vkernel 项目开发文档

一、面向轻量级虚拟化的优化技术项目概述

1.1 项目背景

容器作为一种轻量级虚拟化技术，具有启动快速、部署方便、资源占用少、运行效率高等优点，近年来广泛应用于云计算平台和数据中心的资源管理、系统运维和软件部署中，不仅成为云计算领域的业界热点，而且具有改变软件整体生态的潜力。作为业界最具代表性的云计算平台，亚马逊云、谷歌云、微软云和阿里云等都先后在云计算业务中推出各自的容器云服务。最近容器技术开始走出数据中心，除了传统的微服务、DevOps 等领域，已逐步应用于包括移动计算、物联网等在内的新领域，表现出了良好的发展态势和应用前景。

大量的云服务主机采用的是 Linux 操作系统，Linux 原生提供了命名空间（Namespace）和控制组（Cgroup）等技术来实现系统层级的环境虚拟化。命名空间为每个容器提供独立的操作系统视图，让容器中的应用感觉完全在独立的环境中运行。控制组则对每个容器可以获取到的系统共享资源进行分配与管理。虽然命名空间与控制组为容器提供了隔离性，但是由于所有的容器仍然是共享主机系统的内核，容器在安全性方面的防护仍旧较弱。共享内核一方面为容器带来高效性，但是另外一方面也给容器带来了诸多的安全隐患。该问题成为制约容器发展和应用的主要技术瓶颈，目前业界有以下几种主流的方案：

（1）基于主机内核的隔离方案：容器利用 Linux 内核提供的命名空间和控制群组来实现容器的视图隔离和资源隔离，利用 Capability、Seccomp、Apparmor 机制实现容器的安全隔离。

（2）基于 gVisor 的用户态内核：gVisor 为容器实现了一个用户态内核，使用进程虚拟化的方法隔离容器内的恶意代码对 Host Kernel 的访问。它在内核之外实现了一个“内核进程”——Sentry，Sentry 提供了大部分 Linux Kernel 的系统调用。并且通过巧妙的方式将容器内进程的系统调用转化为对这个“内核进程”的访问，实现对系统调用的重定向和重构。

（3）基于轻量级虚拟机的 guest 内核：比较典型的是 Kata 容器，Kata 容器

基于 Hypervisor 技术为每个容器虚拟化出一个经过裁剪的独立内核，让每个容器运行在一个轻量级的虚拟机中，解决了传统容器共享内核的安全和隔离问题。

第一种方案本质上还是多个容器共享服务器操作系统内核，虽然效率高，但隔离性最弱，而后两种方案虽然在隔离性和安全性上有了很大的提升，但它们均以牺牲性能为代价。传统容器隔离性不足的本质在于多个容器共享同一个操作系统内核，内核如果发生漏洞或崩溃会导致所有容器无法正常运行，且统一的内核使用户无法根据容器的特点进行内核服务的定制，这导致多容器场景下，容器安全性差、灵活性差、稳定性差的问题。

针对上述问题，本项目拟进行的工作是：实现既保证隔离性又拥有近似于原生 Docker 容器性能的容器技术。

1.2 预期目标

本项目旨在对现有轻量级虚拟化技术进行优化，改进目前容器技术存在的安全性和灵活性问题。项目的预期实现目标主要有：

低启动延迟：在实现较强的隔离性与安全性的前提下，能够实现接近于原生 Docker 容器的启动延迟，不会为项目部署带来严重的额外开销。

高性能：不同于 gVisor 和 Kata 等方案以牺牲性能为代价换取隔离性与安全性，本项目希望能够实现接近原生 Docker 容器的负载性能，保持较高的资源利用率。

高安全性：本项目相比原生容器具有较高的安全性，能够防护容器中常见的 CVE 提权漏洞，一定程度上保护运行中的容器和宿主机的安全。

性能隔离：原生容器共享主机内核，因此不同容器之间存在一定的性能干扰，会导致容器无法发挥预期性能并产生一定的安全风险。本项目希望设计一种容器性能隔离机制，能够实现容器之间的性能隔离，减少负载之间的性能干扰。

1.3 系统架构

我们的项目名称为 vkernel（virtual kernel），它使用内核可加载模块技术在内核层实现虚拟内核，旨在打破容器场景中共享内核带来的安全限制，具有安全性强、性能高的特点。如图 1-1 所示，vkernel 由以下几个部分组成：

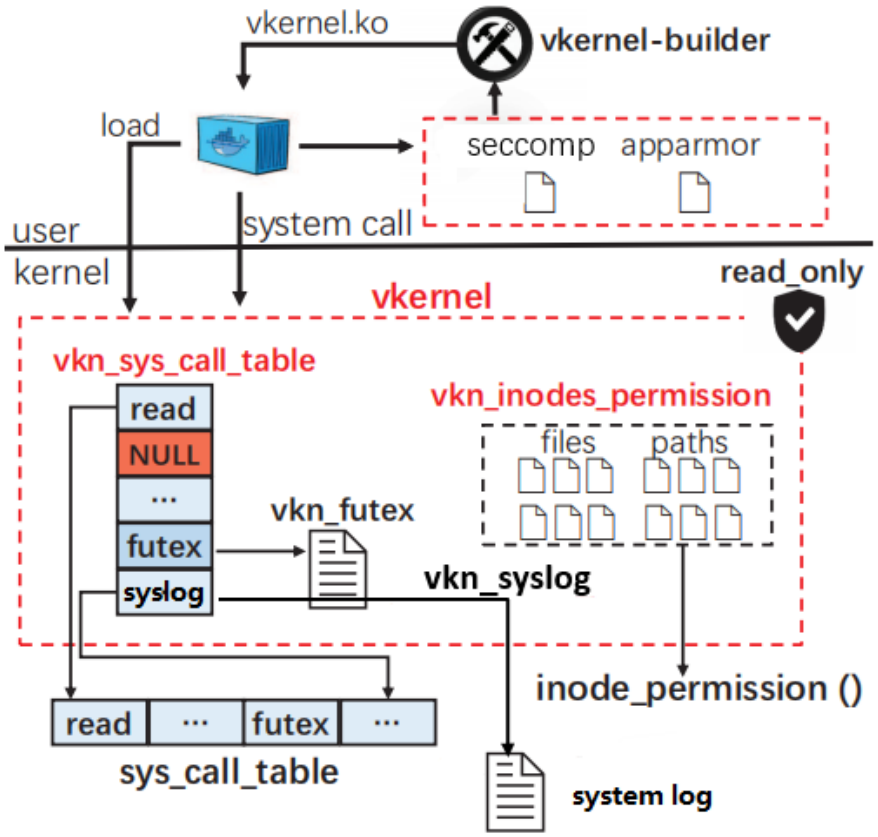

图 1-1 vkernel 系统整体架构图

内核模块隔离系统：模块系统是 vkernel 的关键，实现了容器内核资源的虚拟化和安全。Linux 内核被所有容器共享的，这种共享带来了高性能的同时，也给容器的隔离性带来了巨大的挑战。虽然内核已经面向容器增加了诸多隔离功能，但是内核层的隔离仍然不够完善，需要一步一步进行提升。

