大作业-结题报告

陈思睿　梁恒宇　吕泓涛　汤力宇
中国科学技术大学　安徽合肥

2021 年 7 月 15 日

目录

1 简介 **2**
2 项目背景 **2**
2.1 linux 由内核提供的沙盒工具 2
2.2 gVisor 为例的用户态沙盒分析 3
2.3 恶意进程 4
2.3.1 linux 下的恶意进程 4
2.3.2 经典攻击方法 4
2.3.3 恶意进程行为 4
2.3.4 小结 5
2.4 docker 安全 5
2.5 macOS 沙盒分析 6
2.5.1 macOS 沙盒的权限设置 6
2.5.2 应用程序沙盒化 6
2.5.3 沙盒机制 7
2.5.4 macOS 沙盒优缺点 9
2.5.5 小结 9
3 项目介绍 **9**
3.1 项目设计思路 9
3.2 整体结构 10
3.3 可载入模块 10
3.4 内核的修改 10
3.5 loader 11
4 项目实施过程的记录 **11**
4.1 BPF 的相关尝试 11
4.2 内核模块方式的实现 11

5 功能演示与性能测试 12
5.1 功能演示 12
5.2 性能测试 13
5.3 项目总结 14

1 简介

沙盒程序是用来保护计算机系统，防止潜在的恶意进程成功实施攻击的一类计算机应用。

当前现有的沙盒程序主要包括内核内置的沙盒工具与用户态沙盒程序两大类，两种实现方式都存在着较大的缺点。内核沙盒工具有着难以开发、难以调试、难以提供用户的自由性的问题。用户态沙盒程序则难以逃避多次系统调用带来的性能损耗。

本项目提出一种新的沙盒实现结构，使沙盒程序的结构横跨内核态与用户态，在精简沙盒结构的同时提高了沙盒程序的自由度，降低了沙盒程序的开发难度，并且实现了接近原生的沙盒性能。

2 项目背景

2.1 linux 由内核提供的沙盒工具

- **seccomp**
  seccomp介绍。传统 seccomp 可以让某个进程进入安全模式，并且阻止其调用所有除了 exit()，sigreturn()，read()，write() 外的所有对 file descriptor (FD) 的调用，一旦发现此类调用就立即 kill 进程。后来引入 bpf 后形成了 seccomp/bpf，可以更灵活的设置拦截的规则。seccomp 使用的是 cBPF [1]，seccomp/bpf 以 bpf 程序的形式定义了一个系统调用的白名单，被 seccomp/bpf 约束的进程可以自由的请求白名单内的系统调用，但一旦进程发出了不许可的指令，它将会被立刻终结。这种沙盒工具只是利用 BPF 限制了系统调用，沙盒内外的程序运行环境没有其他区别。[2]

- **Cgroups**
  Cgroups介绍。其主要支持四个功能：

  1. 限制某些 groups 的资源使用量 (包括 CPU，IO，内存等)
  2. 调整优先级，使某些 group 可以分配到更多的资源
  3. 记录某些 group 的资源使用量
  4. 控制某些 group 的运行，具体的可以冻结，快照，重启。

- **Linux Namespace**
  Linux 的 Namespace 机制是一种资源隔离方案。它将 Linux 的全局资源,划分为 namespace 范围内的资源，而且不同 namespace 间的资源彼此透明，不同 namespace 里的进程无法感知到其它 namespace 里面的进程和资源。但是 namespace 机制有缺陷。

  1. Non-namespace-aware system 调用接口可帮助对手破坏容器中运行的应用程序，并进一步利用内核漏洞来提升权限，绕过访问控制策略并逃过隔离机制。[3]

2. 某种沙盒使用的安全方案 [3]：用自动测试锁定一个 container 中程序的系统调用。然后在实际运行中阻止非锁定的任何系统调用。但是锁定系统调用不算特别理想。程序调用的是 API，间接执行系统调用，不容易直接发现。有的程序将近一半的可能系统调用都没被发掘。对于开发者来说，可以加入自己的测试过程来辅助锁定。

2.2 gVisor 为例的用户态沙盒分析

gVisor 是一款新型沙箱解决方案，其能够为容器提供安全的隔离措施，同时继续保持远优于虚拟机的轻量化特性。gVisor 能够与 Docker 及 Kubernetes 实现集成，从而在生产环境中更轻松地建立起沙箱化容器系统。

gVisor 能够在保证轻量化优势的同时，提供与虚拟机类似的隔离效果。gVisor 的核心为一套运行非特权普通进程的内核，且支持大多数 Linux 系统调用。该内核使用 Go 编写，这主要是考虑到 Go 语言拥有良好的内存管理机制与类型安全性。与在虚拟机当中一样，gVisor 沙箱中运行的应用程序也将获得自己的内核与一组虚拟设备——这一点与主机及其它沙箱方案有所区别。（gVisor 主页）

gVisor 的结构与问题

论文 [4] 提出来了如下观点：

1. 传统来说,hyperviser 模式的虚拟化容器有着更好的安全性,但是难以保证性能。hostOS 结构的容器 (如 docker) 的性能更好是由于其运行的若干的虚拟机通过一个统一的完善的通用 OS 来调度各类资源。但是由于 hostOS 结构中 hostOS 本身没有运行在容器中，其本身的内核 bug 容易成为被攻击的目标。(详细分析见此文Are Docker containers really secure? [5])

