大作业-调研报告

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 4 月 11 日

目录

1 小组成员

- 陈思睿
- 梁恒宇
- 吕泓涛
- 汤力宇

2 项目简介

使用新兴的 eBPF 架构，实现兼有安全性和性能的通用沙箱。

任何操作系统都或多或少的潜藏着安全漏洞，近年来各大主流 OS 都被爆出过存在重大安全隐患。当恶意程序侵入用户的系统，则可能破坏、窃取宝贵的用户数据，造成不可估量的损失。而为了防止此类事件发生，沙盒技术正在不断发展。沙盒技术通过对可疑的进程进行隔离与监控，防止其对系统其它部分造成损害。

然而随着恶意程序的攻击策略不断扩展，传统沙盒的安全性也难以长期保持，因此一个理想的沙盒应当在高效、安全同时拥有便于升级维护的特点，这一理想在现有的沙盒中难以实现。用户态的沙盒存在着大量到内核态的状态切换，带来了严重的性能损失，而内核态的沙盒则在损失了升级的灵活性的同时带来了更多可能被攻击的安全漏洞。

面对这一困境，本项目希望使用新兴的 eBPF 架构，实现一个可以从用户态灵活对其升级的内核态沙盒。由于 BPF 架构的设计特征，这一沙盒将不会给内核多带来额外的安全漏洞，并且将可以实现和内核态相仿的性能。

3 项目背景

3.1 eBPF

Extended Berkeley Packet Filter (eBPF，或简称为 BPF) 是一种新兴的技术, 此技术允许用户态进程向 kernel 提交代码段并且将这些代码在 kernel 模式下运行。这项技术已经被 linux 支持，众多基于 eBPF 的项目正在进行中。

- eBPF 的运行原理如下 [1]：

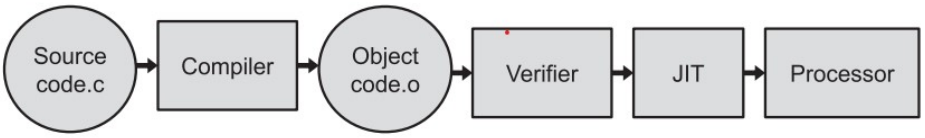

图 1: eBPF 运行原理示意图

1. eBPF 的程序可以直接通过 C 语言或 RUST 语言编写，经过专用的编译器编译成 BPF 字节码（bytecode）并提交给 Verifier 审核
2. Verifier 需要执行若干操作来确保收到的 BPF program 可以安全的在内核中运行，这些操作包括：
   (a) 限制 BPF 程序的总长度。
   (b) 只允许出现固定轮数的循环。
   (c) 只允许在程序结束时调用其他子程序作为延续。
   (d) 只允许通过专用接口调用受限制的若干种系统调用。
   (e) 程序的最坏运行时长不得超过某个值。

   在这些限制条件下，确保了 BPF 程序不会死循环，不会占用过多 CPU 时间，并且难以对 OS 造成损害。通过了 verifier 的程序会被送入 JIT（JUST-IN-TIME Compiler）。
3. JIT（JUST-IN-TIME Compiler）会将 bytecode 翻译成对应架构的二进制程序（如 x86 或 arm 的程序段）然后将此二进制程序放入专用内存空间存储和等待运行，使此程序的运行时拥有几乎原生的性能。

4. 存储层面上，eBPF 程序的代码段一经写入就会被设置为只读，防止其内容被篡改。eBPF 程序也不能直接调用通用的内存空间进行本地数据的存储，其需要使用通过 bpf helpers syscalls 创建 bpf map。内核态可以直接访问 map，写入数据。用户态需要得到 map's file descriptor，间接访问。BPF 程序与用户态进程交互通过 bpf map 实现。
5. 执行时，eBPF 程序不允许主动的调用执行，只能通过 BPF 钩子（hook）被动的调用。通过在对应事件上注册 BPF 钩子即可在事件发生时调用对应程序。简单调用的场景例如：
   当任意系统调用发生时，调用某 BPF 程序记录系统调用的类型事件并写入系统日志；
   当接收到任意网络数据包时将所有类型为 XXX 的包发送给用户态程序进行分析。