内核日志隔离系统：不同容器的进程往往有自身的调试或获取内核状态的需求，当前容器共享主机内核的内核日志，能够通过共享的日志感知到其他容器以及主机进程的事件发生，通过 vkernel 实现内核日志的隔离，不同容器拥有独立的内核日志。

系统调用虚拟化系统：系统调用的共享对容器安全性影响十分明显，一方面多个容器竞争调用相同系统调用会降低该系统调用的性能，另一方面恶意容器可以通过篡改系统调用入口进行恶意操作，破坏系统安全性。设计面向容器环境的 Linux 系统调用虚拟化系统对于容器的安全性非常重要。

文件访问控制模块：实现基于 inode 虚拟化的文件保护，在保证内核安全性与强制访问控制能力的同时，提升内核的运行效率，提高系统整体的文件访问速度。

内核定制工具：用于构建 vkernel 模块的工具。它分析容器镜像的特点，并根据 seccomp、apparmor 规则自动构建 vkernel 模块。

二、面向容器的内核模块隔离系统

2.1 背景简介

目前，在 Linux 环境下，容器主要采用 Linux 内核提供的命名空间和控制组机制实现隔离和限制。但是本质上，Linux 内核仍然是被所有容器共享的。这种共享带来了高性能的同时，也给容器的隔离性带来了巨大的挑战。虽然内核已经面向容器增加了诸多隔离功能，但是内核层的隔离仍然不够完善，需要一步一步进行提升。

本系统主要关注于面向容器的内核模块的隔离。内核模块是一种支持按需加载或卸载的内核代码的机制，可以应用于为容器定制内核功能与访问权限，达到在内核层实现容器强隔离的目的。但是，由于内核共享，内核模块机制本身存在诸多隔离性与安全性的问题。本系统主要面向基于内核模块的容器隔离性解决方案，进行内核模块的隔离以及安全保护。这其中存在的挑战主要在于内核模块机制缺乏容器感知，如何隔离各个内核模块并与相应容器绑定。

为了克服容器场景下内核模块存在的挑战，我们实现了面向容器的内核模块隔离系统。该系统可以在内核中绑定容器与相应模块并克服模块重命名问题。面向容器的内核模块隔离系统充分结合现有内核模块管理机制和容器特性，针对不同的容器加载的内核模块提供安全的内核环境。

2.2 功能实现

2.2.1 内核模块虚拟化

容器通过系统调用与内核交互，在内核空间中，多个容器共享相同功能的内核模块，容器之间会产生竞争，功能模块性能受到严重影响。在这种情况下，需要给相应容器定制一个容器内核模块，提供内核功能和访问权限。一方面，在内核加载模块过程中需进行模块重名检测，如已存在名称相同的内核模块，则该内核模块不能被加载。另一方面，容器内核模块与容器对应，需要在创建容器实例前加载容器内核模块，并与相应的容器 init 进程绑定。如何在内核中绑定容器与

相应模块并克服模块加载重名检测机制是本功能需要解决的关键问题。

内核模块虚拟化是指为内核模块添加容器感知，能够隔离各个内核模块并与相应容器绑定，可以应用于为容器定制内核功能和访问权限。

具体做法是 runc 将容器内核模块文件读取到内存中，根据 ELF 文件格式进行解析，根据模块特定字段名称，定位到内存中容器内核模块名称字符串地址，接着获取容器进程的容器 id，将模块名称修改为模块名加上容器 id，保证容器内核模块名称的不重复。将模块中核心数据添加到 namespace 中，实现内核模块与容器的绑定。

2.3 测试方案与结果

在可加载内核模块的 vkernel 容器运行时的基础上，实现一个具有模块重命名的 vkernel 容器运行时。可以在多容器启动时，通过 vkernel 运行时的重命名功能，将模块名称修改为模块名加上容器 id，保证多个容器成功初始化容器相应的 vkernel 内核模块。

首先通过不具备重命名功能的 vkernel 运行时来启动 10 个容器和相应内核模块，不支持模块重命名，启动冲突，只能运行一个容器。

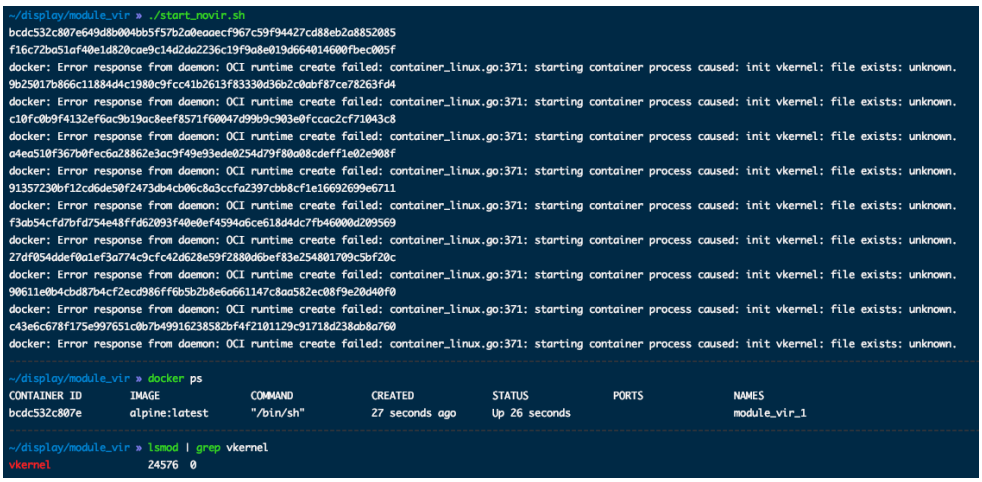

图 2-1 启动 10 个容器和相应内核模块（不支持模块重命名）

接着通过具有重命名功能的 vkernel 运行时启动 10 个容器和相对应的 vkernel 内核模块。成功启动 10 个容器和初始化 10 个相应的 vkernel 内核模块。

```
~ » lsmod | grep vkernel
vkernel_8f01be9afc88    24576  0
vkernel_aec9237e8365    24576  0
vkernel_c083cde12978    24576  0
vkernel_b4a0d511737b    24576  0
vkernel_0a599ecb1481    24576  0
vkernel_2fcf90011b4f    24576  0
vkernel_fd9e1e62ba23    24576  0
vkernel_2fcfb1bb2a49    24576  0
vkernel_d68406933545    24576  0
vkernel_937c1c40e3a4    24576  0

~ » docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
8f01be9afc88        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_10
aec9237e8365        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_9
c083cde12978        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_8
b4a0d511737b        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_7
0a599ecb1481        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_6
2fcf90011b4f        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_5
fd9e1e62ba23        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_4
2fcfb1bb2a49        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_3
d68406933545        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_2
937c1c40e3a4        alpine:latest       "/bin/sh"           4 hours ago         Up 4 hours                              module_vir_1
```

