



大家好，我们现在开始我们组的报告。

CONTENTS

- Part I - rVisor 简介
- Part II - rVisor 设计思路
- Part III - rVisor 具体实现

我们组的报告将围绕这样三个部分来逐步展开，我们先从 rVisor 简介开始。



RVISOR - 内核中的安全容器解决方案



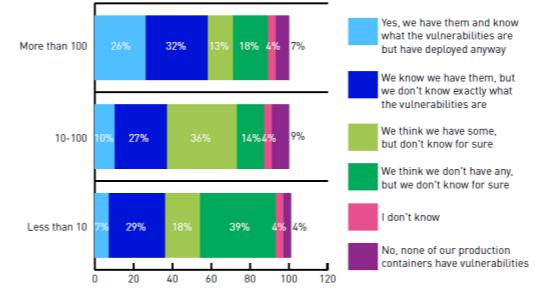
我们 rVisor 致力于在内核中实现更为轻量的安全容器。在具体介绍之前，我们先来看传统容器的问题。

传统容器的局限性 - 安全

- 随着容器技术的不断发展，传统容器隔离性不足的问题逐渐暴露出来。
- 根据 Tripwire 做过的一次调查报告，有将近一半的大型公司任务容器的安全性有一定的问题。

Those with the most containers in production have ignored security issues

Do you currently have vulnerable containers deployed in production at this time?
By # of containers in production

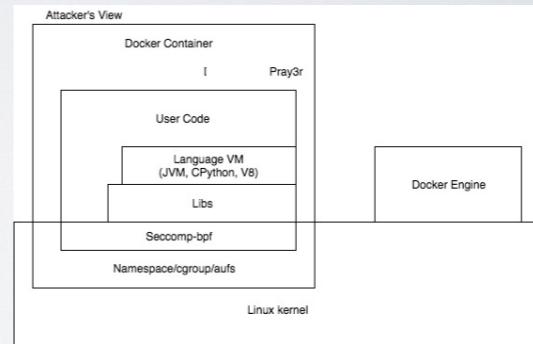


首先，随着容器技术的不断发展，传统容器隔离性不足的问题逐渐暴露出来。

我们先来看 Tripwire 做过的一次调查报告。可以看到 Tripwire 调查了一些容器专家，发现有将近一半的大型公司认为他们容器的安全性有一定的问题。

为什么传统容器缺乏安全性？

- Namespace/cgroup: 由内核提供，无法彻底与内核隔离。
- 由于容器与外部系统同时使用一个 Linux 内核，一旦 Linux 内核出现问题，就可以利用内核漏洞实现容器逃逸。



所以为什么说传统容器缺乏安全性呢？

就拿用Linux的 Namespace/Cgroup 实现的容器举例子，这种容器的架构如是这样的。

Namespace/Cgroup 是内核的一个部分，容器仍然使用主机的 Linux 内核，他解决不了Linux内核中隔离性差的问题，攻击者可以利用Linux内核的漏洞来实施攻击，进而实现我们所说的容器逃逸（简单的说从容器内部获得对容器引擎的控制）。

而相比较而言，虚拟机往往具有比较好的隔离性，攻击者直接面对的是虚拟机的内核，他很难对虚拟机外部获得控制。

如何解决这种安全问题？ - 多层抽象

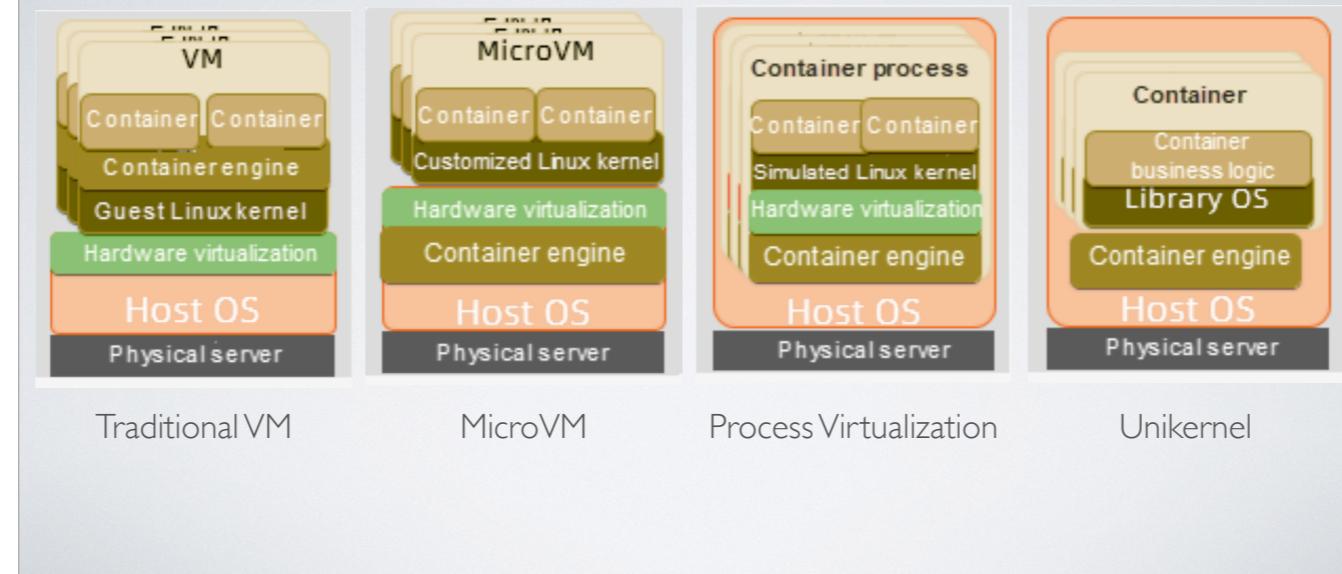
- “only real solution is to admit that bugs happen, and try to mitigate them by having multiple layers of security, so that if you have a hole in one component, the next component will catch it.”