通过以上措施，BPF 实现了灵活且安全的在内核态植入一些程序从而高效的实现了很多重要的功能。

- eBPF 目前在网络分析、性能优化、负载调度等领域有众多应用，包括如下

1. 快速数据包处理，如控制分发操作 (IoT)。
2. 网络路由，子节点的路径管理 (InKeV)。
3. Container 技术，将一套应用程序所需的执行环境打包起来。将 BPF 程序放进每个 Container，对每个应用程序做出更强的管理 (Cilium)。
4. bcc [2]
   bcc 即 BPF Compiler Collection，是一个能高效编写内核追踪和管理程序的工具包。它是主要的 BPF 前端项目。用户能够为 BPF 编写 C 程序，或 Python，lua，由 BCC 转化成 BPF 应用（借由 LLVM 生成字节码），是一种动态编译的方式。
5. bpftrace [3]
   构建于 BPF 和 bcc 之上的追踪工具。动态监测工具，同样用于与 BPF 进行交流。用户使用单行命令或编写程序获取目标进程的行为，或记录过程。
6. libbpf [4]
   libbpf 目标是为了使得 bpf 程序像其它程序一样，编译好后，可以放在任何一台机器，任何一个 kernel 版本上运行。使用 libbpf 可以像编写普通用户态程序一样开发 BPF 程序，比 BCC 更精简快速。
   背景：有些 BPF 程序不需要获取内核数据结构，只是捕获系统调用，但是它们比较稀有；BPF 提供了一些稳定接口，让部分结构具有统一性，但是十分有限。为了对付具有很强动态性的语境，BCC 采用动态编译，但是为此付出了不简洁、存在错误率的代价。
   使用内核的 BTF(BPF Type Format) 信息可以实现结构定位，并且用它成功生成了一个巨大的头文件 vmlinux.h，这个文件代替了特化的内核版本头文件，让程序具有到处运行的可能。
   在某些泛用条件下，BCC 程序可以转化为 libbpf 程序。

7. Falco [5]
   应用反常行为监视工具。能持续性监视和侦测容器、应用、远程计算机、网络的行为，比如 namespace 的切换，敏感读写，执行 shell。
   组成：
   (a) Userspace program，用户与 Falco 交流的命令行界面。
   (b) Configuration，控制 Falco 的运作机制。
   (c) Driver，一个获取系统调用并返回给用户态的工具，其中一个选择就是 eBPF probe。
8. Katran [6]
   一个 C++ 库/BPF 程序，能够搭建高性能的 layer 4 负载均衡器。利用了 XDP 和 BPF 提供内核中的工具来进行快速包处理。Katran 部署在 Facebook 的后端服务器上，它帮助 Facebook 提高了网络负载均衡的性能和可扩展性。
   XDP 是一个内核组成，能进行快速包处理。
9. Hbble [7]
   分布式网络安全保障和观测平台。构建于 Cilium 和 BPF 之上，能够深入获取信息，比如网络服务的交流频率，哪些网络服务被阻挡或者被服务集群外部访问，而借由 BPF 工具这些观测的开销很小。
10. tracee [8]
    实时的系统和应用追踪工具，能够分析、收集事件来探测可疑行为。它作为一个 docker 镜像来运作。使用 go 语言。
    其组成部分 Tracee-eBPF 用于事件收集。而 libbpgo 是一个帮助 go 编写 BPF 程序的库，使用了 libbpf。
11. OpenFlow 是 layer 2 的网络通信协议。[9] 使用 BPF 对网包做归类，能够提供灵活的匹配机制，优于 OpenFlow 采用的在不断拓展的固定匹配字段。一个复杂的 BPF 程序实现了一个端口速度（line-rate）的任意网包归类。
12. 采用 BPF 程序控制网包分配操作。[10] 每个特定的关于网络交通流和网包复制的计算流程被一个内核中的虚拟机控制。通过分配实时分析工作给多个边缘处理器，解决了网络带宽限制的问题。
13. 传统 linux 系统的安全特性集中于 iptables。[11] 由于网速的提高和代码量的暴增，传统实现方案逐渐力不从心。而 BPF 提供了在网包接收或发送是执行几乎任意特定代码的优势，能够实现 Over-the-Air updates。采用 BPF 制作的 iptables 教传统方法具有更为优越的性能。

