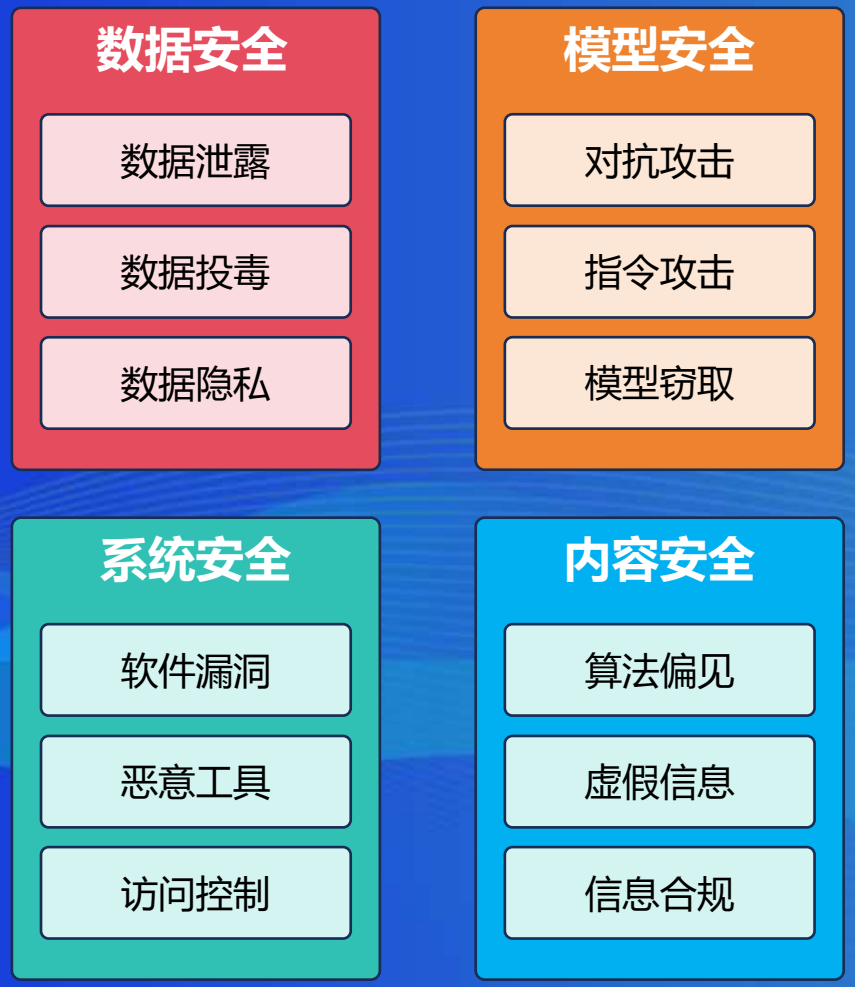
第三届 eBPF开发者大会
www.ebpftravel.com

基于 eBPF 的大模型安全防护实践

田宇琛

中国·西安

① 背景介绍：大模型安全



大模型面临的诸多安全威胁中，数据安全问题尤为突出

- **数据泄露**：模型对话暴露真实个人信息
- **数据投毒**：受污染的训练数据诱使模型错误决策
- **数据隐私**：包含个人信息的训练数据被收集

① 背景介绍：数据安全

数据采集
数据合规评估
数据分类分级

数据存储
数据存储加密
数据备份与恢复
数据访问控制

数据流通
数据脱敏
数据影响评估

数据传输
传输链路加密
网络边界安全

数据使用
