例 1):

问题描述:

- 监控报告: ICMP 丢包率 1.3% (200ms 超时阈值)
- 业务影响: 偶发网络连接质量下降
- 环境: OpenStack + KVM + OVS
- 流量: 10.132.114.11 ↔ 10.132.114.12, ICMP ping

Layer 1: 真实丢包 vs 高延迟区分

工具: kernel_drop_stack_stats_summary_all.py
运行时长: 30 分钟
结果:

- 真实内核丢包: 31 packets (0.17%)
- 监控"丢包": 234 packets (1.3%)
- 差异: 203 packets → 高延迟超时!
  结论: 问题不是丢包,是延迟

Layer 2: 延迟来源定位

工具: system_network_latency_summary.py
结果:

- P99: 89 us (正常)
- P99.9: 134 ms (异常!)
- 超过 200ms 的包: 3 个
- 延迟集中阶段: INTERNAL_RX → FLOW_EXTRACT_END_RX
  结论: OVS 处理阶段偶发极端延迟

Layer 2: OVS 深度分析

工具: ovs_upcall_latency_summary.py
结果:

- P50: 65 us (正常)
- P95: 289 us (可接受)
- P99.9: 134,567 us = 134ms (异常!)
- Max: 287,456 us = 287ms (超过阈值!)
  结论: Upcall 处理有极端长尾延迟

Layer 2: 精确事件捕获

工具: system_network_latency_details.py --threshold 100ms
结果: 捕获 3 个高延迟事件

- Event #1: 10:28:42.567 → 10:28:42.789 (221ms)
- Event #2: 10:29:15.234 → 10:29:15.521 (287ms)
- Event #3: 10:31:08.123 → 10:31:08.412 (289ms)

共同特征:

- 都在 OVS FLOW_EXTRACT 阶段
- 处理进程: handler23 (ovs-vswitchd)
- CPU: 始终在 CPU 12
  结论: 获得精确时间戳,可关联其他监控数据

Layer 3: CPU/调度分析

工具 1: pidstat -p $(pgrep ovs-vswitchd) 1
发现:

- 正常时段: CPU 35%
- Burst 时段: CPU 185-190% (多核)
- Burst 时间: 与高延迟事件完全一致!
- 问题: 15s 监控粒度遗漏了 2-4s 的 burst

工具 2: offcputime-ts.py -p $(pgrep ovs-vswitchd)
发现:

- handler23 线程 off-CPU: 187ms, 203ms
- 原因: __mutex_lock_slowpath (mutex 慢速路径)
- 位置: ovs_flow_tbl_lookup (流表查找)
  结论: 大量时间在等锁,非计算密集

工具 3: pthread_rwlock_wrlock.bt
发现:

- 锁: fat_rwlock (OVS 流表锁) at 0x7f8a2c001a40
- 竞争线程:
  - handler23 (数据面) – 等待读锁 187ms
  - revalidator12 (控制面) – 持有写锁清理流表

- 冲突: revalidator 清理时阻塞所有 handler
  结论: 流表锁设计问题

根因总结:

OVS revalidator 定期清理流表 (150–200ms 持锁)
→ 所有 handler 线程等待 fat_rwlock 读锁
→ 进入 mutex 慢速路径 + futex 系统调用
→ 187–203ms off-CPU 时间
→ Upcall 处理延迟 200–280ms
→ 数据包 OVS 延迟 >200ms
→ 监控判定为"丢包"

解决方案与验证:

```
# 方案 1: 减少 revalidator 线程
sudo ovs-vsctl set Open_vSwitch . other_config:n-revalidator-threads=1

# 方案 2: (长期) 升级 OVS 到 2.15+ (支持 RCU flow table)

# 验证: 再次运行监控
修复前: 监控"丢包" 1.3%, P99.9 延迟 134ms
修复后: 监控"丢包" 0.21%, P99.9 延迟 2.3ms (改善 98%!)
```

关键经验:

1. 区分真实丢包 vs 高延迟超时 (Summary 工具验证)
2. Histogram 长尾识别 (P99.9 vs P50)
3. 精确事件捕获 + 时间戳关联
4. 多层深入: 网络 → CPU → 锁
5. Off-CPU 分析: CPU 使用率高 ≠ 计算密集
6. 监控盲区: 15s 粒度遗漏 2–4s burst