具体的案例见后文相关工作部分

3.2 虚拟化技术

虚拟化（技术）或虚拟技术（英语：Virtualization）是一种资源管理技术，是将计算机的各种实体资源（CPU、内存、磁盘空间、网络适配器等），予以抽象、转换后呈现出来并可供分割、组合为一个或多个电脑配置环境。由此，打破实体结构间的不可切割的障碍，使用户可以比原本的配置更好的方式来应用这些电脑硬件资源。这些资源的新虚拟部分是不

受现有资源的架设方式，地域或物理配置所限制。一般所指的虚拟化资源包括计算能力和资料存储。(wiki)

- 虚拟机的分类

  1. 完全虚拟化：虚拟机管理器将虚拟整个 OS 以满足任意软件的运行需求。一般的，其截获并筛选 guestOS 将要运行的指令，相对安全的指令将被提交给物理 CPU 直接运行，相对危险的指令（如修改系统时钟或修改中断寄存器）将被虚拟机管理器截获并且通过模拟运行结果的方式返回给 guestOS。
     熟知的例子包括：QEMU、VMware 等
  2. 部分虚拟化：虚拟机只针对特定应用程序进行虚拟，而不是虚拟整个操作系统。
  3. hypervisor 模型：主机运行的 OS 直接负责执行虚拟化，VMM 同时负责管理虚拟机和管理系统硬件
  4. host OS 模型：VMM 作为通用 OS(如 linux) 的一个模块被加载，VMM 通过请求系统调用来满足虚拟机的运行需求，VMM 在 OS 视角来看类似于一个普通的进程。随着 linux 的虚拟化功能越来越多，他正在从 Host 模型发展为 Hypervisor 模型。
  5. 混合模型：VMM 作为最底层调度硬件，但是额外运行了一个虚拟的操作系统来执行 IO 的适配，通过这种模式，VMM 开发者不必编写大量代码适配各种复杂的硬件，减轻了开发难度。但相对的 guestOS 的请求需要多次转发才能得到满足，影响了性能。

- 虚拟机的实现结构

  1. hypervisor 模型：主机运行的 OS 直接负责执行虚拟化，VMM 同时负责管理虚拟机和管理系统硬件
  2. host OS 模型：VMM 作为通用 OS(如 linux) 的一个模块被加载，VMM 通过请求系统调用来满足虚拟机的运行需求，VMM 在 OS 视角来看类似于一个普通的进程。随着 linux 的虚拟化功能越来越多，他正在从 Host 模型发展为 Hypervisor 模型。
  3. 混合模型：VMM 作为最底层调度硬件，但是额外运行了一个虚拟的操作系统来执行 IO 的适配，通过这种模式，VMM 开发者不必编写大量代码适配各种复杂的硬件，减轻了开发难度。但相对的 guestOS 的请求需要多次转发才能得到满足，影响了性能。

- 虚拟化可以使用的两种特殊场景

  1. 沙盒：为了保护宿主 OS 不被恶意进程破坏而制作出来的隔离运行环境。
  2. 容器：为了追求部署的便利和运行的效率而制作出来的高效运行环境。

3.3 沙盒

沙盒（英语：sandbox，又译为沙箱）是一种安全机制，为运行中的程序提供的隔离环境。通常是作为一些来源不可信、具破坏力或无法判定程序意图的程序提供实验之用。沙盒通常严格控制其中的程序所能访问的资源，比如，沙盒可以提供用后即回收的磁盘及内存

空间。在沙盒中，网络访问、对真实系统的访问、对输入设备的读取通常被禁止或是严格限制。从这个角度来说，沙盒属于虚拟化的一种。沙盒中的所有改动对操作系统不会造成任何损失。通常，这种技术被计算机技术人员广泛用于测试可能带毒的程序或是其他的恶意代码。(wiki)