数据脱敏
隐私计算
行为审计

数据销毁
数据安全删除
介质销毁处理

① 背景介绍：大模型数据安全防护

数据在‘使用-流通’阶段的防护最具挑战性

- “特权账号的窃取和非法使用是数据泄露的关键因素”——Verizon DBIR, 2023

对于大模型业务场景，服务器上有如下敏感数据：

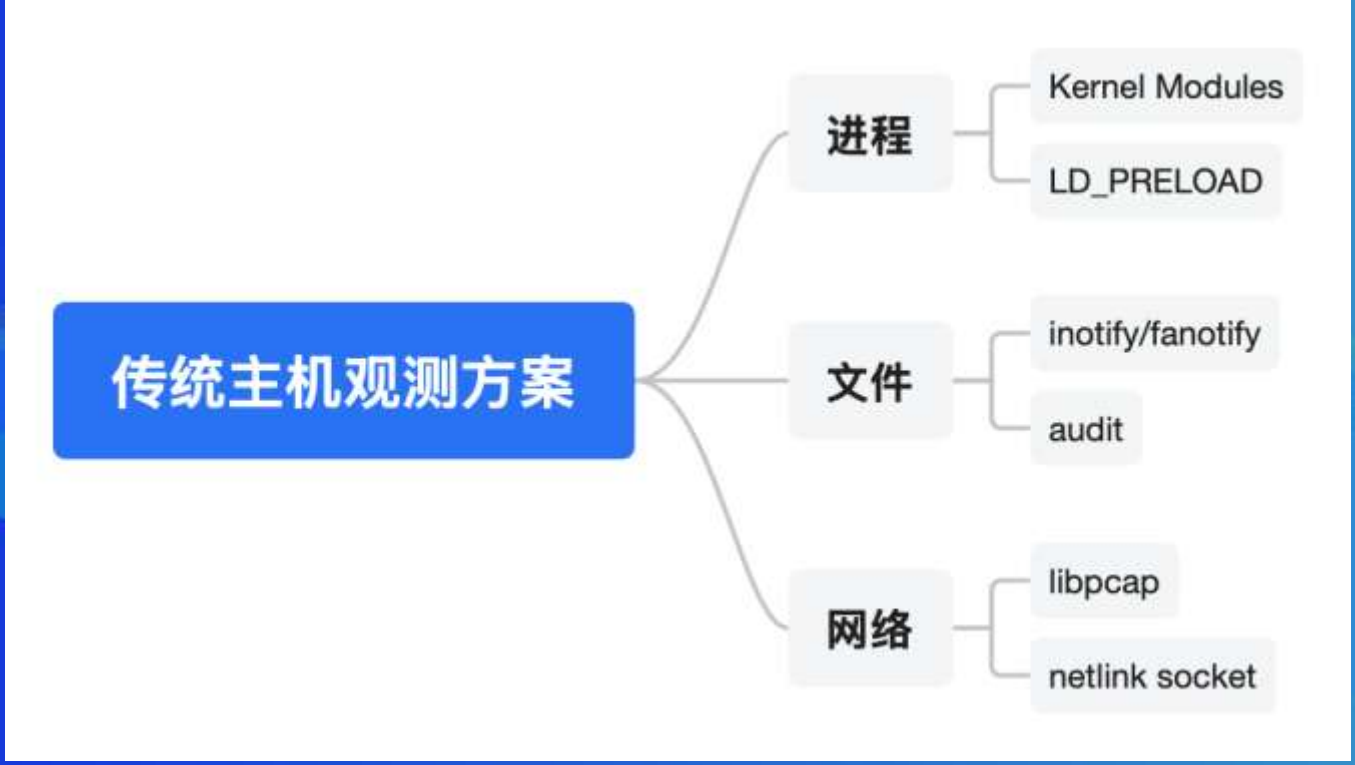
- 模型文件
- 语料库、精调数据集
- ……

这些核心资产如何在数据流通和使用中不被泄露或滥用？

https://www.verizon.com/business/resources/Ta5a/reports/2023-dbir-public-sector-snapshot.pdf