2. gviser 的存在一些性能问题，如打开关闭文件比传统容器慢了 216 倍，其他操作也普遍慢了很多（2 倍到 11 倍）。

3. gVisor 支持 OCI(Open Container Initiative)，因此 docker 用户可以自己配置使用默认引擎，runc 或者 gvisor 作为 runtime engine。

4. gVisor 结构如下 guestApp-Sentry(VMM+guestOS(linux))-hostOS，多层结构确保程序难以同时攻克每一层的安全缺陷，损害 hostOS 的安全。sentry 提供两种工作模式，第一种模式中其追踪并且翻译 gusetAPP 的系统调用，第二种模式中其更像是虚拟机中工作的 guestOS，直接服务 guestAPP。

5. gVisor 为 guestAPP 提供了 211 个 syscall(标准 linux 提供了 319 种)，gVisor 只需要向 hostOS 请求 55 种 syscall，这些 syscall 的种类都是通过 seccomp 技术限制和约束的，当 sentry 被 guestAPP 劫持并且申请了超出允许范围的 syscall 时，seccomp 过滤器会把 gVisor 杀死从而确保 hostOS 的安全。诸如 OPEN 和 SOCKET 这样的操作被设计者认为是极端危险的，因此没有被列入许可的 syscall，这两个功能是通过复杂的结构设计出来的，从而保证可以在不调用 hostOS 的对应 syscall 的前提下安全的为 guestAPP 提供服务。这就是为什么 gVisor 的文件性能如此差。

6. gVisor 对文件服务的实现：

   - sentry 实现了若干个不同的内置文件系统来尽可能满足 guestAPP 的请求。

- 当有必要读取 hostOS 文件系统时，他调用 Gofer 来替其进行文件访问，访问结果（文件句柄）通过一个 P9 Channel 返回给 sentry（进程间通讯），
- sentry 得到句柄后需要进行用户态到内核态的转化和上下文切换才能进行读取。

此过程导致了 gvisor 在文件访问效率上的低速，但带来了可靠的安全性。

7. 本文提出了若干种对此类容器进行性能测试的方法。

2.3 恶意进程

2.3.1 linux 下的恶意进程

Linux 环境下的经典病毒种类有：[6]

- BillGates 基于僵尸网络的 DDOS 攻击
- DDG 蠕虫式挖矿
- SystemdMiner 蠕虫式挖矿
- StartMiner 蠕虫式挖矿
- WatchdogsMiner 蠕虫式挖矿
- XorDDos 基于僵尸网络的 DDOS 攻击
- RainbowMiner 蠕虫式挖矿

2.3.2 经典攻击方法

在这几篇文章 [7] [8] 中介绍了几种经典的恶意进程攻击手段：

- 操作系统中的一个用户态组件——动态装载器，负责装载二进制文件以及它们依赖的库文件到内存中。二进制文件使用动态装载器来支持导入符号的解析功能。有趣的是，这恰好就是一个面对加固应用的攻击者通过泄漏库地址与内容尝试“重塑”一个符号的表现。windows 下的 svchost.exe 攻击或者 linux 下的 elf 攻击都是利用了这个组件进行的攻击。
- 早期的栈溢出利用依赖于向缓冲区中注入二进制代码 (称为 shellcode) 的能力，并需要覆盖在栈上的一个返回地址使其指向这个缓冲区。随后，当程序从当前函数返回时，执行流就会被重定向到攻击者的 shellcode，接着攻击者就能取得程序的控制权。

2.3.3 恶意进程行为

卡巴斯基安全实验室在对一个新型 zeus 变种木马的报告中给出了如下的分析：[9]

- 此木马功能非常丰富，包括使用 VNC 远程桌面控制电脑，截屏并发送，读取本地数据并发送，读取用户所有操作并发送，通过注册表开机自动启动，监测是否处于沙盒环境并且在沙盒中自动停止活动，监测注册表以防止自己的自动启动被清除。

- 此木马的传播主要通过邮件，在恶意邮件中包含一个 doc 文件作为附件，打开后 office 会提示需要启用宏来查看完整信息，一旦用户点击启用宏后，此木马会自动解码并且将自己复制进 svchost.exe ，然后其遍可以开始运行各种功能，包括剩余模块的下载，劫持浏览器，窃取本地数据等。
- 此木马通过读取进程列表以判断是否有正在打开的浏览器，然后当存在浏览器的时其会通过浏览器的安全漏洞劫持网页，修改用户正在访问的银行网页，截取其输入的密码、账号、pin、等内容并且发送给服务器。
- 此木马对沙盒有有特殊的监测机制，运行在沙盒内的时候系统中会出现若干特征性的监控类设备驱动，当发现自己运行在沙盒中时，木马将停止活动以防止自己被安全人员监测出来。

2.3.4 小结

此分析给我们带来了如下的提示：

- 针对某个具体的漏洞来设计沙盒是不切实际的，随着系统体量的膨胀，漏洞的存在是不可避免的，至今 linux 和 windows 也无法完全杜绝对系统代码的恶意篡改。
- 各类资源的隔离对于沙盒的安全性至关重要，必须确保沙盒内的进程只能访问有限的文件服务或者系统服务。
- 任何需要与被隔离进程直接交互的服务都有被污染的可能，因此多层的隔离或者使用类似于 BPF 的无法污染的实现方式可以有效的提高对恶意进程的控制力度。