图 2-2 启动 10 个容器和相应内核模块（支持模块重命名）

三、面向容器的内核日志隔离机制

3.1 背景简介

在 Linux 系统中运行容器时，一直存在着系统全局资源隔离不彻底的问题。Linux 内核为了实现进程之间的隔离提供了若干机制，其中最为重要的就是命名空间机制（namespace）和资源管理控制组机制（Cgroup）。Namespace 机制用于资源的视图隔离，而 Cgroup 用于对进程消耗资源的限制。

内核日志是一个全局资源，没有封装在任何 namespace 中；同时，任何设备或进程产生的日志在结构上没有能身份标识，没有面向隔离的概念。任何具有超级用户权限的主体，包括主机和超级权限的容器，都能够读取所有的内核日志。处于安全性和隔离性的需求，操作系统需要给容器提供一种独占所有系统资源的“错觉”，使容器既不能访问其他容器的文件，也不能允许自身的文件被其他的容器访问。

所以当前的内核日志机制违背容器的需求，我们需要设计并实现一套面向容器的内核日志隔离机制，来做到对内核日志的彻底隔离。

3.2 功能实现

3.2.1 内核日志结构的改进

实现内核日志结构的改进，是为了使内核日志具备能够区别开不同 pid_namespace 的身份标识。

对内核日志结构的改进最终确定为重定义内核日志结构体，即在结构体中添加了一个专门用来存贮日志身份标识的成员变量：log_pid_ns，类型为无符号整型（与 pid_namespace 号相同）。

重定义之后的结构体要生效，必须在生成日志的过程中填充变量的值。

3.2.2 内核日志身份标识的实现

内核日志身份标识主要通过 Linux 系统的进程管理和 namespace 机制实现。通过使用进程管理系统中的一些函数，查询进程所属的 pid_namespace，将

pid_namespace 号作为内核日志身份的标识。

3.2.3 内核日志相关函数的改写

改写内核日志系统调用是整个内核日志隔离机制的实现过程中最重要同时也是工作量最大的一个部分。除了系统调用之外，还有几个相关的内核函数同样需要修改。 表 3-1 说明了需要修改的内核函数。

表 3-1 内核日志相关函数的修改

系统调用（标号）/内核函数	需要进行的修改
log_from_idx()	根据名称空间，返回内容；
devkmsg_read()	根据权限计算日志大小并复制到用户空间；
SYSLOG_ACTION_READ (2)	判断读取权限，然后读取；
SYSLOG_ACTION_READ_ALL (3)	根据权限读取，计算可读取日志的大小；
SYSLOG_ACTION_READ_CLEAR (4)	根据权限读取，计算可读取日志的大小；
SYSLOG_ACTION_SIZE_UNREAD (9)	返回日志中有权限读，但未读取的字符数
kmsg_dump_get_buffer()	判断转储权限，然后转储
kmsg_dump_get_line_nolock()	判断转储权限，然后转储
console_unlock()	根据读取权限判断是否释放控制台锁

3.3 功能测试

面向容器的内核日志隔离机制的功能测试主要是在两个版本内核上对两种读取内核日志方式的执行情况的测试。

测试的形式为对比测试。首先在没有实现隔离机制的原始 Linux-5.7 内核上进行容器日志的读写实验，记录其工作情况；然后在我实现了容器内核日志隔离机制的新版本内核上进行相同的测试，记录运行结果。通过前后对比验证功能的实现情况。

测试的内容主要是针对 Linux 系统中两种读取内核日志的方式：dmesg 命令和 cat /proc/kmsg 展开。其中，在测试 cat /proc/kmsg 命令的过程中，为了避免 rsyslogd 守护进程与该命令产生数据竞争，影响测试结果，我们需要手动杀死该进程。测试的对象为：宿主机、容器 A（名为 test，具有超级用户权限和插入内核模块权限）、容器 B（名为 test1，具有超级用户权限）。测试内容设计如下表 3-2 所示。

表 3-2 测试内容设计

原版本内核	① 在容器和主机上分别运行 dmesg 命令查看主机内核日志
	② 在容器中生成内核日志，重复步骤①
隔离版本内核	③ 在 vkernel 容器和主机上分别运行 dmesg 命令查看主机内核日志
	④ 在 vkernel 容器 A 中生成内核日志，重复步骤③
	⑤ 在 vkernel 容器 B 用 dmesg 查看内核日志