— By Linus Torvalds (LinuxCon)

我们解决这个问题的大体思路是这样的：

针对Linux的安全性问题 Linus 给出了一个解决方案，就是你要不然就允许不安全的错误发生，要不然就建立多个层次来缓解这种安全问题，Linus 认为如果一个组件有一个漏洞，就在这个组件之上增加一层抽象，而我抽象上层的组件能够捕获到它，只要这样层层封装，就可以增强他的安全性。这意味着我们要解决我们之前提到的容器的安全问题，必须建立新的一层抽象。

Add Multiple Layers - Solutions After 2018



为了增加一层新的抽象，我们来看一下目前主流的几个思路：

最左边的做法呢，是传统意义上的虚拟机，它可以实现比较好的隔离性，但是它会对我们的整体运行很大的开销，而且其配置和启动也比较复杂，不如容器使用起来那样方便。

最右边的是我们大家非常熟悉的，Unikernel，它让应用自己带上自己的内核，它有更小的开销也有非常优秀的隔离性。但这样做就意味着要对应用程序作出修改等兼容性问题，这种兼容性问题成为 Unikernel 发展的一个阻碍。

然后我们当前主流的实现容器的是中间两种思路。

首先就是这个MicroVM，它是一种轻量级的虚拟机，使用裁剪过的内核，尽可能的减少传统虚拟机所带来的开销，这时MicroVM在保持容器效率的基础上实现了良好的兼容性，用户直接面对虚拟机的内核，无法对系统内核作出攻击。

另一个方法呢，是这个进程虚拟化，进程虚拟化的方法使用一个特定的内核来运行 Linux 二进制文件，它直接虚拟化Linux的运行环境，而不使用原来完整的 Linux镜像。可以在保证轻量性的同时为Linux应用提供尽可能大的兼容性。同时，应用程序面对着的是全新的内核，也难以对原有的内核进行有效的攻击。（轻量型的安全容器）

Implements of These Solution

Traditional VM	MicroVM	Process Virtualization	Unikernel
<ul style="list-style-type: none">• KVM	<ul style="list-style-type: none">• Kata Container: • 很好的安全性• 运行效率• 较大的内存占用• 较长的启动时间	<ul style="list-style-type: none">• gVisor • 很好的隔离性• 系统调用效率不高• 很小的内存占用和启动时间	<ul style="list-style-type: none">• Nabla Container

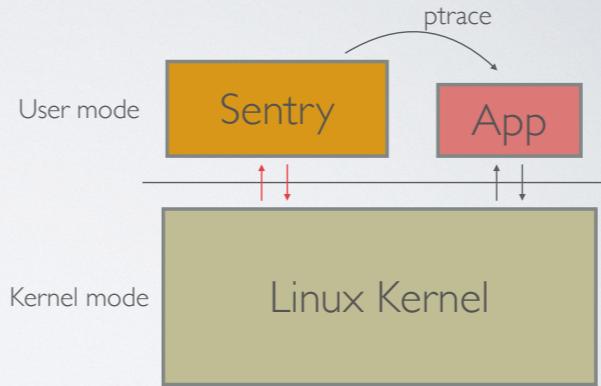
相关的实现有很多，这里我们经常谈到的主要是 Kata Container 和 gVisor。

Kata Container 是MicroVM的一个经典的实现实现，它提供了一个MicroVM，并且有专门提供给 Kubernetes 使用的接口，有比较好的安全性和运行效率，现在已经开始逐步使用。

而 gVisor 是基于进程虚拟化的容器实现，他拥有很好的隔离性，很小的内存占用和启动时间，但是系统调用效率不高，这是我们解决的重点问题。

gVisor - Limitations

- 一方面, gVisor 是使用 GO 语言编写的操作系统内核, 而现有的 Go 语言的 GC 并不适宜于完成操作系统内核的编写。
- 另一方面, Sentry 和 Linux 进行交互时会产生大量内核态到用户态到上下文切换。



让我们来谈一下 gVisor 架构中所存在的问题。

一方面, gVisor 是使用 GO 语言编写的操作系统内核, 而现有的 Go 语言的 GC 并不适宜于完成操作系统内核的编写。

另一方面, gVisor 使用 Linux 提供的 ptrace 实现, ptrace 是一个用于调试进程的系统的调用, 主要用于 gdb 和 strace 这样的调试分析工具。ptrace 可以将一个进程完全控制, 捕获应用程序的所有系统调用和事件, 但是 ptrace 在效率上有一定的问题。