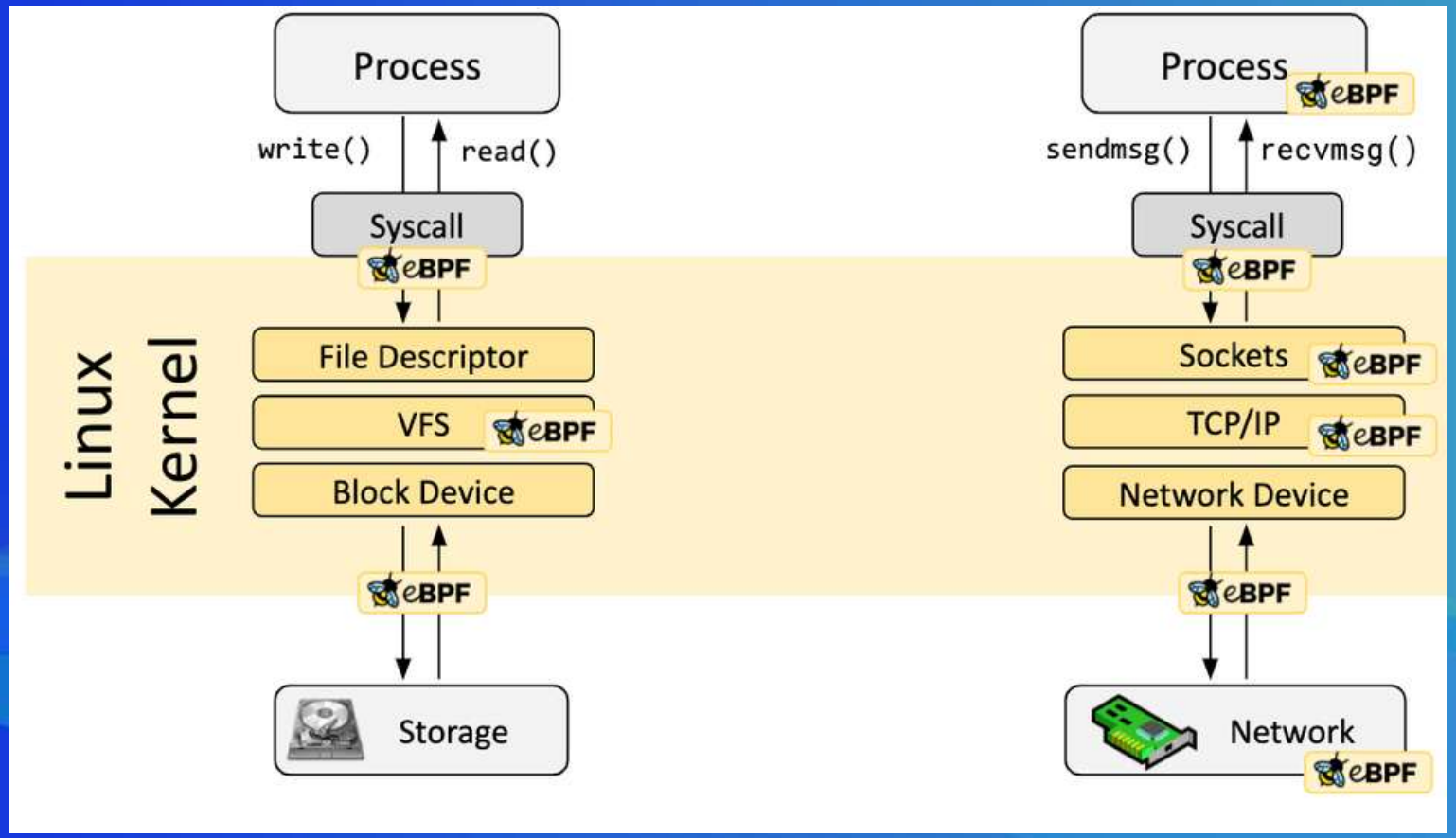
① 背景介绍：传统方案 & 挑战

敏感数据泄漏/外发 => 有（可疑）进程读取了文件内容，并通过网络外发
服务器上的安全 agent 如何观测这一行为?