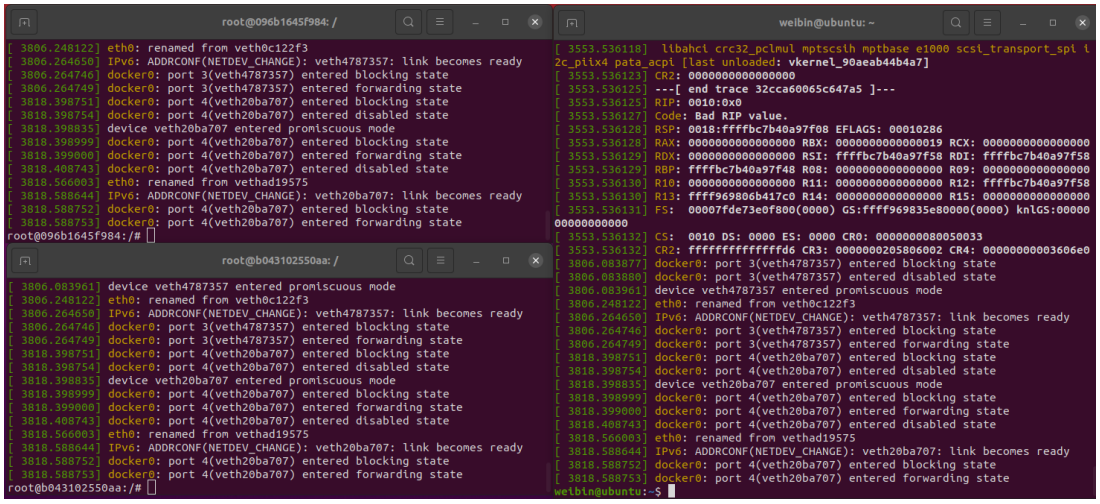

图 3-2(a) 原版本内核：容器和主机分别运行 dmesg 查看内核日志

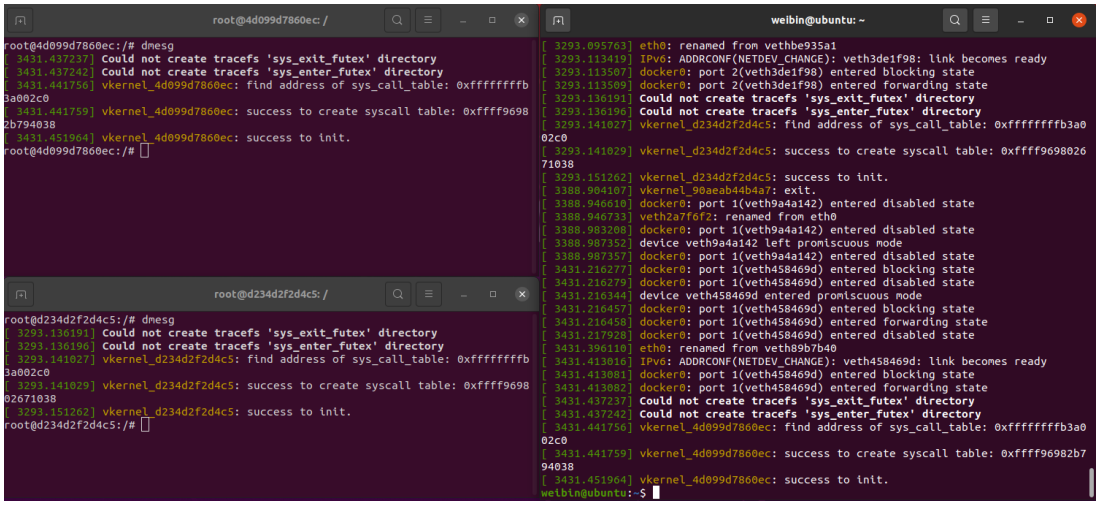

图 3-2(b) 隔离版本内核：容器和主机分别运行 dmesg 查看内核日志

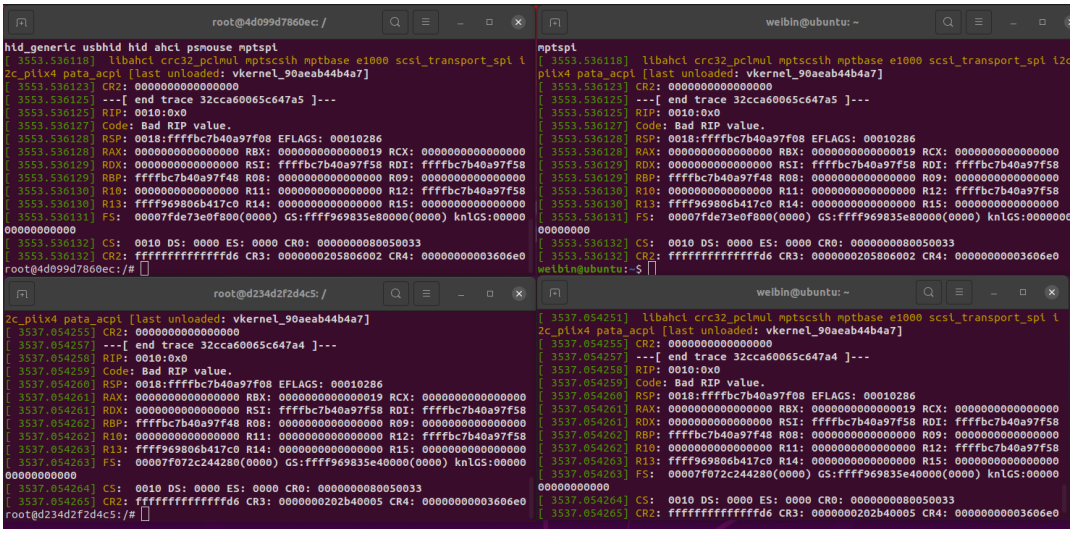

图 3-3 隔离内核版本：容器实时内核日志隔离效果

四、面向容器环境的 Linux 系统调用虚拟化

4.1 背景简介

随着云计算技术的发展，容器由于其轻量性和高效率等特点，受到越来越广泛的重视，但是由于所有的容器共享主机系统的内核，容器在安全性方面的防护仍旧较弱，具有许多的安全隐患。

在诸多安全隐患中，系统调用的共享对容器安全性影响尤其明显。Linux 内核只提供了统一的系统调用入口以及实现了一致的系统调用。一方面，由于系统调用是共享的，当多个容器竞争调用相同系统调用的时候，产生的剧烈竞争比较容易降低该系统调用的性能。另外一方面，恶意的容器可以通过篡改系统调用入口进行恶意操作，或者通过触发系统调用漏洞产生提权甚至引发宕机。除此之外，某些系统调用申请的资源是有限的，恶意容器可以通过恶意占有系统调用资源来让系统拒绝为其他容器提供服务。

现有的容器虚拟化技术，如 gVisor 和 Kata Containers 等，虽然通过使用用户态内核或者 KVM 虚拟化的方式提高容器的安全性和隔离性，但它们都引入了不容忽视的开销，无法满足现有的容器轻量性和隔离性的需求。

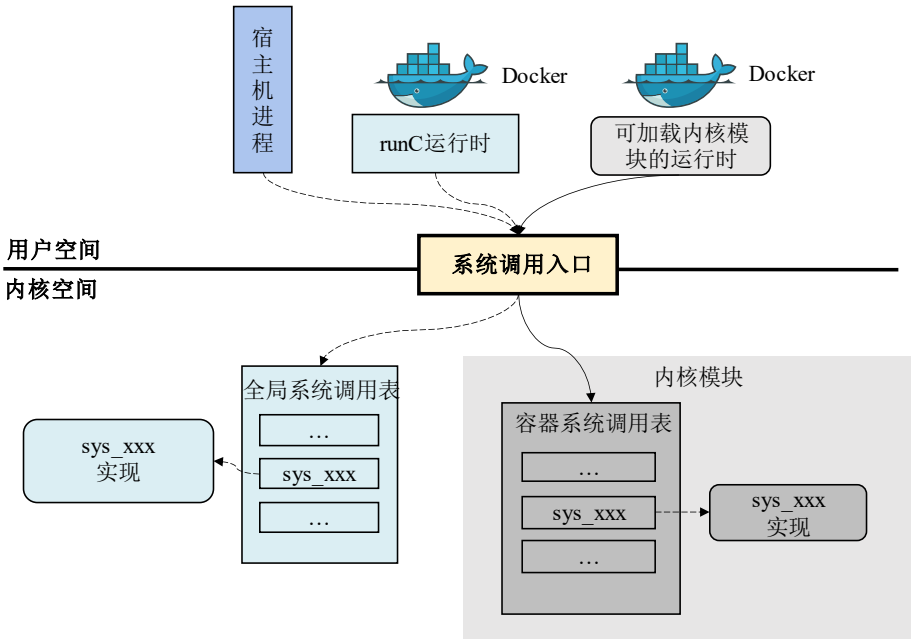

图 4-1 面向容器环境的 Linux 系统调用虚拟化实现架构图

针对以上问题，提出设计一种面向容器环境的 Linux 系统调用虚拟化系统，

同时借助内核模块灵活的可插拔性进行实现。

4.2 功能实现

4.2.1 可加载内核模块的容器运行时实现

面向容器环境的 Linux 系统调用虚拟化系统以内核模块的方式实现。为了加载内核模块，需要在容器开始启动后、容器内部应用进程真正运行前，调用内核模块，完成相应功能的初始化以及模块与容器进程的绑定。同时，内核模块要与容器一一对应，伴随着容器的启动停止而动态加载与销毁。

