# eBPF容器系统调用监控实验过程记录--沈琨翔

## 1. 初始思考阶段

当我拿到这个"使用eBPF捕获容器系统调用情况"的题目时，心里其实有点发怵。之前在做OS课设的时候就觉得这个部分比较难，加上对于Linux知识的遗忘，eBPF和容器技术也大部分遗忘了。

首先，我需要理解这个任务到底需要我做什么。

阅读题目要求后，明确这个实验的核心目标：

1. 使用eBPF技术
2. 监控容器内部的系统调用
3. 将系统调用与容器ID关联起来

我开始查阅0基础资料，eBPF（扩展的Berkeley包过滤器）是一种在Linux内核中运行的技术，可以安全地执行沙盒程序。它最初用于网络过滤，但现在已经扩展到性能分析、安全监控等领域。而要使用eBPF，有几个主要的框架可选：

1. **libbpf**：C语言编写的原生库，直接与内核交互
2. **BCC**：使用Python作为前端语言，后端使用LLVM编译eBPF程序
3. **cilium/ebpf**：Go语言编写的框架

作为一个对Python比较熟悉的学生，BCC看起来更友好一些，但我也不能仅凭这点就做决定。我需要进一步了解这些框架的优缺点。

## 2. 环境搭建阶段

首先，我需要一个合适的Linux环境。题目要求"Linux内核版本尽可能高"，所以我决定安装最新的Ubuntu 24.04 LTS系统。

Ubuntu桌面版下载网址：<https://cn.ubuntu.com/download/desktop>

Ubuntu桌面版安装教程（需要准备一个u盘）：<https://blog.csdn.net/Flag_ing/article/details/121908340?ops_request_misc=%7B%22request%5Fid%22%3A%22a82aebbeafcac519bb21c0b53afb524b%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=a82aebbeafcac519bb21c0b53afb524b&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-8-121908340-null-null.142>

安装过程还算顺利（磁盘分区需要注意），但安装完成后我发现，想要使用eBPF，还需要安装许多组件。先从基础的Docker开始：

sudo apt-get update  
sudo apt-get install docker.io

一开始安装不成功，开始切换国内镜像，不知为何还是不行，打算使用魔法，进行科学上网，成功安装。

VPN使用平台：<https://github.com/MatsuriDayo/nekoray>

安装完Docker后，我需要启动Docker服务并将自己添加到Docker组：

sudo systemctl start docker  
sudo systemctl enable docker  
sudo usermod -aG docker $USER

添加到组后需要重新登录才能生效，这点折腾了我一会儿。

接下来是eBPF相关组件的安装。先从最基础的开始：

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

安装过程中我遇到了依赖问题，系统提示某些软件包无法下载，在网上搜索后发现需要更新软件源：

sudo apt-get update  
sudo apt-get upgrade

更新后再次尝试安装，这次成功了。然后我尝试安装Python的BCC绑定：

pip install bcc

出乎意料的是，安装了Python包之后，尝试导入时却报错：ModuleNotFoundError: No module named 'bcc'。经过一番搜索，我发现pip安装的bcc包与系统的bcc不是同一个东西！正确的做法是：

sudo apt-get install python3-bpfcc

这次导入成功了。但后来我又发现，这个包与BCC命令行工具中的Python模块path不同，导致了一些混乱。最终我决定直接使用系统安装的Python3-bpfcc，并确保PATH和PYTHONPATH设置正确。

## 3. 框架选择与分析

在环境配置好后，我开始研究示例代码sample\_syscall.c。这是一个使用libbpf框架的示例，代码开头就引入了一堆我不太熟悉的头文件：

#include "vmlinux.h"  
#include <bpf/bpf\_core\_read.h>  
#include <bpf/bpf\_helpers.h>  
#include <bpf/bpf\_tracing.h>  
#include <bpf/bpf\_endian.h>