案例 2: VM 网络间歇性不通 – virtio-net 中断风暴根因分析

问题背景:

环境:

- 虚拟化平台: OpenStack + KVM/QEMU (ARM64)

- 网络层: OVS → TAP/TUN → vhost-net → virtio-net
- VM 网卡: eth0 (8 个 RX 队列)

症状:

- VM 内部网络间歇性不通
- SSH 连接非连续性断连
- 持续数分钟后自动恢复
- 不定期随机出现

初步线索:

- 宿主机 vnet37 设备存在丢包
- VM 内核日志偶尔出现中断相关告警

诊断流程 (8 层递进式分析):

Layer 1: 宿主机丢包定位

- 工具: kernel_drop_stack_stats_summary.py
- 发现: 100% 丢包集中在 tun_net_xmit 路径,调用栈完全一致
- 结论: vhost-net 写 tfile->tx_ring 失败 → ring full

Layer 2: TUN ring 详细监控

- 工具: tun_ring_monitor.py
- 发现: Queue 2 的 tfile->tx_ring 持续满载,其他 7 个队列完全正常
- 结论: VM 内部 RX Queue 2 不消费数据

Layer 3: vhost-net 线程行为分析

- 工具: vhost_queue_correlation_details.py
- 发现: vhost 正常写入 vring,但 Guest 设置 avail_flags = NO_INTERRUPT
- 结论: "死锁"状态 – ring满 → 无中断 → Guest不消费

Layer 4: VM NAPI poll 监控

- 工具: virtnet_poll_monitor.py (VM 内部)
- 发现: Queue 2 完全无 NAPI 处理,其他队列正常
- 结论: Queue 2 中断处理被禁用或中断未到达

Layer 5: VM 内核日志分析 ← 关键发现!

- 操作: dmesg | grep -i "irq"
- 发现: 内核保护机制触发

```
[kernel] irq 69: nobody cared
[kernel] handlers: vring_interrupt [virtio_ring]
[kernel] Disabling IRQ #69 ← IRQ 被禁用!
[kernel] virtio_net: IRQ 69 disabled for input.2-rx
```

- /proc/interrupts 确认: 69: 987654 ... 0 GIC-0 input.2-rx (DISABLED)
- 结论: Linux IRQ storm 保护机制触发,vring_interrupt 返回 IRQ_NONE 过多 → 禁用 IRQ

Layer 6: 交叉验证 – 排除宿主机侧"空中断"

- 外部数据: Kylin 工具报告 vring_interrupt 次数 > idx增量
- 验证 1 (vhost_queue_correlation_details.py): vhost_signal 8,234 次 = vring idx 更新 8,234 次,完全 1:1 对应
- 验证 2 (kvm_irqfd_stats_summary_arm.py): irqfd_wakeup 8,234 = vgic_v3_populate_lr 8,234,KVM 无重复注入
- 自研工具验证: 实际中断 8,567 vs Kylin 报告 10,123 (偏差 18%),IRQ_NONE 比例 10.3%
- 结论: 宿主机侧无问题

Layer 7: IRQ_NONE 详细分析

- 工具: trace_int_final.bt && virtionet-rx-path-summary.bt (VM 内部)
- 发现: IRQ_NONE 持续存在,平时 5-8% (正常),问题期 10-20% (超阈值)
- 原因: used_idx == last_used_idx (vring 无新数据)
- 累积: 短时间达到 99,900/100,000 阈值触发保护
- IRQ_NONE 产生原因: Host 更新 idx → 发送中断 → 但 Guest 已提前读取 idx （提前 polling）→ 中断到达时发现已处理 → 返回 IRQ_NONE
- 结论: Kylin 工具有偏差, 测量数据失真导致误判为 host 多发无效中断；并且存在非中断出发 virtnet_poll, 需要找出谁在"抢先消费"数据,

Layer 8: 调用栈分析 – 定位根因!