4.2.2 双重 capabilities 保护

容器创建后，只会使用到少量的 capabilities（默认 14 个）。内核内部有一套自己的 capabilities 安全机制。通常情况下，这些有限的 capabilities 集合在内核的 capabilities 机制下可以保证大部分容器的正常运行，同时也能提高容器环境的安全性。但目前公开的一些漏洞显示，在容器环境下，这些漏洞可以通过利用部分内核 bug 进行提权和逃逸，从而摆脱容器的安全限制，进而可能危害宿主机。

vkernel 运行时在容器应用进程创建前会记录容器初始的 capabilities。之后，容器的所有需要进行权限验证的行为会在内核 capabilities 验证的基础上进行 vkernel 的二次验证，保证容器的所有行为都在初始时的 capabilities 约束集合内，避免越权的行为发生。

4.2.3 容器系统调用表虚拟化实现

容器的系统调用表通过内核虚拟化不同的系统调用入口，移除了容器对全局系统调用表的访问权限，代之以每个容器拥有的位于内核空间不同区域的独立系统调用表。内核模块在初始化后，会在内核空间中开辟一块独立的内存区域，用于存放容器的系统调用表。

容器的系统调用表与全局的系统调用表结构相同，但部分内容有所不同。对于需要进行参数检测的系统调用，会修改系统调用表的表项，对系统调用函数进行二次封装；对于需要屏蔽的系统调用函数，直接返回-1；对于涉及到隔离的子

系统资源，会修改重定向该系统调用函数，将其指向内核模块内部的子系统；其余的则默认使用原来的全局系统调用函数。

4.2.4 内核系统调用资源虚拟化实现

某些系统调用申请的资源是有限的，在容器环境下，多个容器同时执行这类系统调用会对这些资源产生严重的竞争，从而影响各自的性能，甚至恶意容器能够抢占全局资源从而影响其他容器的正常运行。因此，实现内核系统调用资源虚拟化，从而减少甚至避免容器之间对系统调用资源的竞争，是必要的。

系统调用资源虚拟化的实现以隔离futex子系统为例。futex（fast userspace mutexes）是内核提供的一种进程同步机制，所有使用futex进行同步的容器进程都会用到futex子系统。但是，内核只在初始化时开辟了有限空间大小的hash表来管理futex锁，因此需要实现futex系统调用资源的虚拟化。

模块内部的futex子系统的实现逻辑与内核全局的futex子系统逻辑相同，难点在于futex部分的代码和内核其他功能具有比较高的耦合性，需要将它们解耦出来，为每个vkernel容器分配专属的futex子系统。除此之外，在实现隔离futex子系统的过程中，我们发现对futex的调用分为两种：

1) 应用程序通过系统调用 sys_futex 调用 do_futex

2) 内核其他路径直接调用 do_futex

系统调用表虚拟化无法隔离第二种 futex 调用路径，因此不能实现对 futex 子系统的彻底隔离。针对这个问题，我们使用 hook 机制将内核模块对 do_futex 的直接调用转换为对自定义 futex 的调用，从而实现内核系统调用资源的虚拟化。

功能的实现目标是在内核模块中虚拟化出这部分子系统的功能，之后容器的所有 futex 系统调用均使用自己独有的 futex 子系统。

4.3 测试方案与结果

4.3.1 可加载内核模块的容器运行时测试

在原生 runc 运行时的基础上，实现一个可加载内核模块的 vkernel 容器运行时。可以在容器启动时，通过指定自定义的 vkernel 运行时，成功初始化容器的

vkernel 内核模块。

该测试使用目前开源的 LTP（Linux Test Project）11 工具进行实现。LTP 工具旨在向开源社区提供一系列 Linux 平台的测试集，以验证 Linux 系统的可靠性、健壮性和稳定性。测试所使用的 LTP 工具版本为 20210121，容器镜像为 Ubuntu18.04。测试分别使用默认 runc 运行时的容器以及启用系统调用虚拟化功能的容器进行对比，在两个容器中分别使用“./runltp -Q -f syscalls”命令来自动执行所有的系统调用测试项。

表 4-1 LTP 在默认 runc 运行时容器和启用系统调用虚拟化功能的容器中的测试结果

	Total Tests	Total Skipped Tests	Total Failures
默认runc运行时容器	1344	155	303
开启系统调用虚拟化的容器	1344	155	303

4.3.2 容器系统调用表虚拟化测试

容器系统调用表虚拟化通过不同容器调用相同系统调用时产生不同的反应进行表现。由于 vkernel 运行时可以对系统调用表中的系统调用函数进行一些自定义的封装与修改，因此，不同容器使用的同一个系统调用可能会有不同的反应，比如添加一些容器自定义的内核日志输出，以及屏蔽或放开某些系统调用。

测试时，创建多个容器，选取比较典型的 uname 系统调用进行演示。部分容器会允许执行 uname 系统调用，而其他容器不允许执行 uname 系统调用。结果就是，不允许执行 uname 系统调用的容器由于无法正确获取到 hostname 等系统信息，这部分内容会无法显示。同时，可以在 dmesg 中看到不同容器 uname 系统调用封装的自定义内核日志。

4.3.3 系统调用资源虚拟化测试

针对容器系统调用涉及到的内核资源虚拟化的测试，使用 Linux 的 Perf 测试工具对 futex 子系统进行测试。Perf 工具能够创建指定数量的线程阻塞在特定的 futex 哈希队列 bucket 上，我们对 Perf 工具进行了增强，使得其能够创建指定数量的线程阻塞在所有的 bucket 上。在实际的测试中，针对不同的容器运行时，

我们首先启动大量的压力测试容器，在压力测试容器中创建大量线程，通过 futex 子系统的 wait 操作，阻塞在 futex 的哈希队列上；其次我们启动目标容器，目标容器执行 futex 的 wake 操作，将线程从 futex 的哈希队列上唤醒，以此来模拟容器高密度部署时的内核资源竞争环境。我们测试了平均 50 个线程阻塞在一个 futex 队列的 bucket 上时目标容器内线程的唤醒时间，以此作为不同容器运行时针对内核软资源隔离程度的衡量指标。测试结果如图 4-2 所示。