应用程序在调用系统调用时, 会进入内核态, 然后内核会唤醒 Sentry 进程让 Sentry 进程执行, Sentry 进程执行完后, 再进行一次系统调用切换到内核态, 内核态继续转到应用程序中运行。整个过程中多出了很多次上下文切换, 这个开销在需要使用大量系统调用的、这种IO密集型的服务端容器运行环境内是难以接受的, 我们先来看具体的 benchmark:

gVisor -Benchmark

- These are benchmark conducted by Xu Wang & Fupan Li (Kata Container)
- Test Nginx : ab -n 50000 -c 100 http://10.100.143.131:8080/

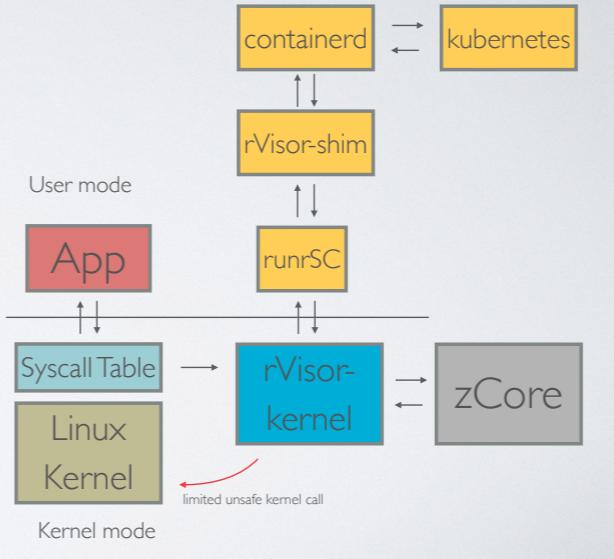
	runC	Kata*	gVisor		
Concurrency Level	100	100	100		
Time taken for tests	3.455	3.439	161.338 seconds		
Complete requests	50000	50000	50000		
Failed requests	0	0	0		
Total transferred	42250000	42700000	42250000 bytes		
HTML transferred	30600000	30600000	30600000 bytes		
Requests per second	14473.73	14541.18	309.91 [#/sec]	(mean)	
Time per request	6.909	6.877	322.677 [ms]	(mean)	
Time per request	0.069	0.069	3.227 [ms]	(mean, across all concurrent requests)	
Transfer rate	11943.65	12127.12	255.73 [Kbytes/sec]	received	

这里有一个对 runC、Kata 和 gVisor 一组 benchmark，可以看到 gVisor 在 Nginx 上比 runC 和 Kata 慢了将近 50 倍。

这个数据足以显示出 gVisor 的在系统调用密集的应用中性能与 Kata 和原本的 docker、runc 还是有较大差距。

rVisor - 我们的解决方案

- rVisor 使用 Rust 语言编写，具有和 Go 语言同样好的安全性
- rVisor 将所有内存的不安全性显式地用 unsafe 表示出来，同时将不安全的系统调用降低到最少，这样可以实现与 Go 同等的安全性。
- rVisor 作为一个内核模块，它使用系统调用劫持的方法实现安全沙箱环境。
- 同时，gVisor 在进行 ptrace 的同时，会进行多次进程切换，而 rVisor 使用系统调用劫持的过程中 Linux 用户线程不会改变，只会进行最简单的上下文切换，总而使 rVisor 拥有可以媲美原生应用的性能。



面对这样的问题，我们 rVisor 给出如下的解决方案。

我们知道，安全性并不是一个绝对的概念，gVisor 对系统调用的使用进行了严格的限制，他仅仅使用了不到20条系统调用来与 Linux 交互，rVisor 使用 Rust 语言，将所有的不安全的系统调用都用 unsafe 封装起来，可以获得与 gVisor 同等的安全性。

rVisor 作为一个内核模块，它使用系统调用劫持的方法实现安全沙箱环境。rVisor 可以完整地为应用程序提供服务，可以与原 Linux 主机做出很好的隔离。

同时，rVisor 在内核中实现，使用系统调用劫持的方法，这样在执行系统调用时，只会进行一次简单的上下文切换，可以拥有媲美原生应用的性能。

RVISOR 目标

- 在 gVisor 基础上，实现可以媲美原生应用性能的轻量的安全沙箱系统。
- 相比 Kata，实现更短的启动时间和更少的内存占用。
- 公共云服务中提供容器服务，在容器中安全运行不可信代码。
- 尝试应用 Serverless 的计算场景。

总的来说，我们 rVisor 致力于借助 Rust 语言和内核中劫持系统调用在性能上的优势，在 gVisor 的基础上实现安全、媲美原生应用性能的轻量安全容器。相比 Kata 而言，实现更短的启动时间和更少的内存占用。

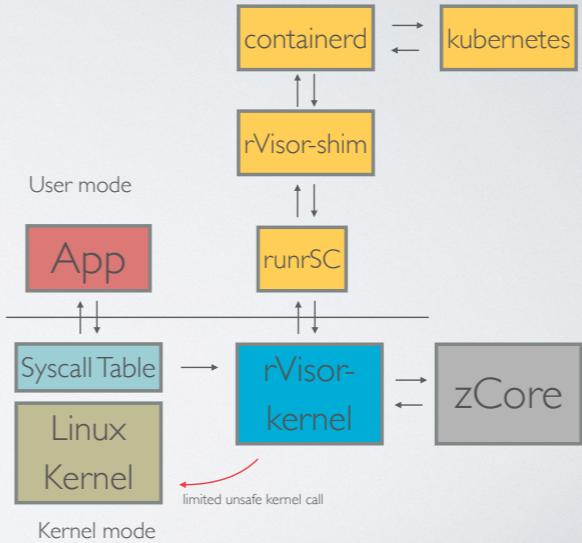
rVisor 可以提供非常轻量的容器服务，可以应用于 Serverless 的计算场景。