1. 传统上为了实现沙盒可以使用如下的若干种方式
   - 软件监狱（Jail)：限制网络访问、受限的文件系统 namespace。
   - 基于规则的执行：通过系统安全机制，按照一系列预设规则给用户及程序分配一定的访问权限，完全控制程序的启动、代码注入及网络访问 [5]。也可控制程序对于文件、注册表的访问。在这样的环境中，病毒木马感染系统的几率将会减小。Linux 中，安全增强式 Linux 和 AppArmor 正使用了这种策略
   - 虚拟机：模拟一个完整的宿主系统，可以如运行于真实硬件一般运行虚拟的操作系统（客户系统)。客户系统只能通过模拟器访问宿主的资源，因此可算作一种沙盒。
   - 在线判题系统：用于编程竞赛中的程序测试。
   - 安全计算模式（seccomp)：Linux 内核内置的一个沙盒。启用后，seccomp 仅允许 write()、read()、exit() 和 sigreturn() 这几个系统调用。(后文将会详细介绍此技术)
2. 近年来计算机安全需求不断升级，又出现了很多新兴的沙盒技术
   - unikernel：给应用带上自己的核，开销较小并且难以被攻击，其缺点在于需要单独设计对应的应用。
   - MicroVM: 运行另一个剪裁过的内核并将其暴露给应用程序，恶意进程将难以同时攻破两个 OS 的安全措施。
   - 进程虚拟化：利用软件模拟出 linux 运行环境，能够强化沙盒的安全性。(其安利 gvisor 将在后文被详细分析)
3. linux 对沙盒技术的支持主要包括以下三个工具
   - **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 [12]，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。[13]
   - **Cgroups**
     Cgroups介绍。其主要支持四个功能：
     (a) 限制某些 groups 的资源使用量 (包括 CPU，IO，内存等)
     (b) 调整优先级，使某些 group 可以分配到更多的资源

(c) 记录某些 group 的资源使用量

(d) 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。

  (a) 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。[14]

  (b) 某种沙盒使用的安全方案 [14]：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。

3.4 容器

操作系统层虚拟化（英语：Operating system─level virtualization），亦称容器化（英语：Containerization），是一种虚拟化技术，这种技术将操作系统内核虚拟化，可以允许用户空间软件实例（instances）被分割成几个独立的单元，在内核中运行，而不是只有一个单一实例运行。

这个软件实例，也被称为是一个容器（containers），虚拟引擎（Virtualization engine），虚拟专用服务器（virtual private servers）或是 jails。对每个行程的拥有者与用户来说，他们使用的服务器程序，看起来就像是自己专用的。

操作系统层虚拟化之后，可以实现软件的即时迁移（Live migration），使一个软件容器中的实例，即时移动到另一个操作系统下，再重新运行起来。但是在这种技术下，软件即时迁移，只能在同样的操作系统下进行。

在类 Unix 操作系统中，这个技术最早起源于标准的 chroot 机制，再进一步演化而成。除了将软件独立化的机制之外，内核通常也提供资源管理功能，使得单一软件容器在运作时，对于其他软件容器的造成的交互影响最小化。

相对于传统的虚拟化（Virtualization），容器化的优势在于占用服务器空间少，通常几秒内即可引导。同时容器的弹性可以在资源需求增加时瞬时复制增容，在资源需求减小时释放空间以供其他用户使用。由于在同一台服务器上的容器实例共享同一个系统内核，因此在运行上不会存在实例与主机操作系统争夺 RAM 的问题发生，从而能够保证实例的性能。(wiki)

3.4.1 Docker 结构

- Docker 是一种操作系统级别的虚拟化技术，具有以下特点
  - Docker Engine 直接运行在 Linux Kernel 之上
  - 多个 containers(若有) 共享同一个 Linux Kernel

  - 从内部来看,container 内部的进程只能看到给他分配的资源
  - 从外部来看, 多个 container 就是一个个普通的进程

- 进程隔离技术：采用了 Linux 内核所实现的 cgroup，namespace
  - Linux namespace
    * 为每个 container 分配自己独立的资源
    * 进程不可访问别人 namespace 内的资源
    * 包括 5 种：mount，hostname，IPC，PID，network
  - CGroups
    * 分配和管理 container 内部的进程可以使用的资源
  - POSIX Capabilities
    * 细化进程权限
  - Seccomp
    * 限定系统调用