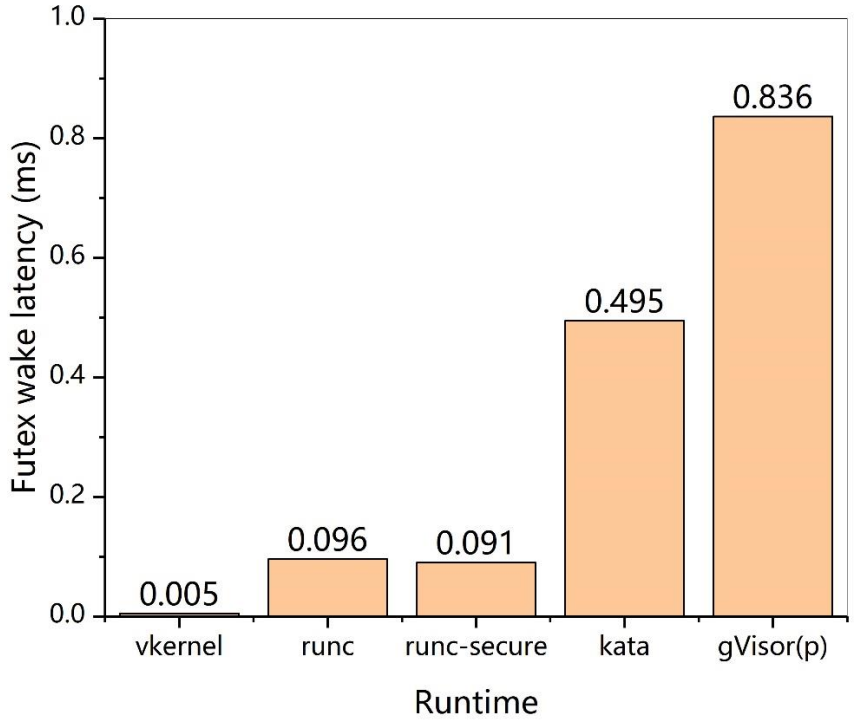

图 4-2 使用不同容器运行时时目标线程唤醒延迟

五、基于 inode 虚拟化的文件访问控制模块

5.1 背景简介

inode 虚拟化是 vkernel 项目的核心组成部分，属于 vkernel 的安全防护模块，主要实现的是基于 inode 虚拟化的文件保护。该模块的背景是目前 Linux 的内核强制访问机制如 AppArmor、SELinux 等在为文件访问提供审核管控时，所有文件在被访问时都需要进行相应规则的权限检测，会产生较大的性能开销，资源利用率低。另外，这些文件访问控制也有较为复杂的一套匹配规则，在面临大量的文件访问操作时会带来较大的性能开销，而且在容器内部用户默认拥有 root 权限，可以通过更改 profile 文件来更改 AppArmor 等的相应规则，具有安全隐患。

为了解决上述问题，该模块基于 vkernel 高效高安全性的设计思想，提出了一种基于 inode 虚拟化的文件保护机制，利用内核中已有的权限检测机制，针对 inode 进行虚拟化，只对需要保护的文件进行访问的审核控制，在保证内核安全性与强制访问控制能力的同时，大大地提升了内核的运行效率，提升了系统整体的文件访问速度。

5.2 功能实现

5.2.1 整体架构

为尽量保证 Linux 内部的数据结构不变，选取 inode 中的 i_mode 字段中未被使用的比特位作为标识位，用于标识该文件或目录是否在 vkernel 中有相应的权限限制，而每个容器的 vkernel 在初始化时都会初始化一个哈希表，key 为文件 inode 号，value 为相应文件的访问权限，并根据用户自定义的配置文件，将权限信息存储在该哈希表内。当文件被访问时，先检查相应标志位，当标志位为 0，文件在 vkernel 中并未被限制访问，直接通过，当标志位为 1，文件在 vkernel 中有访问限制，则查询哈希表，进行权限检测。

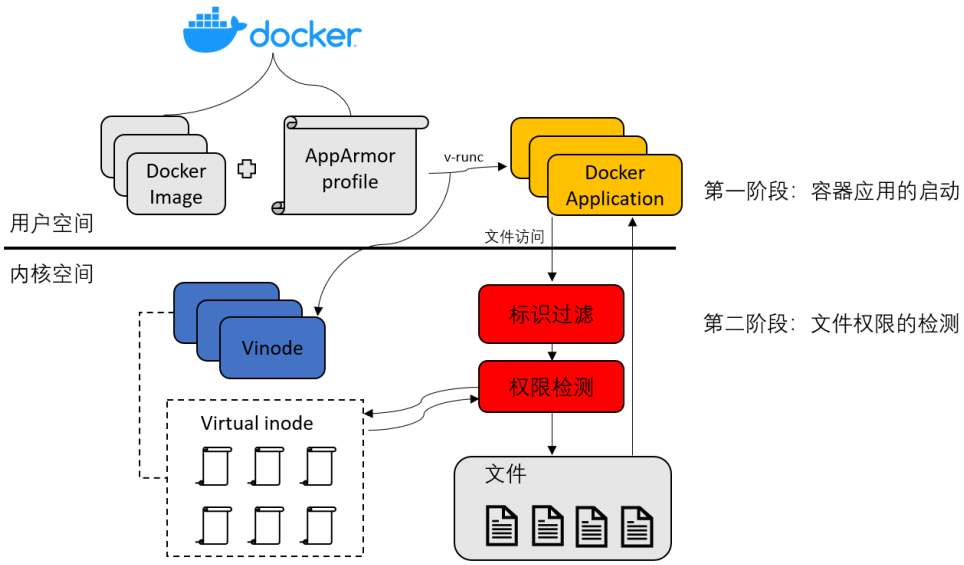

图 5-1 inode 虚拟化整体架构

5.2.2 对单个文件的权限检测

取一位 i_mode 未被使用的比特位，1 表示 vkernel 内有该文件的权限限制，0 表示 vkernel 内无权限限制，在 Linux 内核中的通用权限检测函数 generic_permission 中，添加基于 inode 虚拟化的 vkernel 权限检测，将文件访问申请的权限与 vkernel 内哈希表中的信息进行匹配检测。

5.2.3 对目录的权限检测

在 generic_permission 中对目录的 inode 的权限检测只是针对该目录的权限进行限制，而并非对目录下的文件进行相应的权限限制，结合实际应用中针对目录下文件的权限限制多是针对 procfs、sysfs、cgroupfs，于是取另一位 i_mode 未被使用的比特位用于表示目录在 vkernel 内是否有权限限制，然后在初始化一张新的哈希表用于存储目录的权限控制信息，在 generic_permission 中，针对 procfs、sysfs、cgroupfs 文件系统的文件，会逐级向上查看目录是否有相应的权限进行检测。

5.3 测试方案与结果

5.3.1 测试环境

创建两个容器，一个 runc 启动的普通 Ubuntu 容器，一个装载 vkernel 的 Ubuntu 容器，后者在 vkernel 中应用下表中的权限信息。

表 5-1 测试文件/目录的权限设置

文件/目录	权限(rwx)
/usr/share/doc/adduser/copyright	111
/usr/share/doc/apt/copyright	101
/usr/share/doc/bash/copyright	011
/usr/share/doc/bzip2/copyright	001
/sys/fs/cgroup/cpu	000

5.3.2 对文件的权限检测

在普通 Ubuntu 容器和装载 vkernel 的 Ubuntu 容器中分别对表 4-1 中的 /usr/share/doc/adduser/copyright 、 /usr/share/doc/apt/copyright 、 /usr/share/doc/bash/copyright、/usr/share/doc/bzip2/copyright 文件进行读写操作，测试结果如图 5-2 所示。可以看到，在普通容器内均能正常读写，在 vkernel 容器内，呈现出与自定义权限设置一致的结果。

```
root@e39a8faa2eb9:/# ./test.sh
1. adduser
2. apt
3. bash
4. bzip2
```