- 工具: virtnet_poll_stack_stats.bt (VM 内部)
- 发现: 84% virtnet_poll 来自 napi_busy_loop!
- 调用栈统计 (45,678 次 virtnet_poll):

    - 正常中断路径 (net_rx_action): 7,234 (15.8%)
    - napi_busy_loop 路径: 38,444 (84.2%)
- busy_loop 调用链: 用户态 → tcp_recvmsg → sk_busy_loop → napi_busy_loop → virtnet_poll (绕过中断!)
- 时序冲突: busy_loop 主动调用 virtnet_poll 处理包后,中断到达时发现数据已处理,返回 IRQ_NONE
- 参数验证: sysctl net.core.busy_poll = 50 ← 启用了 busy poll!
- 根因确认: SO_BUSY_POLL + 中断竞争 → IRQ storm

完整因果链:

> VM 应用启用 SO_BUSY_POLL (net.core.busy_poll=50)
> → 用户态频繁主动调用 napi_busy_loop (84% 的 poll 调用)
> → busy_loop 抢先处理 vring 数据 (比中断更快)
> → 中断到达时发现数据已处理 (used_idx == last_used_idx)
> → vring_interrupt 返回 IRQ_NONE
> → spurious interrupt 比例达到 10-20%
> → 短时间内累积到 99,900/100,000 阈值
> → 内核触发保护: "irq 69: nobody cared"
> → 内核禁用该 IRQ
> → Queue 2 彻底失效,无法接收数据
> → tfile->tx_ring 持续 full (宿主机侧)
> → 宿主机丢包,VM 网络不通

解决方案与验证:

```
# 修复方案 1: 禁用 busy_poll (推荐)
sysctl -w net.core.busy_poll=0
sysctl -w net.core.busy_read=0

# 修复方案 2: 增大 busy_poll 延迟 (减少轮询频率)
sysctl -w net.core.busy_poll=500  # 50us → 500us

# 持久化配置
cat >> /etc/sysctl.conf <<EOF
net.core.busy_poll = 0
net.core.busy_read = 0
EOF

# 验证修复效果
# Before fix (busy_poll=50):
#   IRQ_NONE rate: 10.2% (878/8567)
#   中断禁用: 每 2-5 分钟触发
#   TUN 丢包: 1,247 drops/30s
#   网络状态: 间歇性不通
#
# After fix (busy_poll=0):
#   IRQ_NONE rate: 0.49% (42/8567) ← 下降 95%!
#   中断禁用: 7 天无发生
#   TUN 丢包: 0 drops
#   网络状态: 完全稳定
```

关键经验:

1. 调用栈 100% 一致性: 立即排除随机因素,聚焦系统性原因
2. 队列级别隔离: 单队列问题快速缩小排查范围
3. 内核日志优先级: dmesg 的 "nobody cared" 是关键线索
4. 交叉验证外部数据: 发现第三方工具偏差,避免误判
5. 理解内核保护机制: IRQ storm protection 的阈值和触发条件
6. 时序分析: busy_poll 与中断的竞争关系
7. 用户态参数影响: sysctl 参数会显著影响内核行为
8. 工具组合模式: Summary → Details → 系统日志 → 参数验证
9. 双向交叉验证: 宿主机测量 ↔ VM 内部测量

10. 工具互补: BCC Python (灵活过滤) + bpftrace (快速原型)

案例对比总结:

维度	案例 1 (OVS 锁竞争)	案例 2 (virtio-net 中断风暴)
问题域	系统网络层 (OVS)	虚拟化网络层 (virtio/vhost)
表象	监控报告"丢包"	实际间歇性不通
真实问题	高延迟超时误判	中断被内核禁用
根因	OVS 流表锁竞争	napi_busy_poll 竞争中断
关键工具	offcputime + pthread_rwlock	virtnet_poll_stack_stats.bt
诊断层数	6 层	8 层
修复难度	中等 (需升级 OVS)	简单 (参数调整)
修复效果	P99.9 延迟改善 98%	IRQ_NONE 下降 95%
核心洞察	CPU 高 ≠ 计算密集	用户态配置 → 内核行为

---

5. 性能测试数据

5.1 关键性能数据点

本节展示标志性的性能测试数据,说明不同类型工具的性能特征:

1. Summary 工具: 性能开销极小,对 workload 影响极小

工具: system_network_perfomance_metrics.py

测试条件:

- 流量类型: TCP RX
- 数据包速率: ~120 万 PPS
- 测试时长: 30 分钟

测试结果:

指标	无工具基线	运行工具	变化	分析
CPU 使用率 (avg)	–	0.59%	+0.59%	极低开销
CPU 使用率 (峰值)	–	4.0%	+4.0%	峰值仍然很低
延迟影响	基线	-6.9%	提升 6.9%	性能轻微提升(误差范围)
吞吐量影响	基线	-0.33%	-0.33%	几乎无影响