- 一些概念
  - Docker Image
    构建 container 的模板, 一般包含应用程序及其依赖
  - Docker Container
    Image 的实例, 相当于进程至于程序的概念
  - Dockers Registry
    可信镜像的仓库
  - Docker Engine （一般主指 Dockers Daemon）
    负责创建和管理 containers(如分配其对应的 namespace)
  - Kubernetes
    编排多个 containers, 如提供 containers 间通信等复杂功能

3.4.2 Docker 安全问题

根本问题 [15–18]：

- docker 直接共享内核，减少的抽象层次带来比 vm 更轻量而高效。
- 同时恶意进程也更容易的攻破 Host 系统（需要攻破的层次更少）。

当关闭 selinux 时，container 通过攻击不受 namespace 限制的 kernel keyring 取得其他容器的存于此的 key。[19]

由于操作系统内核漏洞，Docker 组件设计缺陷，以及不当的配置都会导致 Docker 容器发生逃逸，从而获取宿主机权限。[20]

docker daemon

- Docker 守护进程也可能成为安全隐患。Docker 守护进程需要根权限，所以我们需要特别留意谁可以访问该进程，以及进程驻留在哪个位置。

折中解决方案

- 由于频发的安全及逃逸漏洞，在公有云环境容器应用不得不也运行在虚拟机中，从而满足多租户安全隔离要求。而分配、管理、运维这些传统虚拟机与容器轻量、灵活、弹性的初衷背道而驰，同时在资源利用率、运行效率上也存浪费。

4 立项依据

本项目旨在实现一种兼有安全性与运行效率的沙盒架构。提高沙盒的安全性与运行效率是当今计算机界的重要课题，而沙盒应用的升级问题和兼容问题更是可以带来重大的优化。本项目突破了用户态与内核态二选一的固有思维方式，将 eBPF 技术引入沙盒这一领域，尝试利用 eBPF 这一新兴技术实现安全性和性能的最大化。eBPF 技术可以在用户态进程的引导下在内核态运行特定代码段，从而此框架可以实现用户态的灵活性和内核态的高效。近年来，这一框架的潜力正不断被业界挖掘出来，基于 eBPF 的各种功能不断的被实现，众多传统模块也已经在 eBPF 框架下实现了升级。我们的项目也将借助 eBPF 架构的优点，借鉴同类 eBPF 项目的经验，解决传统沙盒的各种弊端，实现对沙盒应用的全方位优化。

5 项目重要性

- 性能方面
  性能方面，BPF 程序由于其运行在内核态的特征，在运行过程中避免了多余的特权状态切换，保证了其在同等逻辑结构的沙盒中可以有着出色的运行性能，减少了不必要的性能损失。

- 安全性方面
  安全方面，BPF 程序受到统一的保护和管理，其代码段的只读特性可以防止其内容被恶意篡改，其受限制的系统调用形式确保了其无法发出可能损害 OS 的 syscall，其受到统一管理的局部存储池则保证了其不会出现各类内存相关的漏洞或错误。因此使用 BPF 程序的方式实现的安全沙箱可以最小化其对 OS 带来的潜在安全隐患，可以实现优化系统的安全性的目标。

- 热升级特性
  BPF 程序通过名为事件源或钩子的方式进行调用运行，此调用方式类似于函数指针，因此当需要对沙盒本体进行升级时，不必关闭或暂停所运行的服务，只需要简单的对钩子进行重定向便可实时的完成升级，即拥有了热升级的可能性。

- 便于移植
  BPF 程序的结构设计上有着较好的移植性，在支持同一套 BPF 接口协议的不同 OS 上，即使存在架构层面的不同，也可以简单的实现移植，因此通过 BPF 程序实现的沙盒将可以拥有类比于网页对浏览器的兼容性。