图 5-2(a) 普通 Ubuntu 容器中文件读写情况

```
root@0a6eaba974f8:/# ./test.sh
./test.sh: line 3: /usr/share/doc/apt/copyright: Permission denied
./test.sh: line 5: /usr/share/doc/bzip2/copyright: Permission denied
1. adduser
Apt is copyright 1997, 1998, 1999 Jason Gunthorpe and others.
Apt is currently developed by APT Development Team <deity@lists.debian.org>.

License: GPLv2+

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

See /usr/share/common-licenses/GPL-2, or
<http://www.gnu.org/copyleft/gpl.txt> for the terms of the latest version
of the GNU General Public License.
cat: /usr/share/doc/bash/copyright: Permission denied
cat: /usr/share/doc/bzip2/copyright: Permission denied
```

图 5-2(b) 装载 vkernel 的 Ubuntu 容器中文件读写情况

5.3.3 对目录的权限检测

对/sys/fs/cgroup/cpu 内的多个文件，分别在两个容器内进行读取操作，测试结果如图 5-3 所示。可以看到，在普通容器内可正常读取，在 vkernel 容器内读取显示 permission denied。

```
root@e39a8faa2eb9:/sys/fs/cgroup/cpu# cat cpu.shares
1024
root@e39a8faa2eb9:/sys/fs/cgroup/cpu# cat cpu.stat
nr_periods 0
nr_throttled 0
throttled_time 0
```

图 5-3(a) 普通 Ubuntu 容器中目录下文件读写情况

```
root@0a6eaba974f8:/sys/fs/cgroup/cpu# cat cpu.shares
cat: cpu.shares: Permission denied
root@0a6eaba974f8:/sys/fs/cgroup/cpu# cat cpu.stat
cat: cpu.stat: Permission denied
```

图 5-3(b) 装载 vkernel 的 Ubuntu 容器中目录下文件读写情况

六、基于容器镜像的最小化内核定制工具

6.1 背景简介

与运行自己的操作系统的虚拟机相比，用户可以在主机的同一操作系统内核之上启动多个容器，这使得容器更加轻巧，有着更好的资源利用率和更好的性能。但是，容器性能的提升是以隔离度较弱为代价的。由于主机上的容器都共享同一内核，因此容器彼此之间的隔离是进程级别的，只能由底层操作系统内核通过软件机制来保证。因此，如果有攻击者有权通过某个容器来访问主机的内核并对内核的漏洞加以利用，就会对主机和其他容器的安全带来极大的威胁。

尽管操作系统提供了严格的软件隔离机制试图解决这一问题，例如 Linux 中用以进行细粒度进程权限控制的 Capability 和用于资源视图隔离的 namespace 等等，但是，目前的 namespace 仍不支持对系统调用进行视图隔离，因此，恶意的租户仍然可以通过系统调用访问共享内核，并利用内核漏洞绕过这些隔离机制。例如，“waitid”系统调用中的漏洞（CVE-2017-5123）允许恶意用户运行特权升级攻击，并逃脱容器以获得对主机的访问权限。

vkernel 本质上是一个内核模块，旨在替代 Linux 内核中 seccomp 等的安全机制，减少性能开销，增强隔离性。每个容器都有一个自己专用的 vkernel 为自己服务。容器只能通过 vkernel 来访问主机资源，因此可以在 vkernel 中对容器的行为进行一些限制，提高容器的隔离性。

但是，vkernel 是专用的，不同镜像，甚至同一镜像的不同容器，他们所需要的内核功能也是不一样的。如果每次启动容器时都要手动修改代码，编译模块，那用户的使用体验就会大大降低，也不利于实际应用和自身的推广。所以需要一个 vkernel 的自动构建工具，用户只需要提供容器镜像信息及其对应的配置文件，就能分析容器所需要的最小内核功能，并根据分析结果，自动生成容器专属的 vkernel 模块的代码，构建出目标容器专用的 vkernel 模块。

6.2 功能实现

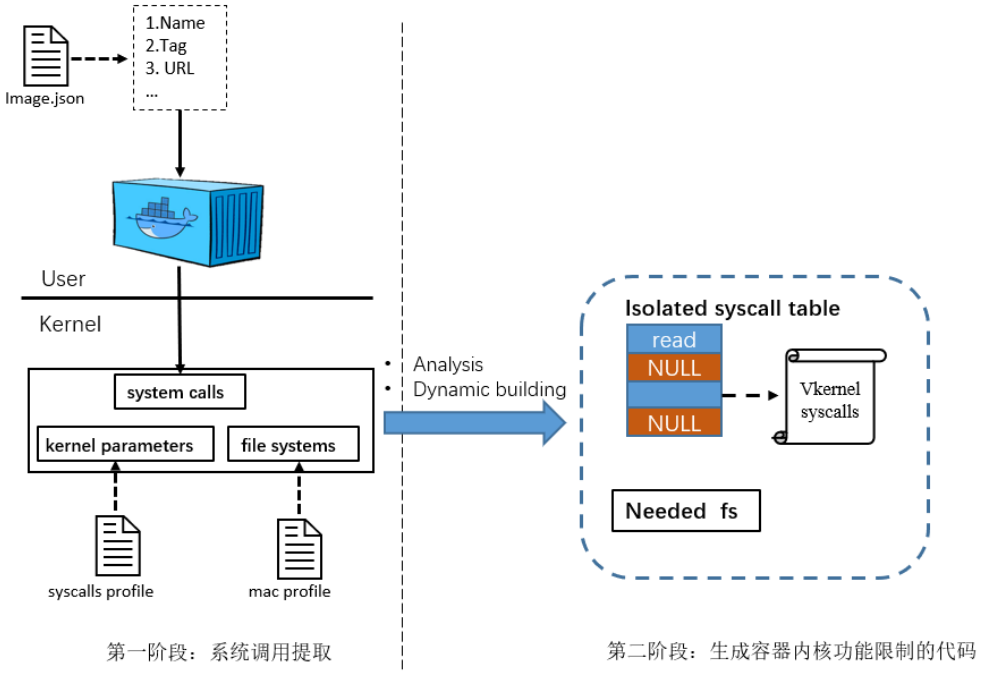

图 6-1 基于容器镜像的最小化内核定制工具架构图

6.2.1 容器镜像系统调用提取

该模块用以对用户提供的容器镜像进行动态分析，以提取其所需要用到的系统调用，供后面生成限制性策略的模块使用。

具体实现上，首先需要对用户提供的包含容器镜像信息的 json 文件进行解析，获取容器镜像的名称，版本号，URL 等信息，随后将主机上现存的容器全部停止并删除，再启动目标容器，保证主机上只有目标容器在运行。然后启动 sysdig 监控程序，设定运行时间，并将这段时间内监控的结果以文本的形式保存在指定目录下。最后对监控结果进行分析，提取与目标容器相关的条目，截取系统调用相关的字段并保存。