结论: Summary 工具采用内核侧聚合策略,仅定期提交统计数据而非每包提交,因此性能开销极小,可持续运行。

2. Details 工具最坏情况: 大流量且所有包走完整路径并提交用户态

工具: trace_conntrack.py (无过滤器)

测试条件:

- 流量类型: TCP RX
- 所有包都执行完整 eBPF 测量逻辑
- 所有包都提交到用户态

测试结果:

指标	无工具基线	运行工具	变化	分析
延迟影响	82.31 us	139.46 us	+69.46%	显著延迟增加
吞吐量影响	基线	基线	-35.19%	吞吐量大幅下降

结论: 在最坏情况下(所有包都走完整逻辑并提交用户态),Details 工具对 workload 的性能影响最大。这种场景应避免在生产环境长期运行。

3. 过滤器灵活控制 eBPF 程序开销及对 workload 的影响

工具: eth_drop.py / system_network_latency_details.py (带过滤器)

过滤策略对比:

过滤条件	事件提交率	CPU 开销估算	对业务影响	适用场景
无过滤 (1M PPS)	1M evt/s	~50% CPU	高 (30%+)	禁止使用
IP 过滤 (特定源/目标)	~10K evt/s	~5% CPU	中 (5–10%)	短期诊断
五元组过滤	~1K evt/s	~0.5% CPU	低 (<2%)	针对性追踪
五元组 + 延迟阈值 (>100ms)	~100 evt/s	<0.1% CPU	极低	异常事件捕获

示例: system_network_latency_details.py --threshold 100000 (100ms)

```
# 只捕获延迟超过 100ms 的异常事件
sudo python3 system_network_latency_details.py \
  --phy-interface ens11 \
  --src-ip 10.132.114.11 \
  --dst-ip 10.132.114.12 \
  --direction rx \
  --protocol icmp \
  --latency-threshold 100000  # 100ms in microseconds
```

过滤机制原理:

- 在内核态尽早过滤,避免无效数据提交
- 协议过滤 → IP 过滤 → 端口过滤 → 阈值过滤
- 只有通过所有过滤器的包才会提交到用户态
- 绝大多数包在测量逻辑中提前返回

结论: 通过合理配置过滤条件,Details 工具的资源占用和对业务的影响可灵活调节,从而在精度和开销之间取得平衡。

4. 多阶段整合提交配合过滤条件/阈值,进一步调节开销

工具: system_network_performance_metrics.py / vm_network_performance_metrics.py

多阶段整合策略:

1. 在多个 probe 点收集数据 (6–8 个关键点)
2. 通过 skb 指针关联不同阶段
3. 在最后阶段整合所有信息

4. 仅提交一次完整事件 (而非每个阶段都提交)

效果对比:

策略	Probe 点数	提交次数/包	CPU 开销	数据完整性
每个 probe 都提交	6	6 次	高 (20%+)	完整
多阶段整合提交	6	1 次	低 (<5%)	完整

配合阈值过滤的增强效果:

```
// 仅在最后阶段检查阈值
if (total_latency < THRESHOLD) {
    // 清理中间状态
    delete_intermediate_data();
    return 0;  // 不提交
}

// 超过阈值才提交完整路径信息
submit_complete_event();
```

实际效果:

- 正常延迟的包 (99%+): 完全不提交,仅清理状态
- 异常延迟的包 (<1%): 提交完整路径信息
- CPU 开销降低 99%,同时保留完整诊断能力

结论: 多阶段整合提交是 Details 工具在保持精确测量能力的同时控制性能开销的关键技术。配合过滤条件和阈值,可以将工具开销降至极低水平。

---

6. Demo 演示

6.1 Demo 概述

本节演示三个单工具的典型使用场景,每个工具约 3–5 分钟:

1. kernel_drop_stack_stats_summary_all.py – 丢包分析

2. system_network_latency_summary.py – 延迟分析

3. ovs_userspace_megaflow.py – OVS Megaflow 追踪

6.2 演示脚本

Demo 1: 丢包定位 (3 分钟)