2.4 docker 安全

下面是 docker 安全性的具体实现方式 [10] [11]

1. 通过 namespace，不同 docker 容器无法访问其他的进程，在容器位置向系统请求进程列表会只能看到少数几个局部的容器内进程，无法发现主机的其他进程。通过这种技术，类似于 zeus 的劫持浏览器进程的木马难以危害主机安全。
2. 通过 namespace，每个 docker 会被置入隔离的网络环境中，对外的网络功能是通过在每个 docker 上运行虚拟的网卡并且以桥接模式（默认）与主机网卡链接来实现的。在这种情况下可以通过网络安全策略的方式直接控制容器进程的非法网络访问。
   libnetwork：docker 的网络功能实现的具体技术
3. 利用 libcontainer（以及 namespace）来实现了对文件系统的保护，libcontainer 中的 chroot 技术可以限制某个子系统对应的根目录（rootFS），即在容器内的进程来看，当前 FS 的 root 就是实际所在的子目录，因而其无法读取或访问主机上的其他文件。
4. cgroup（控制组）是用于限制进程对 CPU、内存、网络带宽等运行资源的占用强度的，其也可以用来限制容器内程序对设备的访问。不同的进程被组合成一个 cgroup，作为一个整体参与资源的调度，并且可以通过 cgroup 组策略来限制当前 group 可以占用多少资源。且 cgroup 可以嵌套，一个 cgroup 里面可以包含多个子 cgroup。如整个 docker 可能被放在一个 cgroup 中以限制总资源使用量，然后 docker 里面的每个容器中的进程也各自建立 cgroup，参与划分 docekr-group 分配到的总的资源。

5. 联合文件系统（Unionfs），实质上概念很简单，此文件系统不管理物理存储，只是依赖于某一个通用的文件系统，并且把不同文件夹的内容映射到同一个文件目录内。似乎是 docker 的重要组成部分。

在这篇文章中，分析了 docker 结构的安全性：[12]

- 文章基于的模型如下：当 hostOS 中运行的 docker 容器中有一部分被恶意进程完全控制了，其可以对系统进行如 Denial-of-Service 和 Privilege escalation 的攻击。
- 为了在这种情况下保护系统安全，容器应当做到如下几点：
  - process isolation 进程间的隔离
  - filesystem isolation 文件系统的隔离
  - device isolation 设备的隔离
  - IPC isolation 进程间通讯的隔离
  - network isolation 网络的隔离
  - limiting of resources 限制资源的使用量

对于路线一或路线三，我们可以参考 docker 的安全策略，此策略在各个角度上都有较好的安全性，而且性能相当的高。

2.5 macOS 沙盒分析

2.5.1 macOS 沙盒的权限设置

macOS 沙盒中有如下的权限设置 [13]：

- Essentials：是否使用访问控制技术
- Network：接收和发送网络数据的权限
- Hardware：硬件权限，如照相机
- App Data：访问地址簿，位置，日历的权限
- File Access：其中设置是否拥有访问系统全部文件的权限已被剥离。读写限制十分严格，只有用户主动操作才能实现一般文件的读写（通过脱离程序语境的 Powerbox 机制——一个选择文件的对话框），否则只有应用程序自己的目录是可见的。下载内容、图片、音乐、电影各有特殊限制规则。

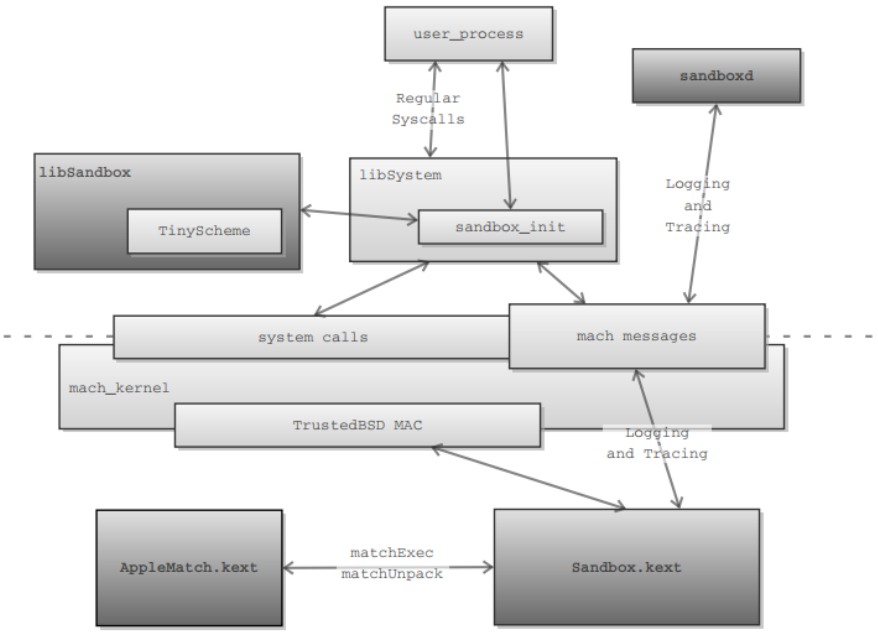
2.5.2 应用程序沙盒化

- 在使用苹果的开发软件（如 Xcode）进行 build 时启用沙盒并进行沙盒权限确认即可。
- 建立于 MacOS 沙盒不是程序的义务，但却是上架 Mac App Store 的必要条件。就算上架 MAS，沙盒机制也是可以绕过去的。开发者在商店之外提供一个程序外挂供用户自由安装。由上架 MAS 的程序指挥外挂进行沙盒外操作即可。