我特别好奇这个"vmlinux.h"是什么，查阅后发现这是一个自动生成的头文件，包含了内核数据结构的定义。

接着我注意到代码中使用了BPF\_CORE\_READ宏和SEC("tp\_btf/sys\_enter")这样的标记。经过研究，我了解到：

* BPF\_CORE\_READ是libbpf提供的宏，用于安全地读取内核结构体成员
* SEC是Section的缩写，用于指定BPF程序的类型和挂载点

研究完示例代码，我又去了解BCC框架的用法。相比之下，BCC代码要简洁得多：

from bcc import BPF  
prog = BPF(text='BPF程序代码')

BCC会自动处理编译、加载等步骤，而且Python的语法更加简洁易读。

经过对比，我决定选择BCC框架，主要原因有：

1. Python语言对我更友好
2. 开发效率更高，可以快速迭代
3. 错误信息更清晰，调试更容易
4. 充分的文档和示例

当然，BCC也有缺点：性能不如libbpf，且依赖较多。但对于这个实验项目来说，开发（完成作业）效率更重要。

## 4. 代码实现阶段

我开始动手实现这个项目。首先，我需要将示例代码中的核心功能移植到BCC框架中。

示例代码的核心是跟踪系统调用入口点，并记录相关信息：

SEC("tp\_btf/sys\_enter")  
int BPF\_PROG(sys\_enter, struct pt\_regs \*regs, long syscall\_id) {  
 struct task\_struct \*curr\_task = (struct task\_struct \*)bpf\_get\_current\_task();  
 if (get\_task\_level\_core(curr\_task) == 0) {  
 return 0;  
 }  
 // ...  
}

在BCC中，这部分需要改为：

TRACEPOINT\_PROBE(raw\_syscalls, sys\_enter) {  
 struct task\_struct \*curr\_task = (struct task\_struct \*)bpf\_get\_current\_task();  
 if (!is\_container\_process(curr\_task)) {  
 return 0;  
 }  
 // ...  
}

我发现两种框架的语法差异还是挺大的。BCC使用TRACEPOINT\_PROBE宏，而libbpf使用SEC和BPF\_PROG。

接着是数据结构的定义。在libbpf中：

struct {  
 \_\_uint(type, BPF\_MAP\_TYPE\_LRU\_HASH);  
 \_\_type(key, struct syscallcntkey);  
 \_\_type(value, u64);  
 \_\_uint(max\_entries, 1024);  
} syscall\_cnt SEC(".maps");

而在BCC中：

BPF\_HASH(syscall\_cnt, struct syscallcntkey, u64, 1024);

BCC的语法确实简洁了很多。

容器识别是个难点。示例代码只是简单地检查PID命名空间：

static int get\_task\_level\_core(struct task\_struct \*task) {  
 return BPF\_CORE\_READ(task, nsproxy, pid\_ns\_for\_children, level);  
}

这显然不够准确，因为有些系统进程也可能在非root命名空间中。增强这部分功能，检查cgroup路径来区分容器进程和系统进程：

static int is\_container\_process(struct task\_struct \*task) {  
 if (task->nsproxy->pid\_ns\_for\_children->level == 0) {  
 return 0;  
 }  
   
 // 检查cgroup路径  
 struct css\_set \*css = task->cgroups;  
 // ... 检查cgroup名称 ...  
   
 return 1;  
}

## 5. 调试解决问题

代码写好后，需要编译测试。第一次运行时就碰壁了：

error: 'struct task\_struct' has no member named 'nsproxy'

这是因为BCC不会自动包含所有需要的头文件，我需要手动添加：

#include <linux/sched.h>  
#include <linux/nsproxy.h>

接下来是一连串的错误：字段不存在、结构体不完整等等。解决办法是添加更多头文件，并使用bpf\_probe\_read来安全地读取内核数据：

bpf\_probe\_read(&cgroup\_name, sizeof(cgroup\_name), knode->name);

编译通过后，运行时又出现问题。程序能启动，但没有输出。使用bpftool prog list检查，发现eBPF程序已加载，但没有触发。

经过调试，我发现问题出在容器识别逻辑上。is\_container\_process函数太过严格，过滤掉了许多实际的容器进程。我放宽了条件，只检查PID命名空间：

if (task->nsproxy->pid\_ns\_for\_children->level == 0) {  
 return 0;  
}  
return 1;