场景: 快速定位内核丢包位置

```
# 1. 启动丢包监控 (Summary 模式)
sudo python3 ebpf-tools/linux-network-stack/packet-drop/\
kernel_drop_stack_stats_summary_all.py \
  --src-ip 10.0.0.1 \
  --dst-ip 10.0.0.2 \
  --l4-protocol icmp \
  --interval 30 \
  --top 5

# 2. 生成流量
ping -c 100 10.0.0.2

# 预期输出:
# === Drop Stack Statistics ===
# Found X unique stack combinations:
#
# #1 Count: 15 [device: br-int]
#    Flow: 10.0.0.1 -> 10.0.0.2 (ICMP)
#    Stack:
#      kfree_skb
#      ip_rcv_core  ← 丢包位置
#      __netif_receive_skb_core
#
# 分析: IP 层丢包,可能是 TTL 或路由问题

# 3. 演讲要点:
# ● 自动聚合相同栈的丢包
# ● 精确定位到内核函数
# ● 关联网络流信息
# ● 低开销,可持续运行
```

演示要点:

- 自动栈聚合,按调用栈分类
- 精确到内核函数级别
- 流信息关联 (五元组)
- 极低性能开销

Demo 2: 网络延迟分析 (3 分钟)

场景: 识别网络延迟分布和长尾

```
# 1. 启动延迟监控 (Summary)
sudo python3 ebpf-tools/performance/system-network/\
system_network_latency_summary.py \
  --phy-interface ens11 \
  --direction rx \
  --protocol icmp \
  --interval 30

# 2. 生成正常流量
ping -i 0.1 10.0.0.2  # 10 PPS

# 预期输出 (正常情况):
# Stage: INTERNAL_RX → FLOW_EXTRACT_END_RX
#      latency (us)    : count    distribution
#         0 -> 1       :   12    |****                |
#         2 -> 3       :   45    |***************     |
#         4 -> 7       :   78    |****************************|
#         8 -> 15      :   65    |**********************  |  ← P50
#        16 -> 31      :   23    |********            |
#        32 -> 63      :   5     |*                   |
#
# Total packets: 228
# P50: ~10us, P95: ~40us, P99: ~60us

# 3. 模拟异常 (可选,如果环境允许)
# 在另一个终端制造 OVS CPU 压力:
stress-ng --cpu 4 --timeout 10s

# 预期输出 (异常情况):
# (histogram 中出现长尾)
#     1024 -> 2047    :   2     |                    |
#     2048 -> 4095    :   1     |                    |  ← 异常延迟!
#
# Packets with latency > 1ms: 3 packets

# 4. 演讲要点:
# ● Histogram 直观展示延迟分布
# ● 可清晰识别 P50/P95/P99
```

```
# ● 长尾延迟一目了然
# ● 分阶段测量,定位瓶颈在 OVS 层
```

演示要点:

- Histogram 可视化,直观易懂
- 自动计算 P50/P95/P99
- 长尾延迟识别
- 分阶段测量,精确定位瓶颈

Demo 3: OVS Megaflow 追踪 (3 分钟)

场景: 展示 OVS 内部工作原理的可观测性

```
# 1. 清空 OVS 流表 (制造 upcall)
sudo ovs-appctl dpctl/del-flows

# 2. 启动 Megaflow 追踪
sudo python3 ebpf-tools/ovs/ovs_userspace_megaflow.py \
  --ip-src 10.0.0.1 \
  --ip-dst 10.0.0.2 \
  --ip-proto 1

# 3. 触发流量
ping -c 3 10.0.0.2

# 预期输出:
# === UPCALL Event ===
# Time: [时间]
# Flow: 10.0.0.1 -> 10.0.0.2 (ICMP)
#
# === FLOW_CMD_NEW Event ===
# Time: [时间 + 数百微秒]
# Flow Key:
#   ipv4_src: 10.0.0.1
#   ipv4_dst: 10.0.0.2
#   ip_proto: 1
# Actions:
#   output: port 5
#
# Installation Latency: 234 us

# 4. 演讲要点:
# ● 完整追踪 Megaflow 生命周期
# ● Upcall → Userspace 决策 → 内核安装
# ● 测量 Installation Latency
# ● 可用于诊断 OVS 性能问题
```

演示要点:

- 黑盒系统 (OVS) 的内部可观测性
- Netlink 消息完整解析
- 延迟精确测量 (微秒级)

- 可用于流表策略分析

6.3 Demo 总结要点

向决策者强调:

1. **快速定位** ⏱️
    - 传统方法: 数小时 – 数天 (组合多工具 + 手动分析)
    - 本工具集: 分钟 – 小时 (结构化输出 + 自动关联)
    - **效率提升 10–100 倍**
2. **精确测量** 📊
    - 传统监控: 黑盒,只能看到宏观指标
    - 本工具集: 白盒,精确测量每个阶段耗时 (纳秒级)
    - **从"猜测"到"量化"**
3. **安全可控** 🛡️
    - eBPF 沙箱机制,不会崩溃内核
    - Summary 工具性能影响极小,可持续运行
    - Details 工具通过过滤器灵活控制影响
    - **生产环境可用**
4. **覆盖全面** 🌐
    - 30+ 工具覆盖虚拟化网络全链路
    - 丢包/延迟/OVS/虚拟化/CPU 全方位
    - Summary + Details 双版本策略
    - **一站式解决方案**
5. **易于使用** 🚀
    - 参数简单直观 (IP/端口/协议过滤)
    - 输出清晰 (Histogram + 详细事件)
    - 开箱即用,无需修改应用代码
    - **降低运维门槛**

---

7. 结论

7.1 核心成果

本项目成功构建了一套 **轻量、精确、易用** 的 eBPF 网络故障排查工具集:

核心创新点:

1. **三层诊断模型**
    - Summary (长期监控) + Details (精确追踪) + 专项工具 (根因分析)
    - 平衡性能开销与监控精度
2. **性能开销可控**
    - 基于 eBPF 性能模型的工程化实践
    - Summary: 极低开销,对业务性能影响极小
    - Details: 通过多级过滤和多阶段整合,资源占用灵活可调
    - 过滤器使绝大多数包在测量逻辑中提前返回
3. **全链路覆盖**
    - VM → vhost → TUN/TAP → OVS → Linux 栈 → 物理网卡
    - 30+ 工具,按层次、按问题分类
4. **精确测量**
    - 分阶段延迟测量 (纳秒级)
    - skb 指针关联,跨层追踪
    - 时间戳 + 调用栈,完整上下文
5. **实战验证**
    - 案例 1: ICMP "丢包"根因 (延迟误判 → OVS 锁竞争)
    - 性能测试: Summary 工具可持续运行,Details 工具开销可调节

应用价值:

- **降低 MTTR**: 从数小时降至分钟级
- **量化网络性能**: 建立各层次性能基线,精确定位瓶颈
- **降低运维成本**: 自动化故障定位,减少对专家依赖
- **赋能技术团队**: 深入理解虚拟化网络,提升技术能力

从"无法观测"到"精确测量"
从"经验猜测"到"数据驱动"
从"事后救火"到"主动监控"

eBPF: 让网络故障排查进入"可观测时代" 🚀

附录

A. 参考资料

1. 项目文档:
   - `troubleshooting-practice.md` – 实战案例详解
   - `docs/BCC_eBPF_Performance_Analysis.md` – 性能开销模型
   - `test/automate-performance-test/analysis/` – 性能测试数据与分析
2. eBPF 学习资源:
   - Brendan Gregg, "BPF Performance Tools" (2019)
   - Linux 内核文档: https://www.kernel.org/doc/html/latest/bpf/
   - BCC GitHub: https://github.com/iovisor/bcc
3. 虚拟化网络:
   - OVS 文档: https://www.openvswitch.org/
   - KVM/QEMU 虚拟化: https://www.linux-kvm.org/

B. 快速参考卡片

丢包问题排查:

```
# 1. 定位丢包位置 (Summary)
sudo python3 kernel_drop_stack_stats_summary_all.py \
  --src-ip <IP> --dst-ip <IP> --interval 60

# 2. 查看具体丢包包 (Details)
sudo python3 eth_drop.py \
  --src-ip <IP> --dst-ip <IP> --verbose
```

延迟问题排查:

```
# 1. 延迟分布 (Summary)
sudo python3 system_network_latency_summary.py \
  --phy-interface <IF> --direction rx --interval 60

# 2. 高延迟事件 (Details + 阈值)
sudo python3 system_network_latency_details.py \
  --phy-interface <IF> --direction rx \
  --latency-threshold 100000  # 100ms
```

OVS 问题排查:

```
# 1. Upcall 延迟 (Summary)
sudo python3 ovs_upcall_latency_summary.py --interval 60

# 2. Megaflow 生命周期 (Details)
sudo python3 ovs_userspace_megaflow.py \
  --ip-src <IP> --ip-dst <IP>
```

报告结束