- iOS 是一个从一开始就彻底应用沙盒机制的系统，正常情况下，所有 iOS 设备都只能安装 App Store 中的应用程序，而所有这些应用程序都彻底采用了沙盒机制，不能访问除自己目录之外的资源。虽然保证了安全性，但是也牺牲了一些便利性，如他们的应用程序无法调用其它程序配合完成任务，导致必须重复实现各种已有的功能（如 pdf 阅读功能）。

2.5.3 沙盒机制

沙盒机制见下图：



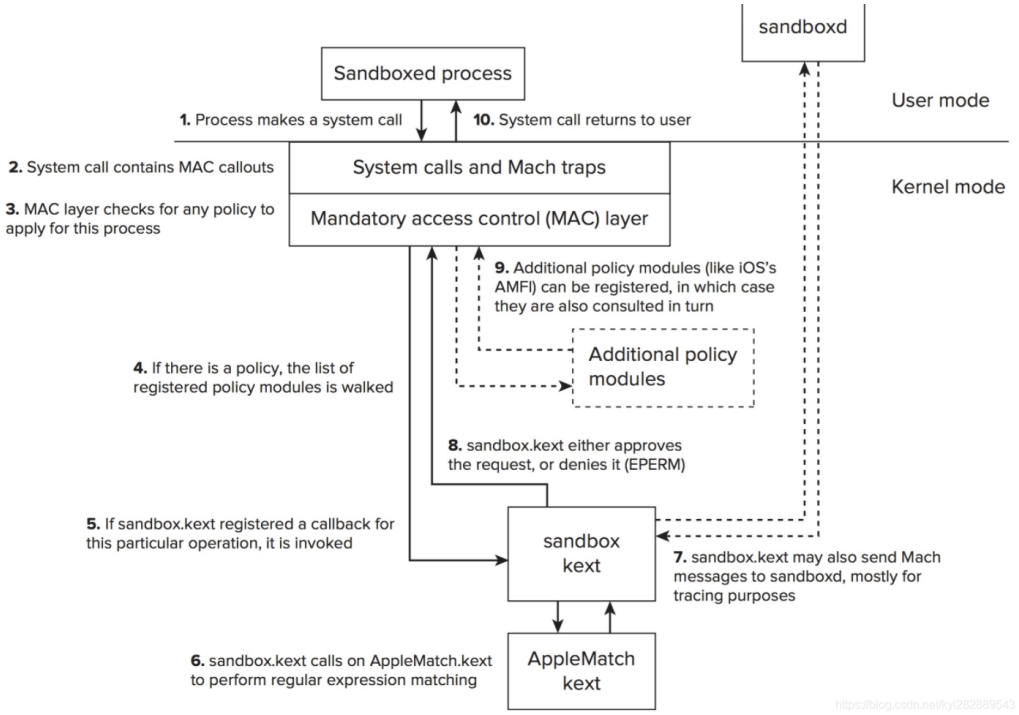


图 1: 沙盒机制 [14]

- 新的应用程序需要经过初始化。原先的、还未运行过的程序没有被 sandbox 约束，但是在初始化过程中，如果动态链接器确认了程序启用 sandbox，就会根据程序中的权限设置 (entitlements) 编译出 profile。在每次程序运行时，动态链接器都会在应用程序的语境中，根据这个 profile 初始化沙盒，然后才把控制权递交给程序。
- 流程
  - 1 进程尝试进行一次系统调用（system call），调用内核功能。
  - 2、3 MAC 层需要根据该进程的安全策略判断此次系统调用是否可以执行。
  - 4、5、6、7、8、9 如果存在策略的话，通过 sandbox.kext（提供 system call 的 hook 函数）和 AppleMatch.kext（解析沙盒的 profile）两个内核扩展实现权限的检查。
  - 10 返回调用结果
- XPC 服务
  - 是 MacOS 下的一种 IPC(进程间通信) 技术, 它实现了权限隔离, 使得 App Sandbox 更加完备。
  - 将程序隔离成多块能提高稳定性，在某部分 crash 时，主程序依然运作。
  - 沙盒中应用无法让子进程拥有比自身更高的权限，但是开发者可以设定 XPC helpers，它们各自运行在自己的沙盒中并拥有刚好供功能实现所需的权限。实现权限分割，安全性就进一步提高了。
- 和 linux 下的 seccomp 有相似之处，即在内核中对 system call 进行过滤。沙盒初始化就像 seccomp 程序启用。

2.5.4 macOS 沙盒优缺点

- **优点**
  1. 安全性高。不影响性能。
  2. 开发者使用方便。
  3. 对于用户需求的敏感访问有系统接管办法，如 Powerbox。
- **缺点**
  1. 自由度低。非 MAS 平台的应用，大部分没有采用 Sandbox 机制。
  2. 沙盒有不可避免的漏洞：
     - 沙盒在应用程序的语境下初始化，如果程序偶然在沙盒初始化完成前就运行了代码，就存在保护失效的可能性。事实上 bug 确实发生了。[15]
     - 在规则内做出背离沙箱的行为。[16] 尽管 Apple 阻止应用程序注册全局接收分布式通知，但没有任何机制能够阻止沙箱应用程序注册接收任何通知。因此，如果按照名称，注册所有的分布式通知，恶意应用程序可以轻而易举地避开 Apple 针对沙箱的限制。尽管这需要一些额外的代码来实现，但其带来的影响是造成任何应用程序都可以通过遍历注册，来接收（捕获）到所有分布式通知，包括在沙箱中也是如此。