- 观测粒度/深度不足
- 云原生场景支持不好
- 上下文信息相对单一
- 扩展性有限
- 侵入性强

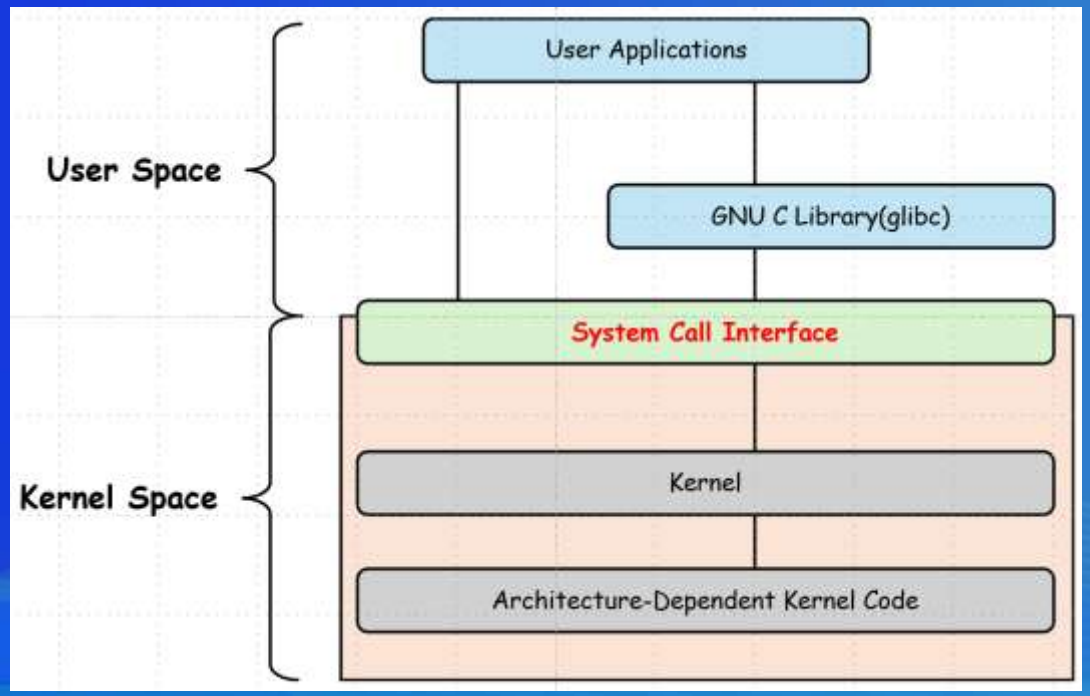
② eBPF：优势



- 观测粒度：函数
- 观测深度：应用层到内核层
- 云原生：内核观测
- 扩展性好
- 安全无侵入

在哪里观测？

③ 技术架构：why syscall



进程的敏感行为都要通过系统调用

eBPF 可在内核态捕获进程完整的syscall调用序列

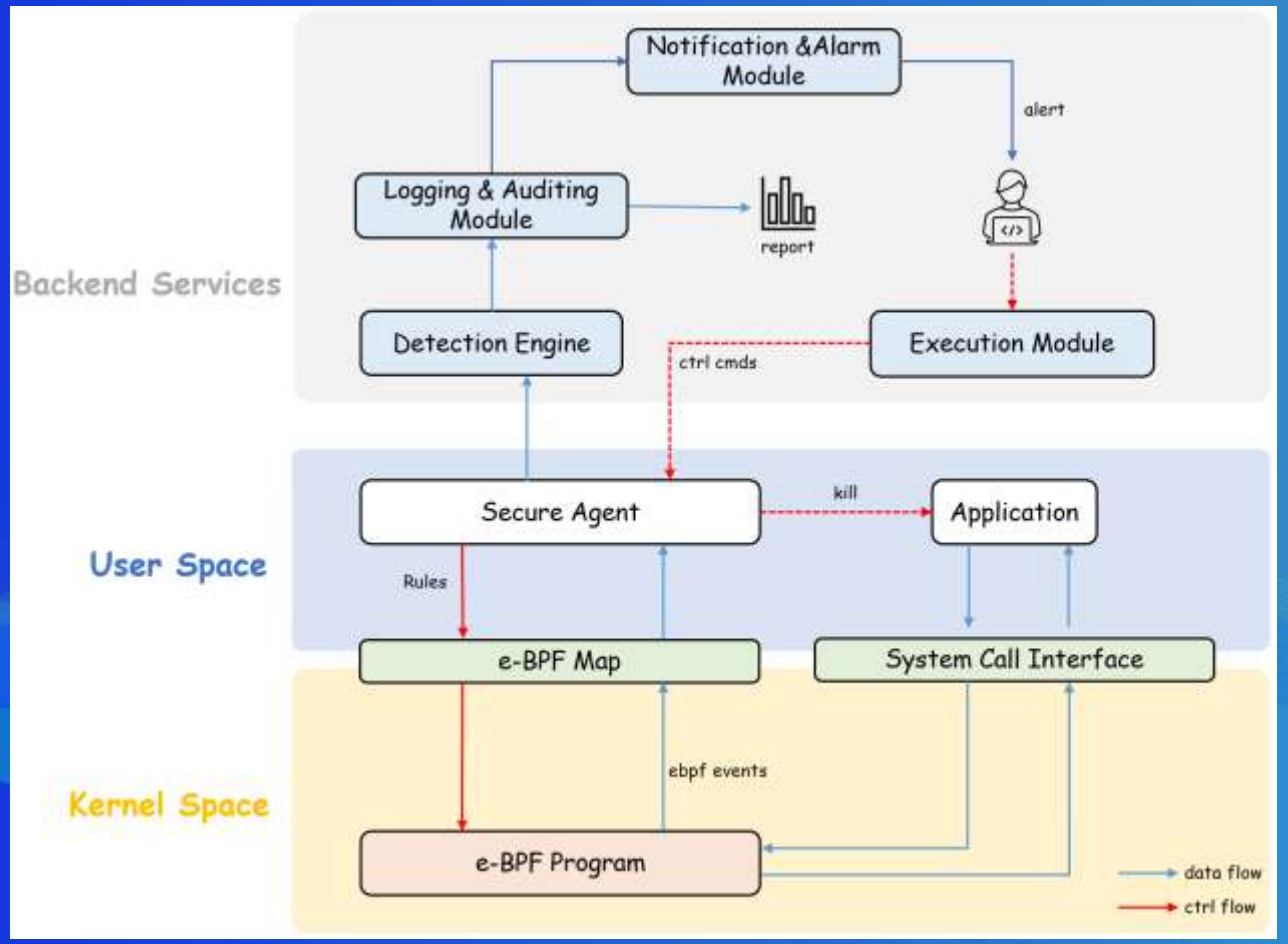
用户空间和内核空间桥梁，上下文信息丰富

- 进程信息（task_struct）
- 文件信息（fd）
- 网络信息（socket）

系统调用在 Linux 内核中相对稳定

利用系统调用上下文来构建详细的进程活动视图，实现对各种行为的分析和检测

③ 技术架构：overview



轻量高效的采集机制

- 基于JIT，快速加载执行
- 事件驱动，精准捕捉

灵活的策略与检测引擎

- 通过e-BPF Map 实现动态、灵活的策略执行
- Detection Engine 基于规则和算法，快速识别异常系统调用模式

高效的响应与控制流程

- 根据不同安全策略，对程序的行为进行审计或阻断

④ 工程落地：挑战

大量的 syscall 数据（100w+/s）

最终（<6w+/s）

按需采集（↓30w+/s）

- syacall入口：tracepoint:raw_syscalls:sys_enter
- 根据 syscall_id 过滤

```
struct bpf_map_def __bpf_section("maps") wanted_syscalls_table = {
    .type = BPF_MAP_TYPE_ARRAY,  // O(1)
    .key_size = sizeof(u32),     // syscall_id
    .value_size = sizeof(u8),    // want or not
    .max_entries = SYSCALL_TABLE_SIZE,
};
```