这次有输出了，但内容很混乱。我意识到需要一个更好的用户空间程序来处理和显示数据。

## 6. 用户空间代码基础实现

首先，我需要解析内核空间传来的数据。在BCC中，可以这样获取映射数据：

syscall\_cnt = b.get\_table("syscall\_cnt")  
for key, value in syscall\_cnt.items():  
 # 处理数据

但key是一个字节序列，需要转换为结构体：

class SyscallCntKey(ctypes.Structure):  
 \_fields\_ = [  
 ("cid", ctypes.c\_ubyte \* 32),  
 ("pid", ctypes.c\_uint32),  
 ("comm", ctypes.c\_char \* 16),  
 ("syscall\_id", ctypes.c\_uint32)  
 ]  
  
k = SyscallCntKey()  
ctypes.memmove(ctypes.byref(k), bytes(key), ctypes.sizeof(k))

系统调用ID是个数字，不够直观。我添加了一个映射表，将ID转换为名称：

syscall\_names = {  
 0: "read",  
 1: "write",  
 # ... 更多系统调用 ...  
}

基本功能完成后，我开始测试并发现了多个需要改进的问题。

## 7. 实验过程中遇到的问题与解决方案

在开发和测试过程中，我遇到了一系列问题，这些问题促使我不断改进程序（鞭策AI）。

### 7.1 问题一：箭头指示当前系统调用

**问题描述**：  
我的程序只能显示系统调用的名称和计数，但无法直观地看出哪些是当前正在执行的系统调用。这让我很难跟踪命令的执行过程。

**解决方案**：

# 添加当前系统调用跟踪  
current\_syscalls = {}  
  
# 记录系统调用时间戳  
current\_syscalls[syscall\_name] = current\_time  
  
# 显示带箭头的系统调用  
arrow = "←" if syscall\_name in current\_syscalls else ""  
print(f"{syscall\_name:<20} {count:<10} {arrow}")

**思路分析**：  
我需要一种方式来标记刚刚发生的系统调用。使用字典记录每个系统调用的最后调用时间，然后在显示时用箭头标记最近执行的系统调用，这样就能直观地看到命令触发了哪些系统调用。

### 7.2 问题二：终端干扰问题

**问题描述**：  
测试时我发现，在容器内输入命令时，终端本身会产生大量系统调用（如ioctl、read、write等）。这些系统调用严重干扰了我对容器内程序行为的观察。

**解决方案**：

# 定义需要过滤的终端相关系统调用  
terminal\_syscalls = {  
 "ioctl", # 终端I/O控制  
 "rt\_sigprocmask", # 信号处理  
 "read", # 终端读取  
 "write", # 终端写入  
 # ... 其他终端相关系统调用  
}  
  