2.5.5 小结

Apple 系统的安全性很大程度上都无法离开一种强大的闭环生态，想在设备上运行就必须得遵守相应的约定。而 Linux 是一个非常开放的环境，就此而言，我们不可能让应用在开发的时候就强制开启沙盒，所以我们几乎是没有办法实现一个兼顾安全性、性能、通用性的沙盒的。

对 macOS 沙盒结构的研究，对后续路线一和路线三均有启发。拦截系统调用是目前大多数沙盒的做法，在 Linux 下，我们可以借助 BPF 实现系统调用的拦截。同时，我们可以用编写安全策略加额外策略模块的方式来认证每次系统调用是否安全，若安全则允许调用，若不安全则拒绝调用。

3 项目介绍

3.1 项目设计思路

本项目的设计基于如下观察：用户态的程序编写简单，测试运行简单，适合用来尝试实践复杂的沙盒逻辑，但是用户态的程序为了实现沙盒的功能，需要操作系统通过复杂操作来传递其所需要的上下文数据，需要操作系统通过复杂方式进行沙盒程序的回调，这两者都会产生较大的性能开销。更甚至，沙盒程序的功能与内核功能高度重合，因此需要频繁的进行系统调用，更是会带来很多的上下文切换，产生不必要的性能开销。相对应的，内核中直接实现的沙盒程序则需要通过重新编译整个内核才能测试其在内核正常工作流程中是否能正常工作，一旦出现错误，更可能会导致整个操作系统发生崩溃，所以只有少数几种高度重要的沙盒工具被内置在了 linux 内核中，并且也少有人尝试自己在内核中增加新的沙盒特性来实现某些具体的沙盒保护目的。

而我们的项目基于沙盒程序的现有现状，尝试借鉴用户态沙盒程序的运行流程，但是将进行回调的用户态程序替换成通过模块方式载入到内核代码链接范围的内核态程序段。由于内核态程序可以原生的调用大部分内核函数 ABI，因此在实现沙盒功能的时候有着更好的性能表现，同时上下文的传递也由于同样在内核空间中进行，因此可以直接进行读取。在这种结构下，沙盒程序可以实现较好的性能，编写沙盒程序也变得与编写用户态程序一样非常简单。

最终的项目结构体现出了“策略与机制相独立”的操作系统设计哲学，我们的项目对内核的修改仅限于 20 行以内的修改，简单的创建了静态的系统调用劫持点和相对应的程序回调机制。而实际的回调函数则是在用户态进行独立编译后，使用内核模块动态装载机制，临时的载入到内核空间中进行运行，产生沙盒的隔离功能。

3.2 整体结构

本项目整体分为三个部分：第一部分为可装载到内核中的内核模块，其中包装了实现了沙盒逻辑功能的一系列回调函数和服务回调函数和沙盒功能持久化的一系列数据结构。第二部分通过修改内核源码实现，用来调用模块中实现的回调函数，将沙盒程序加入到内核正常的工作流程中。第三部分是用户态的装载器，其负责将被隔离程序作为子进程调用，并且自动完成沙盒模块的装载工作，将子进程纳入 cgroup 的监测范围内，之后其等待其子进程运行结束，并且在运行结束后自动把装载进去的模块卸载出来，还原其对操作系统的影响。

3.3 可载入模块

本项目共实现了三个不同的模块，实现了三种不同的沙盒隔离逻辑。

- `sBPF_redir`：这个模式的 sBPF 程序会将被隔离程序的所有文件访问路径替换到一个指定的由沙盒程序维护的路径中，阻止对原始文件目录的所有文件访问请求，全部通过篡改访问操作传入参数的方式重定向到沙盒目录中。

- `sBPF_cow`：这个模式的 sBPF 程序会允许程序对原文件系统的文件读取请求，便于运行大部分通用的程序，但是一旦出现写操作的文件请求，会在沙盒目录中复制一份对应的文件，并且把从此以后对同一个文件的所有读写操作全部重定位到沙盒目录中的备份中。通过这种模式，可以在保证程序正常运行的条件下，保护原文件系统的文件内容不被修改，保护用户数据安全。

- `sBPF_permission`: 这个模式的 sBPF 沙盒运行用户指定一个文件目录，则会限制被隔离程序的所有文件请求不能超出此文件目录和其子目录的范围。linux 的文件权限管理只能限制在某个用户文件夹层次上，而这个模式让用户可以给被隔离程序建立一个工作目录，使其文件访问不得超出此目录范围，从而保护用户其他数据的安全。

3.4 内核的修改

内核的修改较为简单，主要是添加了标志了是否载入了沙盒程序的标志位，以及添加了调用沙盒程序的调用入口，通过函数指针就可以简单的实现出来。通过 export 宏定义使其可以被载入的模块读取和修改，就完成了所有的内核修改。