然后我们讲一下 rVisor 的设计思路。

rVisor - 整体架构

- rVisor-kernel: rVisor 的核心部件，实现容器的基本功能。
- zCore: 我们为 rVisor 专门改写的内核。
- runrSC: rVisor 的命令行客户端，为 rVisor-shim 提供接口。
- rVisor-shim 将我们整体架构与 containerd 连接



rVisor-kernel: rVisor 的核心部件，实现容器的基本功能。

zCore: 我们为 rVisor 专门改写的内核。

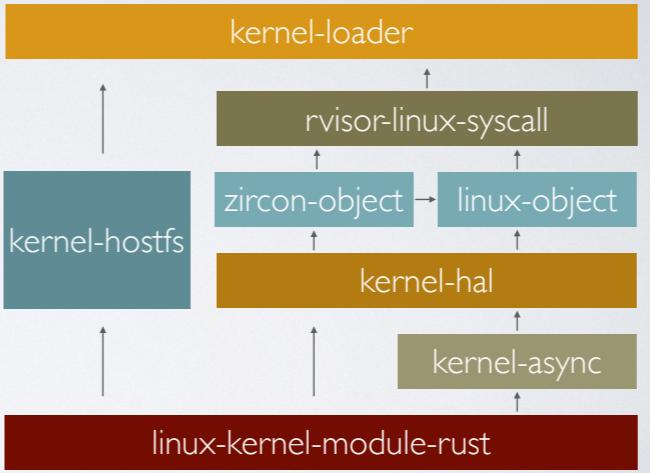
runrSC: rVisor 的命令行客户端，为 rVisor-shim 提供接口，也可以当作简单的客户端使用。

rVisor-shim 将我们整体架构与 containerd 连接，使得 rVisor 可以运行现有的容器。

我们将分别介绍这几个组件的功能。

RVISOR-KERNEL 内部架构设计

- kernel-loader 内核模块，实现容器架构，系统调用劫持
- linux-kernel-module 修改过的原 Github 项目，添加了安全的用户指针，和 logger 等部件。
- kernel-hostfs 用来在内核中实现 HostFs（本机文件系统）
- kernel-async 在内核中添加对 Rust async 语法的支持。
- zircon-object、linux-object 原 zCore 模块。
- rvisor-linux-syscall 所有劫持的 syscall 的实现。



然后我们 rVisor-kernel 的整体设计如下：

首先，kernel-loader实现了我们整体的内核模块，包括容器架构的实现，系统调用劫持的代码。

kernel-hostfs 作为我们在内核中实现的一个“本机文件系统”，也就是通过给本机进行系统调用的文件系统，我们将这个文件系统挂载到 zCore 里面就被 zCore 使用了，这一部分由 zzy 同学完成。

kernel-async 这一部分我们添加了对 Rust async 语法的支持。

zircon-object、linux-object、rvisor-linux-syscall 是我们从 zCore 内核移植过来的关键模块。

RVISOR-KERNEL

- 兼容性：
 - 添加 zCore 内核
 - 安全
 - 宿主机和容器的隔离
 - 容器与容器的隔离
 - 性能

我们将会从这样几个方面来讨论 RVisor 的具体设计。首先来看兼容性

实现 RVISOR-KERNEL 的兼容性

添加 ZCORE 内核

- rVisor 在内核部分使用 zCore 的内核代码。
- zCore 是清华大学操作系统课程实验的一个项目，由多届助教和学生共同开发、重构、和完善。
- zCore 是一个用 Rust 编写的 Zircon 微内核，Zircon 是 Google 推出的 Fuchsia OS 的微内核，zCore 的基本设计方法都是参照 Fuchsia 设计的。
- zCore 的实现代码非常精简（1w行），但却对 Linux 有比较好的支持，值得我们研究和学习。

为了保证 rVisor 的兼容性，我们的选择 zCore 作为 rVisor 的内核，zCore 是一个用 Rust 编写的 Zircon 微内核，借鉴自 Fuchsia OS，同时可以作为 Linux 内核使用。zCore 是清华他们 OS 课程实验的一个项目，是由多届助教和学生共同开发的，相比我们自己独立开发要更为完善。我们整体上使用 zCore 来保证 rVisor 系统的兼容性。

RVISOR-KERNEL 的兼容性 WHY ZCORE ?