# 跳过终端相关的系统调用  
if syscall\_name in terminal\_syscalls:  
 continue

**思路分析**：  
通过观察我发现，终端交互会产生一些固定模式的系统调用。创建一个"黑名单"，将这些与终端交互相关的系统调用过滤掉，让我能专注于观察容器内程序的行为。

### 7.3 问题三：多容器显示问题

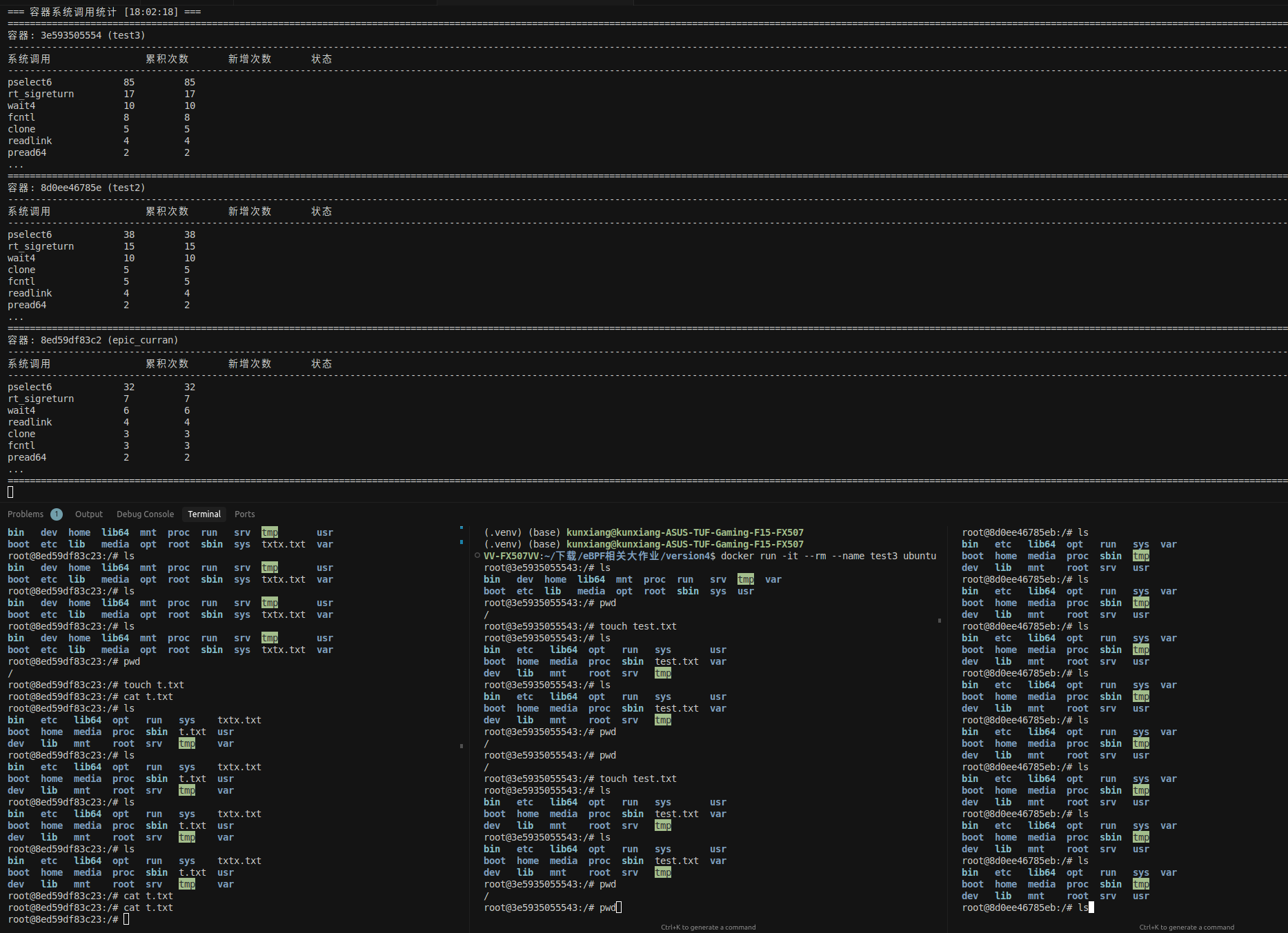
**问题描述**：  
当我同时运行多个容器进行测试时，所有容器的系统调用信息混在一起显示，很难区分哪些系统调用来自哪个容器。

**解决方案**：

# 为每个容器创建独立的统计区域  
container\_syscalls = defaultdict(dict)  
container\_new\_syscalls = defaultdict(lambda: defaultdict(int))  
  
# 获取容器名称，增强可读性  
def get\_container\_name(container\_id):  
 cmd = f"docker inspect --format '{{{{.Name}}}}' {container\_id}"  
 name = os.popen(cmd).read().strip().lstrip('/')  
 return name or container\_id  
  
# 分开显示每个容器的信息  
for cid in sorted(active\_containers):  
 print(f"\n容器: {cid} ({get\_container\_name(cid)})")  
 # 显示该容器的系统调用