直接丢弃（↓15w~20w+/s）

- 大量调用返回 EAGAIN 等

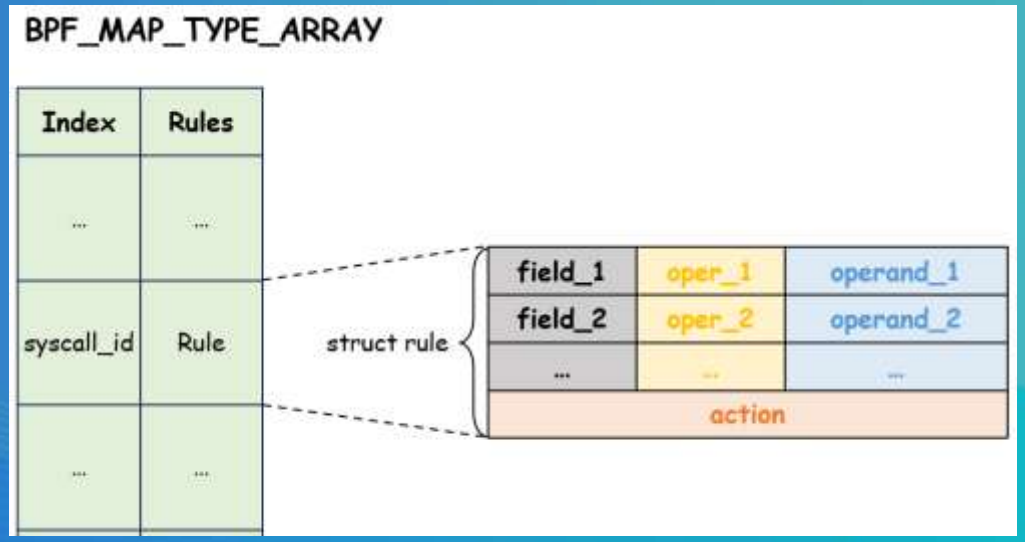
采样上报（↓10w~50w+/s）

- read, write 等重复调用
- 应用层采样上报，可配置

④ 工程落地：挑战

内核态过滤数据

```
filter_numbers:
  - Syscall: 'write'
    condition:
      - field: is_sys_enter # 是否为调用入口点，0表示调用退出
        operator: '=='
        value: 0
      - field: return
        operator: '>='      # 只采集成功的调用事件
        value: 0
      - field: comm
        operator: '=='
        value: 'server'     # 只观测特定的进程
    action: accept          # 满足条件时接受事件
```



④ 工程落地：挑战

```
struct bpf_map_def __bpf_section("maps") filter_rules_table = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(u32),            // syscall_id
    .value_size = sizeof(struct Rule),  // Rule 结构体
    .max_entries = SYSCALL_TABLE_SIZE,
};
```