- zCore 是一个设计理念非常先进的操作系统，既可以作为 Zircon 微内核，又可以作为 Linux 内核，既可以在裸金属上运行，也可以作为 LibOS 在用户态运行，它对硬件做了非常好的抽象，方便我们移植。
- zCore 相比用 Rust 编写的成熟的微内核操作系统 RedoxOS 更简单。
- zCore 的代码可复用性高。
- 实现了 zCore 之后，我们也可以对其他的更复杂的 OS 内核做移植。

然后，我们为什么选择 zCore 呢？

首先 zCore 是一个设计理念非常先进的系统，既可以作为 Zircon 微内核，又可以作为 Linux 内核，既可以在裸金属上运行，也可以作为 LibOS 在用户态运行。在这个方面做得比较好的 Rust 内核，也就只有 zCore 了。它对硬件做了非常好的抽象，方便我们移植。

然后相比一些其他用 Rust 编写的操作系统，比如说 RedoxOS，RedoxOS 也是一个用 Rust 编写的较为成熟的微内核操作系统。但 Redox 整体不仅仅是一个简单的内核，还包含其他的一系列操作系统组件和 GUI，结构非常复杂，不利于我们修改和使用。然后 zCore 本身要比 Redox 简单很多。更利于我们使用。

zCore 是清华设计的第二个操作系统，它的设计理念比较先进，代码的可复用性也很高，甚至支持编写单元测试，同时也便于我们实现容器的隔离。

RVISOR-KERNEL 的安全性

宿主机与容器的隔离

- RVisor 的所有系统调用都是通过 zCore 间接完成的，我们为 zCore 在内核中的运行提供基本的支持，然后所有系统调用都直接交由 zCore 运行。
- 为保证 rVisor 的安全性，我们会严格控制使用的系统调用，所有使用的系统调用如下：

```
8    使用的系统调用:
9        stat lseek read fdatasync fsync openat close mkdir rename (文件系统)
10       clock_gettime write fork clone exit wait4
11
12   内核函数/宏:
13       printk BUG
14       copy_from_user copy_to_user strncpy_from_user
15       set_fs get_fs
```

然后我们来看一下 rVisor 如何保证系统的安全性。

首先，我们先来谈一下宿主机与容器的隔离性。

RVisor 的所有系统调用都是通过 zCore 间接完成的，我们为 zCore 在内核中的运行提供基本的支持，然后所有系统调用都直接交由 zCore 运行。为了保证 rVisor 的安全性，我们会像 gVisor 一样严格地控制使用的系统调用，具体使用的系统调用都是一些基本的系统调用，包括一些对文件系统的操作以及其他的一些系统调用。

另外我们还使用了一些基本的内核函数，这些内核函数的安全性是可以保证的。

RVISOR-KERNEL 的安全性

宿主机与容器的隔离

- RVisor 使用 mmap 系统调用来为 zCore 分配内存。
- RVisor 创建一个 tmpfs 文件作为物理地址空间，然后我们修改了 zCore，zCore 会使用 mmap 来完成用户虚拟地址空间到tmpfs 文件物理地址空间的映射。
- 这样 zCore 就可以获得一个与原有 Linux 内核完全独立的物理内存空间，这样 RVisor 内部的内存系统可以完整地与 Linux 宿主机隔离。
- 同时，由于 tmpfs 是在内存中实现的文件系统，可以保证 zCore 直接读写物理内存时的效率。

然后就是 rVisor 在内存管理方面对 zCore 的隔离性。

rVisor 使用 mmap 系统调用为 zCore 分配内存，Mmap 系统调用可以将文件的内容同步映射到内存上，一般用来实现分配内存、共享内存、加载动态库之类的功能。

RVisor 创建一个临时文件作为物理地址空间，然后我们修改了 zCore，zCore 会使用 mmap 来完成用户虚拟地址空间到tmpfs 文件物理地址空间的映射。

这样 zCore 就可以获得一个与原有 Linux 内核完全独立的物理内存空间，这样 RVisor 内部的内存系统可以完整地与 Linux 宿主机隔离。

另一方面，由于 tmpfs 是在内存中实现的文件系统，可以保证 zCore 直接读写物理内存时的效率。

RVISOR-KERNEL 的安全性

容器与容器的隔离

- 内存管理的隔离：RVisor 会对 zCore 中的一些重要的静态变量进行了隔离，RVisor 为每个容器都提供一个临时文件作为物理地址和一个独立的空闲帧表。
- 文件系统的隔离：zCore 每个进程都有独立的文件系统节点指针，且 zCore 内部没有其他的静态的文件系统节点，故对文件系统的访问只能通过这个节点完成。

然后是容器和容器的隔离，为了实现容器和容器层面的隔离，我们回对 zCore 中的一部分可以被多个进程进行访问的静态变量进行处理，比如我们为了保证每个 zCore 容器拥有不同的物理地址，可以将每个容器都提供一个临时文件作为物理地址，同时提供一个独立的空闲帧表。这样不同的 zCore 容器就会拥有不同的物理内存，可以保证不同容器之间的隔离。

然后是文件系统的隔离，这里我们的方法就是利用 zCore 每个进程都有独立的文件系统指针，而且 zCore 没有全局的静态文件系统，这样可以有效保证进程不会访问到其他的容器的文件系统。

RVISOR-KERNEL 的性能

- rVisor-kernel 利用系统调用劫持，这个过程中只会出现两次内核态和用户态之间的转换。
- rVisor-kernel 整体在内核中运行，有效地避免了在用户空间的 gVisor 频繁地上下文切换问题。
- rVisor 的系统调用劫持与 Linux 原生的系统调用运行过程基本相同，有可以媲美原生应用的性能。
- 后面会有具体的 benchmark 来测试 rVisor-kernel 的性能。