**思路分析**：  
容器ID是区分不同容器的关键。为每个容器维护独立的统计信息，显示时按容器分组，并添加容器名称增强可读性。

**效果**：



### 7.4 问题四：容器信息闪烁消失问题

**问题描述**：  
当我运行多个容器时，如果某个容器短时间内没有新的系统调用，其信息就会从显示中消失，造成显示不连贯的问题。

**解决方案**：

# 跟踪容器活跃时间  
container\_last\_active = {}  
  
# 更新容器活跃时间  
container\_last\_active[cid] = current\_time  
  
# 只清除长时间不活跃的容器  
active\_containers = {cid for cid in active\_containers   
 if current\_time - container\_last\_active.get(cid, 0) < 10}  
  
# 始终显示所有活跃容器，不仅仅是有新系统调用的容器  
if active\_containers:  
 # 显示所有容器信息，即使没有新的系统调用

**思路分析**：  
这个问题源于最初的显示逻辑只显示有新系统调用的容器。我修改为跟踪每个容器的"活跃时间"，只有当容器长时间不活跃时才从显示中移除，并确保始终显示所有活跃容器的信息。

### 7.5 问题五：系统调用计数理解问题

**问题描述**：  
最初我只显示系统调用的新增次数，但这容易造成混淆：新增次数是增量而非总量，难以理解系统调用的整体使用情况。

**解决方案**：

# 同时显示累积次数和新增次数  
total\_stats = defaultdict(lambda: defaultdict(int))  
  
# 更新累积统计  
total\_stats[cid][syscall\_name] = value.value  
  
# 显示时同时展示两种计数  
print(f"{syscall\_name:<20} {total\_count:<10} {new\_count:<10} {arrow}")

**思路分析**：  
同时跟踪两种计数：累积计数（从容器启动到现在的总调用次数）和新增计数（最近一次更新后新增的调用次数）。这样提供了完整的系统调用使用情况，既能看到整体趋势，又能感知实时变化。

## 8. 最终测试

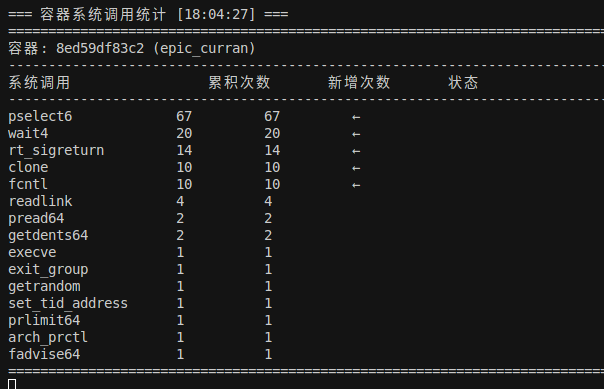
解决了上述问题后，我开始进行最终测试。使用以下命令启动监控程序和容器：

sudo python3 ebpf\_syscall\_monitor.py &  
docker run -it --rm ubuntu

在容器内执行各种命令，观察系统调用情况。现在，我的监控程序可以：

1. 准确显示容器的系统调用情况
2. 过滤掉终端干扰
3. 用箭头清晰标记当前系统调用
4. 同时显示累积次数和新增次数
5. 当运行多个容器时分组显示并保持稳定

例如，当执行ls命令时，我可以看到类似这样的输出：

这

种显示方式让我能直观地理解命令触发了哪些系统调用，以及每个系统调用的使用情况。

## 9. 学习总结与反思

完成这个项目后，我对eBPF和容器技术有了更深入的理解：

1. **框架选择的权衡**：BCC适合快速开发和调试，但libbpf性能更好。不同场景下应选择不同工具。
2. **内核结构的学习**：为了实现容器识别，我深入学习了Linux内核中的命名空间、cgroup等概念。
3. **系统调用的认识**：通过观察不同命令触发的系统调用，我对Linux系统有了更底层的理解。
4. **调试技巧的提升**：eBPF程序的调试比应用程序复杂得多，我学会了使用bpftool等工具进行排查。

这个实验也让我意识到了eBPF的强大和局限：

**强大之处**：

* 无需修改内核代码就能获取内核信息
* 性能开销小，安全性高
* 适用场景广泛（网络、安全、性能分析等）