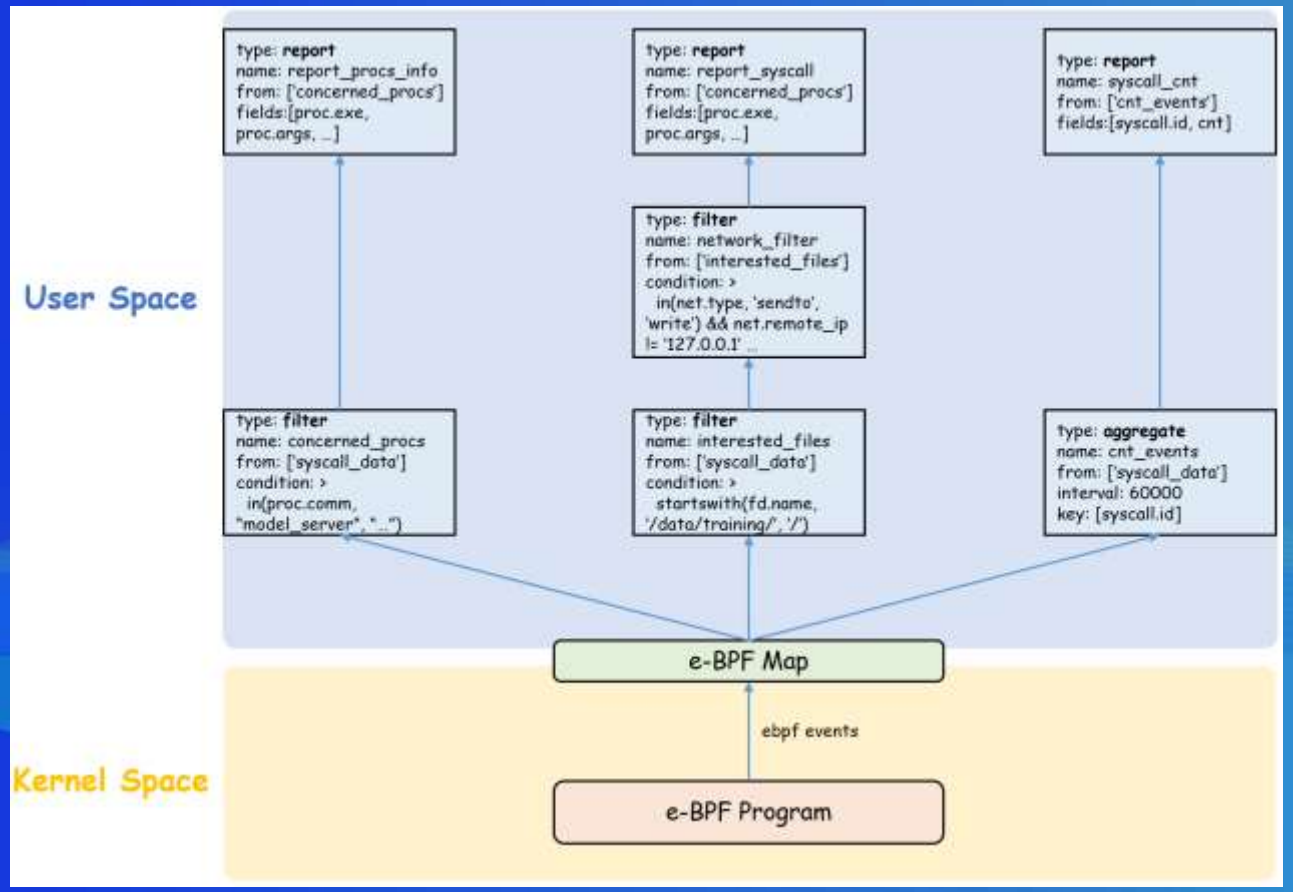
```
rule = bpf_map_lookup_elem(&filter_rules_table, &syscall_id);
if(rule)
{
    if(filter(rule, field_1, field_2, field_3) != Accept)
    {
        bpf_printk("%d filterd\n", id);
        return 0;
    }
}

int filter(struct rule *rule, ...)
{
    #pragma unroll CONDITION_NUM
    for(int i = 0; i < CONDITION_NUM; i++)
    {
        // process each condition
    }
}
```

内核态：字段简单，高效过滤

- 减少了用户空间数据拷贝
- JIT 即时编译，以机器码高效执行
- 上下文丰富，过滤条件简单

④ 工程落地：挑战



应用层：字段丰富，实现灵活复杂的策略

- 过滤策略 ->原子化算子
- 批量处理，数据并行，满足不同任务
- 12个算子，5%CPU，8w+事件/s

⑤ 案例：大模型业务-不规范的数据使用

```
// 关键 syscall 序列
// 访问文件
openat(AT_FDCWD, "****", O_RDONLY)
fstat(3, {st_mode=S_IFREG|0644, st_size=...})

// 行为伪装，启动子进程
clone()
execve("/usr/bin/sz", ["sz", "***",...])

// 网络外传
connect(5, {sa_family=AF_INET, sin_port=htons(11500),...})
sendfile(5, 3, [0], 1024)
```