然后，我们来讨论 rVisor 性能方面的保证。

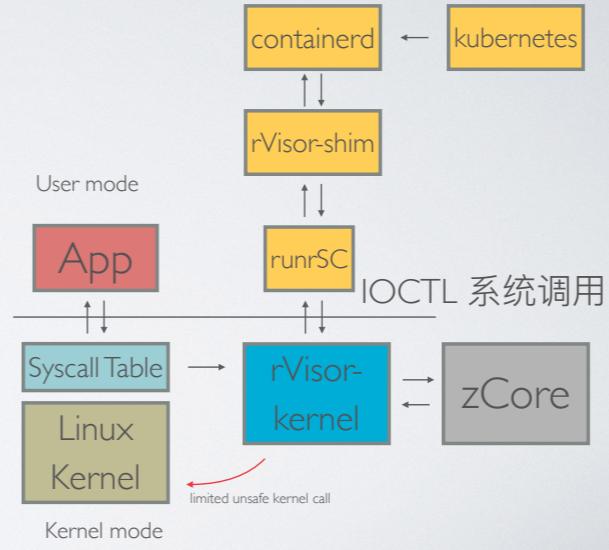
rVisor-kernel 利用系统调用劫持，使得 rVisor-kernel 可以整体在内核中运行，有效地避免了在用户空间的 gVisor 频繁地上下文切换问题。

rVisor 的系统调用劫持与 Linux 原生的系统调用运行过程基本相同，有可以媲美原生应用的性能。

后面我们会有简单的 benchmark 来验证 rVisor 的性能。

RVISOR 的外部组件 - 从外部控制 RVISOR

- runrsC 允许用户直接控制内核中的 rVisor
- RVisor 创建虚拟设备节点来实现与用户空间的交互，runrsC会从外部使用 ioctl 与rVisor 交互。
- 采用这种方式可以避免增加新的系统调用。



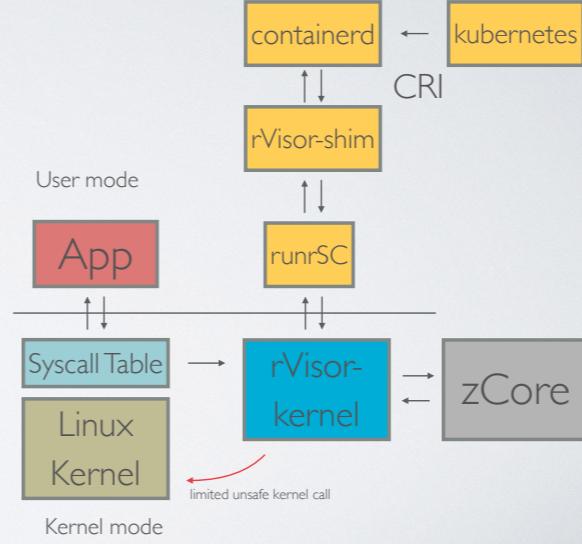
然后来谈一谈 rVisor 的外部组件

runrsC 允许用户直接控制内核中的 rVisor，这一部分工作由hhd 同学完成。

这里我们为了避免 rVisor 增加系统调用，RVisor 创建虚拟设备节点来实现与用户空间的交互，runrsC会从外部使用 ioctl 与rVisor 交互。runrsC 允许用户直接控制内核中的 rVisor 了。

RVISOR 的外部组件 - CONTAINERD-SHIM

- containerd-shim 为 runrsc 提供 containerd 接口，使得 rVisor 可以运行由 containerd 提供的各种容器。



然后还有我们的 containerd-shim。

containerd-shim 为 runrsc 提供 containerd 接口，由 containerd 来对 rVisor 的容器进行更细致的管理。

PART III - RVISOR 具体实现和演示

然后我们来看一下我们 rVisor 的具体实现。

实现的所有系统调用

- 目前的实现还比较简陋，只能完整支持 53 个系统调用。

```
1  read write openat close fstat fstatat lseek ioctl pread pwrite readv writev
2  sendfile fcntl fsync fdatasync truncate ftruncate getdents64 getcwd chdir
3  renameat mkdirat linkat unlinkat readlinkat faccessat copy_file_range sync
4  mmap mprotect munmap brk exit_group set_tid_address clock_gettime getpid gettid
5  uname getppid open stat lstat access dup2 fork vfork rename mkdir rmdir link
6  unlink readlink (53 个)
```

目前的 rVisor 的实现还比较简陋，只能支持 53 个系统调用，并不能运行所有 Linux 程序。

我们的实现还是相比较而言比较简单只是验证整体的设计思路。

RUNRSC 使用演示

- 整个程序是在我自己编写的一个 shell 和一个alpine程序中运行的。
- 内部的pid和外部的pid不一样。
- 可以运行像ls这种简单的busybox 命令。
- zCore 在 /dev 中只支持四个设备

```
ubuntu@vm800:/home/share/x-chital/runrsc$ sudo ./runrsc create /home/share/x-chital/runrsc/alpine/mknod: /dev/rvisor: File exists
[INFO] boot ready to accept
ubuntu@vm800:/home/share/x-chital/runrsc$ sudo ./runrsc exec /bin/sh
[INFO] son process = 26630
[INFO] Command Recieved.
/ # ls /dev
null
random
urandom
zero
/ # pid
1027
/ # [INFO] Command Recieved.
PID TTY      TIME CMD
26630 pts/0    00:00:00 sh
[INFO] Command Recieved.
[INFO] 26630 has been killed.
ubuntu@vm800:/home/share/x-chital/runrsc$ sudo ./runrsc ps
ubuntu@vm800:/home/share/x-chital/runrsc$ sudo ./runrsc shutdown
ubuntu@vm800:/home/share/x-chital/runrsc$ █
```