3.5 loader

loader 是我们实现的一个简单的前端，自动化的完成调用目标程序，装载和卸载 sbpf 模块的，简化了沙盒的使用体验。其还附加了与 cgroup 等内核态程序通讯的功能，增强了沙盒的整体功能性。

4 项目实施过程的记录

4.1 BPF 的相关尝试

本项目最初计划通过使用 bpf 程序的形式来实现沙盒的运行逻辑，此 bpf 程序的功能与当前结构中的沙盒内核模块相同，这是考虑到 bpf 程序可以直接通过检查寄存器组的方式获取到系统调用的传入参数和相关上下文，并且有着更为严谨的安全策略保证其安全性。(bpf 相关的基本概念可以参考前几次报告的内容，由于最终实现方式中没有使用 bpf 结构，于是没有这本文中涉及）但是经过了大量的调研与实践，我们最终没能成功的实现基于 bpf 的安全沙盒，这也导致我们大量的工作白费，导致了最终成品中没能实现太多更加复杂可靠的安全策略，而只是作为技术 demo 性质，实现了一些较为简单的安全特性。我们的 bpf 模式沙盒实质上已经完成了系统调用的侦测和系统调用上下文的获取，但是却在将沙盒程序的运行结果传回到原始运行流的过程中遇到了无法解决的困难。这是 linux 环境下可以编译 bpf 程序的两个主流编译器各自存在不同的问题，原先使用的直接从 C 进行 bpf 编译的程序在编译 kprobe 程序进行系统调用侦听时，由于其使用的 ABI 不正确（我们尝试了若干个不同的 kernel 版本，都不成功），总是不能稳定的获取到正确的上下文。而更加高级的 bcc 编译工具组则不知为何，没有支持我们需要使用的一个将数据直接写入用户态内存的 bpf_helper 函数，这导致我们完全无法编译出原先设计的 bpf 沙盒程序，而程序本身又过于复杂，几乎没办法直接用汇编方式实现，于是最终只能放弃了使用 bpf 实现的计划。作为替代，我们便使用了内核模块的方式来进行加载。内核模块的运行原理与 bpf 程序几乎相同，并且可以更加简单的直接通过静态调用点来执行调用，在工期告急的情况下变成了我们的唯一选择。

4.2 内核模块方式的实现

基于我们对 BPF 内核源代码的部分研究，现在我们设计了一种与 BPF 的思想类似，都能支持一种灵活的调整内核态程序的功能的方式。这种方式通过修改内核 + 使用内核模块来实现。

在确定使用内核模块来实现本项目后，我们就开始着手对内核进行修改。虽然内核修改的代码量非常少，但是由于对 linux 的源码编程体系不熟悉，我们通过了大量的努力才成功的编译出了第一版可以正确进行回调的内核。每次修改内核都要经历长达十几分钟的内核编译过程，并且每次替换内核都有导致系统崩溃的可能，于是这个阶段给我们带来的巨大的痛苦。

在内核修改完成后，后续的工作就变得简单了许多，只要不断的编写出不同的测试模块和测试程序，并且将他们组合起来进行多项测试，就完成整体的内核态的工作。而用户态的 loader 则是经过一段时间的调试后，就可以顺利的和内核模块配合进行工作了。

在我们的实现过程中，如何使用内核模块来实现具体的沙盒功能是我们面临的一大困难之一。这是由于内核模块与用户态 C 程序不同，不能使用通用的 libc 库函数，而是只能

使用内核态的内核函数 api。事实上，内核函数的功能比 libc 能提供的功能要多上许多，但是由于没有明确的可用函数列表，我们又对其不够熟悉，因此在前期我们始终难以顺利的开展工作。知道后来，我们才发现内核中的可调用 api 大多在.c 文件中通过 export symbol 宏定义进行了申明，只要大致了解自己需要的功能包含在哪个源文件中，就可以通过翻看所有带有 export symbol 标志的函数，就可以找到自己需要使用的 api。通过这种方式，一下子简化了编程的过程，让内核模块的编写变得和用户态程序的编程一样简单。

最开始的时候，我们希望直接在内核里添加一些全局变量，作为是否有 sBPF 程序载入的标志变量，而这样一个设想也成功的通过了内核的编译过程和模块的编译过程。但是在测试过程中我们却发现内核中的全局变量和模块中读取到的全局变量实质上是两个不同的变量，在装载的模块中修改这个变量没有对内核的运行过程产生任何影响。通过多方面的研究了 linux 内核及其模块的工作机制，我们了解到了 linux 内核源码中通过 `EXPORT_SYMBOL` 这个宏函数来实现了所有的全局变量和全局函数。所有需要在不同模块中互相访问的变量和函数都需要通过这个宏定义来注册到 linux 内核中。我们通过给我们的全局标志变量加上了 `EXPORT_SYMBOL`，成功的让我们的内核模块在装载后可以读取到内核中声明好的全局变量，并且成功的通过修改这个变量影响了内核的工作流程，实现了我们的目标。实际上，在真正解决全局变量这个问题之前，我们还尝试过一种更“手动”的方法，即先将我们想要修改值的变量的地址输出出来，再在内核模块中按地址去修改值。这种方法虽然有用，但是明显非常麻烦。在我们发现用 `EXPORT_SYMBOL` 能申明全局变量后，这种方法就被废弃掉了。