6 相关工作

6.1 MBOX

MBOX 是一个为非 root 用户提供的沙盒环境，主要面对 filesystem 进行保护。

在与 MBOX 有关的一篇论文 [21] 中，详细介绍了此沙盒面对的使用场景与其解决方案，此沙盒使用了上文提到的 seccomp-bpf 作为工具增强其安全性。

- 此沙盒面对的使用场景如下：
  1. 为非管理员用户构造虚拟的 root 权限，便于这类用户执行一些特殊的任务。例如 vlab 提供的 fakeroot 即为此类应用。
  2. 安全地运行不可信的二进制文件，可以用于分析潜在的恶意程序或病毒。
  3. 为文件系统提供检查点（check point）。当用户需要处理危险的文件的时候，传统上一旦文件出现错误就需要使用专用工具修复文件系统。使用 MBOX 则可以在开始处理前把运行环境转移到虚拟的文件系统中，这样在发生错误后可以提取 sandbox 中剩余的错误信息，并且可以正常的时候原理的文件系统，如果运行成功了也可以直接把 sandbox 的系统与原文件系统合并（听起来很像 git）。
  4. 使用 MBOX，用户可以简单的构建开发环境，构建过程只需要将配置好的开发环境整体移植到沙盒内即可完成。
  5. 细化文件权限管理，通用 os 如 linux 中用户建立的进程有权限访问用户的所有个人文件，使用 MBOX 可以使特定用户进程只能访问必须的文件，保护了其他文件。
- 其大致上的实现原理如下：
  1. 给文件系统增加了一个 private layer，位于原生文件系统的上层，每个 sandbox 会对应生成一个 MBOX 文件系统，用于服务沙盒中的进程。此文件系统的储存结构从原生文件系统的角度来看只是普通的目录，但是沙盒中的程序必须通过 MBOX 获得文件服务。
  2. 虚拟文件系统中的文件变动不会实时更新到 host 文件系统，但是 host 文件系统中的变动可以实时同步到虚拟文件系统中，当两个系统出现冲突的时候由用户决定保留哪一个版本。
  3. 使用了 **seccomp-bpf** 和 **ptrace** 干预系统调用并且实现 fakeroot。具体的，其作为过滤器来干预沙盒中进程的 syscall，如限制某进程对 socket 的调用。
  4. 其对 sefccomp-bpf 的使用与我们的设想相同，将每个 syscall 的进入调用挂在某个 bpf 程序上，bpf 程序接受 syscall 的类型，进程的属性等信息，计算是否有相应权限。

6.2 gVisor

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 [22] 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? [23])

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。

参考文献

[1] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[2] I. Visor, "bcc." https://github.com/iovisor/bcc, 2021.

[3] I. Visor, "bpftrace." https://github.com/iovisor/bpftrace, 2021.

[4] A. Nakryiko, "Bpf portability and co-re." Website, 2020. https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html.

[5] Falco, "falco." https://github.com/falcosecurity/falco, 2021.

[6] F. Incubator, "Katran." https://github.com/facebookincubator/katran, 2021.

[7] Cilium, "hubble." https://github.com/cilium/hubble, 2021.

[8] A. Security, "tracee." https://github.com/aquasecurity/tracee, 2021.

[9] S. Jouet, R. Cziva, and D. P. Pezaros, "Arbitrary packet matching in openflow," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–6, 2015.

[10] S. Baidya, Y. Chen, and M. Levorato, "ebpf-based content and computation-aware communication for real-time edge computing," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 865–870, 2018.

[11] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with ebpf iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, (New York, NY, USA), p. 108–110, Association for Computing Machinery, 2018.

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

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

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

[15] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[16] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

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

[18] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[19] D. Walsh, "Yet another reason containers don't contain: Kernel keyrings." Website, 2014. https://www.projectatomic.io/blog/2014/09/yet-another-reason-containers-don-t-contain-kernel-keyrings/.

[20] A. Team, "深度解析 aws firecracker 原理篇一 虚拟化与容器运行时技术." Website, 2019. https://aws.amazon.com/cn/blogs/china/deep-analysis-aws-firecracker-principle-virtualization-container-runtime-technology/.

[21] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 139–144, USENIX Association, June 2013.

[22] 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.

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