关于 runrsc 和 containerd 的使用演示，由 hhd 同学来给大家讲一下。

RUNRSC 使用演示

- cargo build —release 下编译的效果。

```
ubuntu@vm800:/home/dnailz/benchmark$ sudo ./runrsc benchmark `pwd`/Alpine/test
[INFO] son process = 20758
[INFO] Command Recieved
Writting disk: 2793108
Mkdir/Rmdir: 3401685
ubuntu@vm800:/home/dnailz/benchmark$ sudo ./runrsc benchmark `pwd`/Alpine/test
[INFO] son process = 20764
[INFO] Command Recieved
Writting disk: 2764177
Mkdir/Rmdir: 3590713
ubuntu@vm800:/home/dnailz/benchmark$ sudo ./runrsc benchmark `pwd`/Alpine/test
[INFO] son process = 20772
[INFO] Command Recieved
Writting disk: 2768281
Mkdir/Rmdir: 3362251
```

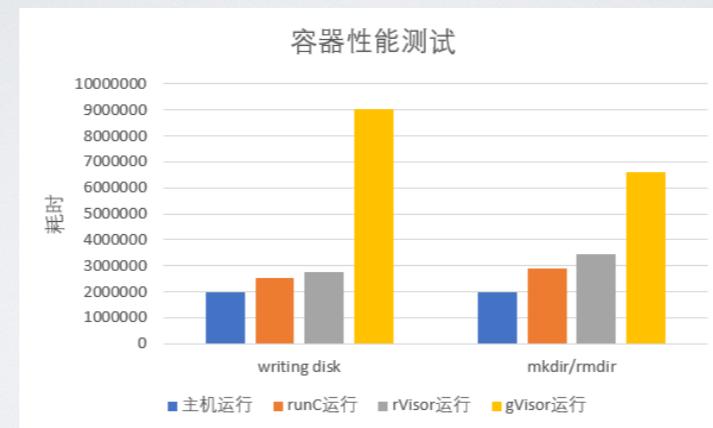
RUNRSC 使用演示

```
root@vm800:/home/dnailz/benchmark/bundle# docker run test0
Writting disk: 2555408
Mkdir/Rmdir: 3144911
root@vm800:/home/dnailz/benchmark/bundle# docker run test0
Writting disk: 2557819
Mkdir/Rmdir: 2843594
root@vm800:/home/dnailz/benchmark/bundle# docker run test0
Writting disk: 2508437
Mkdir/Rmdir: 2736177
```

```
root@vm800:/home/dnailz/benchmark/bundle# docker run --runtime=runsc test0
Writting disk: 10030000
Mkdir/Rmdir: 6870000
root@vm800:/home/dnailz/benchmark/bundle# docker run --runtime=runsc test0
Writting disk: 7890000
Mkdir/Rmdir: 6170000
root@vm800:/home/dnailz/benchmark/bundle# docker run --runtime=runsc test0
Writting disk: 8690000
Mkdir/Rmdir: 6670000
```

```
root@vm800:/home/dnailz/benchmark# ./runrsc2static
Writting disk: 1968831
Mkdir/Rmdir: 2001885
root@vm800:/home/dnailz/benchmark# ./runrsc2static
Writting disk: 1947289
Mkdir/Rmdir: 1978002
root@vm800:/home/dnailz/benchmark# ./runrsc2static
Writting disk: 1958874
Mkdir/Rmdir: 1901747
```

RUNRSC 性能测试



CONTAINERD-SHIM 使用演示

Dockerhub: hello world 的运行

```
[sudo] password for ubuntu:  
root@vm800:/home/yzf/go/shimproject# ./start.sh  
start ctr  
1:/run/containerd/io.containerd.runtime.v2.task/default/y87/rootfs  
2:/hello  
what?  
/run/containerd/io.containerd.runtime.v2.task/default/y87  
mknod: /dev/rvisor: File exists  
bind: Address already in use  
  
Hello from Docker!  
This message shows that your installation appears to be working correctly.  
To generate this message, Docker took the following steps:  
1. The Docker client contacted the Docker daemon.  
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
   (amd64)  
3. The Docker daemon created a new container from that image which runs the  
   executable that produces the output you are currently reading.  
4. The Docker daemon streamed that output to the Docker client, which sent it  
   to your terminal.  
  
To try something more ambitious, you can run an Ubuntu container with:  
$ docker run -it ubuntu bash  
  
Share images, automate workflows, and more with a free Docker ID:  
https://hub.docker.com/  
  
For more examples and ideas, visit:  
https://docs.docker.com/get-started/
```