编写内核程序还包括的一个困难点是内存管理。Linux 内存被划分为用户空间和内核空间，我们遇到的一个很大的难题是：内核程序很难申请用户空间的内存。我们的程序有时需要使用到系统调用，虽然说内核程序找到系统调用的入口并不是很难——只需要找到相应的 `SYSCALL_DEFINE`，再调用相应实现的函数就可以了。但是部分系统调用只能传入用户空间的内存，而正如前面所说，内核程序想申请用户空间的内存并不是一件简单的事情。期间，我们尝试过使用 `brk` 系统调用增大用户空间内存，但是 `brk` 在内核中的实现较为复杂，所以我们并未成功。到最后，我们找到了一种方式勉强解决这个问题，不过并没有真正解决好。如果能真正解决这个问题，沙盒的运行效果会更好。

5 功能演示与性能测试

5.1 功能演示

这个功能演示演示的是使用 cow 模式隔离一个测试程序，这个测试程序会尝试读取原始文件系统中本来存在的一个文件并且打印其内容，之后会尝试篡改统一个文件的内容，然后其会再进行一次读取和打印。可以看到这个程序的两次读取都成功了，运行的效果与其正常运行相同，然而原始文件并没有被修改，而是在沙盒目录中创建了另一个文件并且修改了这个文件。

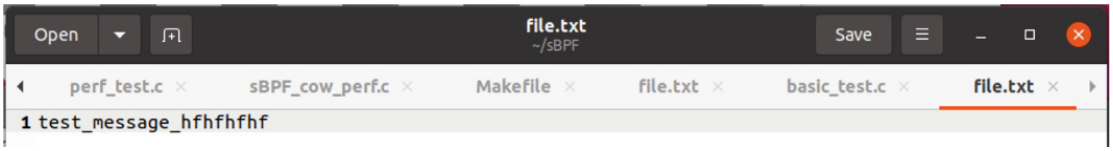


图 2: 初始文件

首先，将一个初始文件放置在 `~/sBPF/file.txt`，文件的内容如上图所示。

```
lhy@ubuntu:~/sBPF/test_module$ ./loader sBPF_cow.ko /home/lhy/sandbox_test ./a.o
ut
pid=8883, u_mem=281474427917000, sdir=/home/lhy/sandbox_test
test_message_hfhfhfhf
test_message_222222
```