6.2.2 自动构建 vkernel 模块

工具会对第一阶段的容器镜像分析结果进行解析，结合用户提供的容器的系统调用和文件访存的配置文件，并生成对应的限制性策略的代码。

具体来说，针对系统调用配置文件，由于文件定义了容器允许访问哪些系统调用，因此只需要对配置文件进行解析，并结合第一阶段提取的系统调用，两者取并集，就能得到允许容器访问的系统调用的结合。其次，配置文件还定义了哪些系统调用在访问时需要进行参数检测，因此系统还会对这些系统调用进行重新封装，插入参数检测的代码，用重新封装好的系统调用去替换系统调用表上的默认的系统调用，实现限制系统调用的策略。而针对文件访存的配置文件，文件中的条目定义了容器对于某个路径下的文件是否具有各类权限（读、写、链接权限等），因此工具会对配置文件进行解析。由于文件路径是 glob 正则表达式，因此首先需要对路径进行解析，得到 Linux 下的标准路径，并且根据路径去读取出该路径下的所有文件或文件夹，随后使用位图来表示这些文件对应的权限。在得到这些信息之后，将文件及对应的权限以键值对的形式存入哈希表中，并自动生成哈希表初始化的代码，容器在访问这些文件时，需要先去哈希表中进行查询，确认自己持有相关权限才可访问，进而达到限制容器对文件进行访存的目的。当系统调用限制和文件访存限制两部分代码生成之后，工具会自动执行编译指令，对 vkernel 源码进行编译，生成.ko 模块文件，供容器运行使用。

七、真实应用性能测试

除了 Docker 默认使用的 runc 运行时以外，当前广泛使用的符合 OCI 标准的运行时包括基于进程沙箱的 gVisor 和基于轻量级虚拟化的 Kata Containers。作为对比，本章节对使用不同容器运行时启动相应的容器实例，并对容器实例中的应用程序性能进行测试。

7.1 NGINX 性能测试

Nginx 是一款轻量级的高性能 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，在 BSD-like 协议下发行。其特点是占有内存少，并发能力强，Nginx 的并发能力在同类型的网页服务器中表现较好，其属于 I/O 密集型负载。在 Nginx 性能测试中，分别使用 runc 运行时（runc）、采用了 Apparmor 的 runc 运行时（runc-secure）、Kata Containers 运行时（kata）、ptrace 模式下的 gVisor 运行时（gVisor(p)）、kvm 模式下的 gVisor 运行时（gVisor(k)）测试容器实例中 Nginx 负载的吞吐量，作为采用 vkernel 时容器实例中 Nginx 负载吞吐量的对比。测试工具选择 Apache Benchmark，测试配置采用 100 并发量发送 100000 个请求。测试结果如图 7-1 所示。

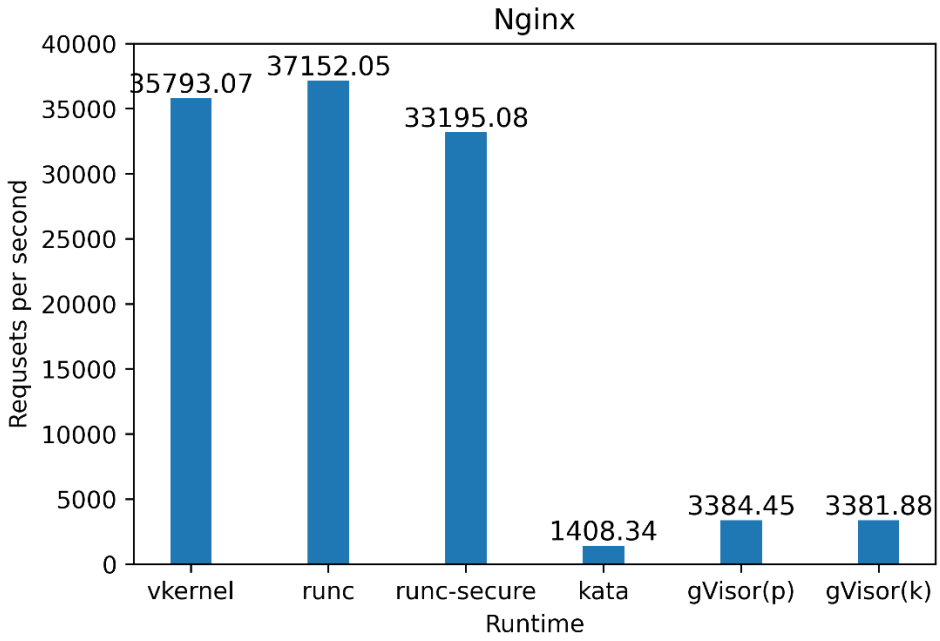

图 7-1 不同运行时启动容器实例中 Nginx 的吞吐量

7.2 PWGEN 性能测试

pwgen 是 Linux 中常用的密码生成工具，用于生成不同长度的安全密码，被广泛应用于 Serverless 场景下的密码生成函数，pwgen 属于计算密集型负载。与 6.1 节相似，使用不同的运行时启动运行 pwgen 负载的容器实例，测试 pwgen 生成 1024 个长度为 100 的密码的生成时间。测试结果如图 7-2 所示。

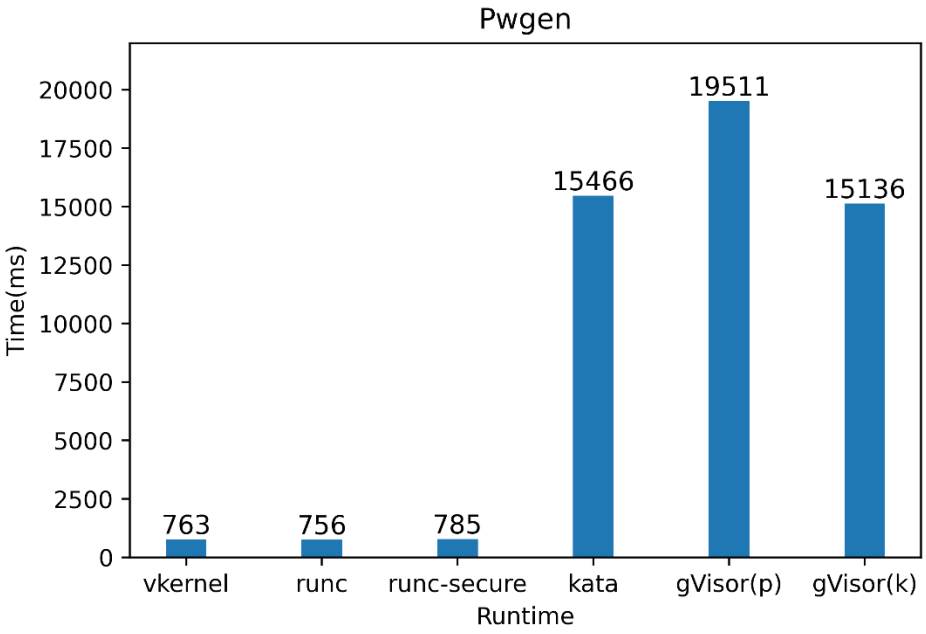

图 7-2 不同运行时启动容器实例中 pwgen 的密码生成时间