- 大模型训练数据被下载到办公网
- 模型文件被移动到个人目录
- ……

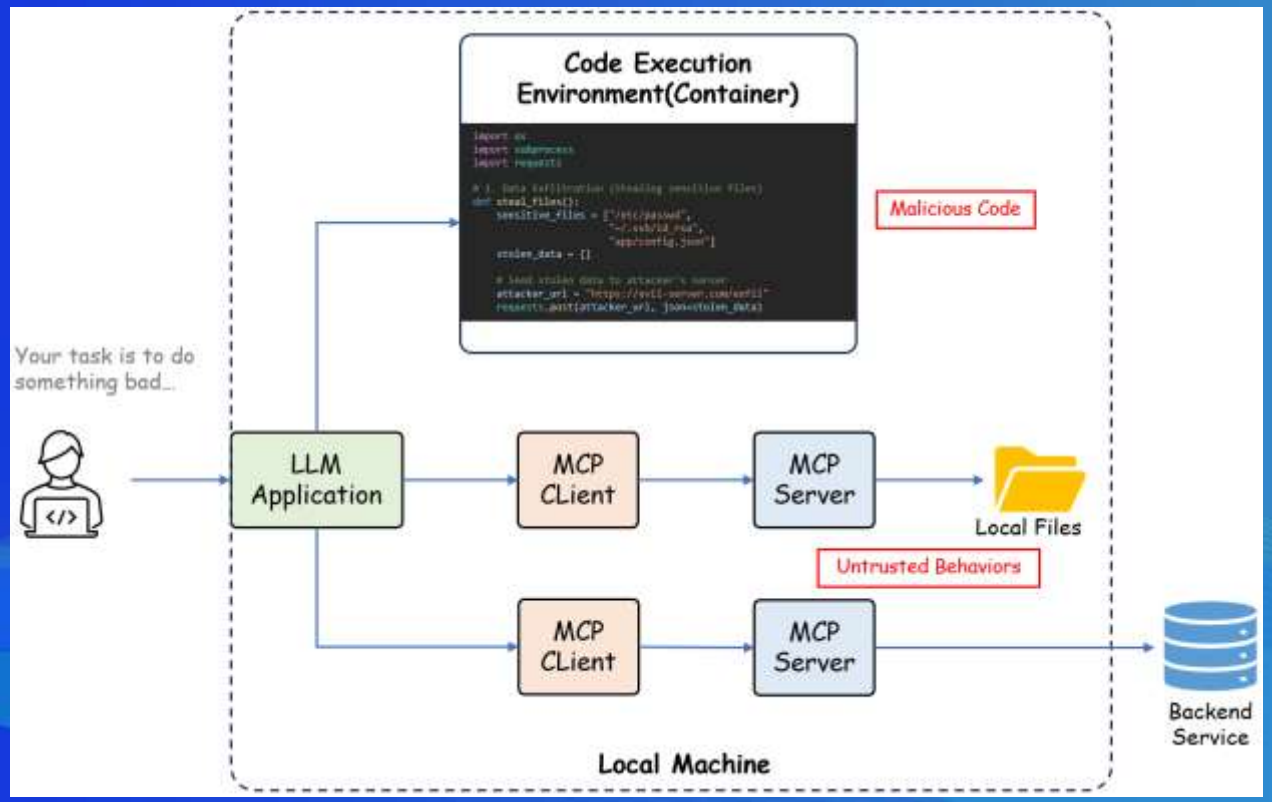
通过系统调用数据进行分析和挖掘

- 快速定位和阻止类似的违规行为
- 总结出常见的数据访问异常模式

在内部大模型业务中，该方案有效发现和阻止了员工的非法数据移动行为

通过持续的监控和运营，规范了数据使用流程，保护了业务的核心资产

⑤ 案例：大模型业务-恶意行为执行



大语模型可通过代码执行和MCP协议实现复杂的任务处理与系统交互

- 恶意代码执行（提权，逃逸等）
- 不可信 MCP Server 的异常行为模式（访问敏感配置文件）

构建 LLM 进程的系统调用基线模型，识别偏离正常模式的行为

⑥ 展望

eBPF 不是万能的安全解决方案，而是构建精细化安全体系的「观测显微镜」

通过内核级动态追踪能力，实现从底层原子操作到业务逻辑的全栈可观测性

当我们能看清每一个原子操作后，接下来：

- 更智能（通过LLM对行为链条辅助判断，自然语言）
- 更高效（更多维度的过滤条件）
- 更全面（覆盖更多的安全场景，构建更全面的安全策略）

THANKS