图 3: 将沙盒载入内核

然后，将沙盒载入内核，沙盒管控一个程序，该程序会对该文件进行一次读取、写入、读取操作，写入的内容为 `test_message_222222`。

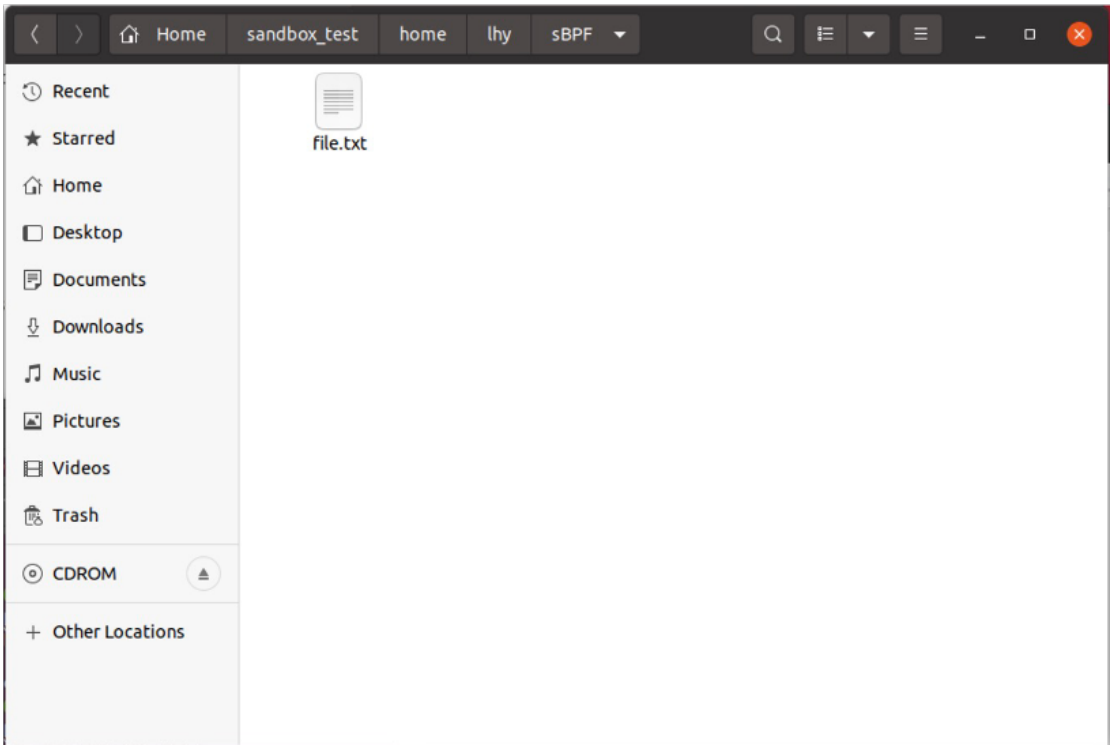


图 4: 沙盒文件路径

程序执行结束后，写入的文件出现在沙盒路径中。

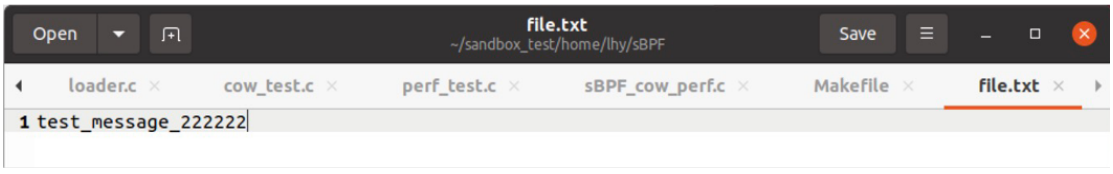


图 5: 沙盒文件内容

打开文件，内容是写入的新内容，原文件则依旧位于原先的位置，并且内容没有发生改变。

5.2 性能测试

我们对我们的沙盒程序在 cow 模式下进行了性能测试（这是由于这个模式比较接近正常沙盒系统的运行状况），性能测试的内容是重复百万次文件的打开，读写，关闭。因为在

虚拟机中运行，时间无单位，只进行相对对比。可以看到我们的程序很接近原生的性能，明显好于 gVisor 的性能 [4]。

图 6: 正常读写（左图），沙盒读写（右图）

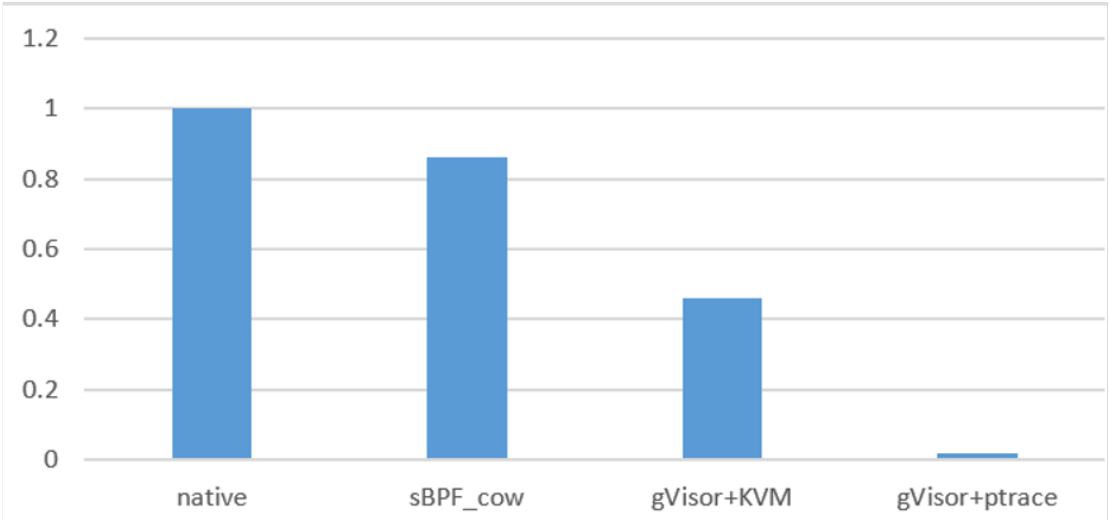


图 7: 效率对比

5.3 项目总结

本项目在整个过程中历经波折，甚至修改过立项之初的一些想法，最终才达到了今天的完成状态。期间本组的队员们遇到了很多超出想象外的困难，并且在解决这些困难的过程中增加了我们对 linux 内核运行过程的很多理解。开学时，我们对操作系统的运行原理一窍不通，甚至一开始时对内核态用户态的区别都一切不通。通过在内核态进行很多的修改和测试，我们解决很很多诸如内存管理权限问题、不同用户进程对内存地址的理解不同的问题（由于段页式内存对每个进程进行的内存抽象），在完成了此项目的过程中我们通过查找资料与多种编程尝试，成功的从理论上和实践上解决了这些问题。在项目的整个推进过程中，我们也实现了成功的分工和时间规划，高效有序的完成了整个项目的推进，并且最终完成了这个项目。

参考文献

[1] D. Calavera and L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[2] I. Korchagin, "Sandboxing in linux with zero lines of code." Website, 2020. https://blog.cloudflare.com/sandboxing-in-linux-with-zero-lines-of-code.

[3] Z. Wan, D. Lo, X. Xia, and L. Cai, "Practical and effective sandboxing for linux containers," *Empirical Software Engineering*, no. 8, 2019.

[4] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[5] D. J. Walsh, "Are docker containers really secure?." Website, 2014. https://opensource.com/business/14/7/docker-security-selinux.

[6] "An introduction to common linux viruses." Website, 2020. https://segmentfault.com/a/1190000022761270.

[7] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "How the {ELF} ruined christmas," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 643–658, 2015.

[8] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125, Springer, 2008.

[9] "Revealed bank trojan chthonic: the latest variant of the online silver thief zeus." Website, 2018. https://cloud.tencent.com/developer/article/1036506.

[10] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[11] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[12] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[13] A. Hoog and K. Strzempka, *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier, 2011.

[14] D. Blazakis, "The apple sandbox," *Arlington, VA, January*, 2011.

[15] M. Blochberger, J. Rieck, C. Burkert, T. Mueller, and H. Federrath, "State of the sandbox: Investigating macos application security," in *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pp. 150–161, 2019.

[16] "[0day] mojave's sandbox is leaky." Website, 2018. https://objective-see.com/blog/blog_0x39.html.