The screenshot shows a Visual Studio Code interface connected via SSH to a 'New_Group_Server'. The left sidebar displays a file tree for a 'shimproject' directory, which contains files like quickstart.sh, container.go, service.go, shim.go, and start.sh. The right side features a terminal window titled 'startsh - home [SSH: New_Group_Server] - Visual Studio Code' showing the following command sequence:

```
1 | cp quickstart.sh /usr/bin/quickstart.sh
2 | export GO111MODULE=on
3 | export GOPATH=/home/yzf/gopath
4 | export PATH=$PATH:/snap/bin
5 | go build -o containerd-shim-rvisor-v1 main.go
6 | sudo mv containerd-shim-rvisor-v1 /usr/bin
7 | cd ..
8 | echo start ctr
9 | #ctr --debug run --rm --runtime io.containerd.rvisor.v1 docker.io/library/hello-world:latest y117
10 | ctr --debug run --rm --runtime io.containerd.rvisor.v1 docker.io/library/alpine:latest y120
11 | #ctr run --rm --runtime io.containerd.rvisor.v1 docker.io/library/newi:test y121
```

The terminal also shows a root prompt with a list of system directories (bin, lib, media, etc.) and ends with a '#'. The status bar at the bottom indicates the terminal is running 'bash'.

Containerd-Shim Example: Alpine

展望

- zCore 还不算特别完善，随着 zCore 的完善，我们可以支持更多的系统调用。
- rVisor 还需要进一步在实际的程序中验证其可用性。

好的，下面由我做一下简单的总结和展望。

由于 zCore 还不够完善，我们现在 rVisor 还仅仅处于一个功能验证的状态，不过随着 zCore 的完善我们可以支持更多的系统调用。另一方面，之前的benchmark 还比较简单，下一步还需要更进一步验证 rVisor 的可用性。

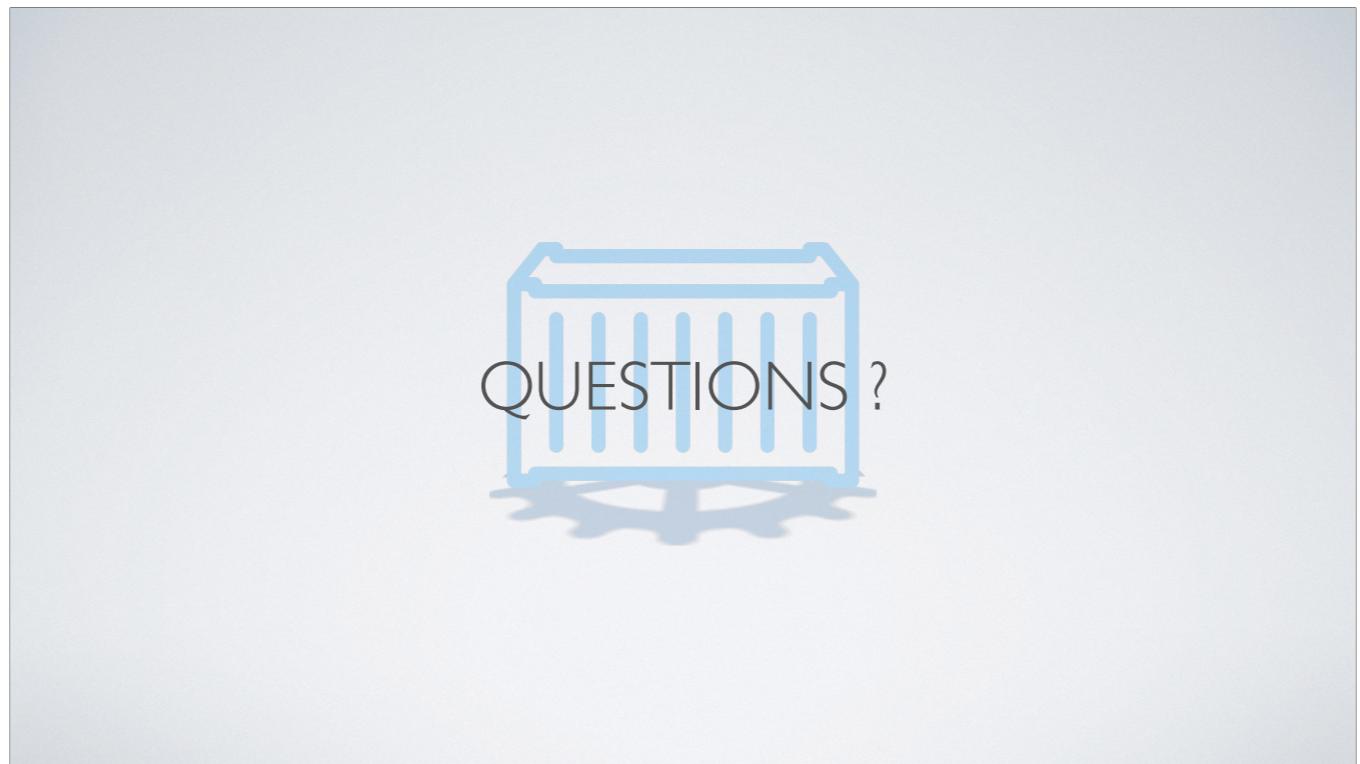
ABOUT CHITAL

- 丁垣天 dnailzb@outlook.com
- 叶之帆 rabbitforyou@foxmail.com
- 何灏迪 hardyhe2019@outlook.com
- 郑在一 donpanica@outlook.com



好的，这是我们小组的成员表。

benchmark



谢谢大家，大家有什么问题吗？