CS 423
Operating System Design: OS Support for Containers

Tianyin Xu

Thanks for Adam Bates and Julie Evans
• Dominated by Infrastructure-as-a-Service clouds (and storage services)
• Big winner was Amazon EC2
• Hypervisors that virtualized the hardware-software interface
• Customers were responsible for provisioning the software stack from the kernel up
Hypervisors

- Guest OS (separate kernel & user space)
- Guest OS (separate kernel & user space)
- Guest OS (separate kernel & user space)
- Hypervisor/VMM
- Kernel of the Host
- Hardware
Hypervisors

• Strong isolation between different customer’s virtual machines
• VMM is ‘small’ compared to the kernel... less LoC means less bugs means (∼)more security.
• ‘Practical’ attacks on IaaS clouds relied on side channels to detect co-location between attacker and victim VM

• E.g., we could correlate the performance of a shared resource
  • network RTT’s, cache performance

• After co-resident, make inferences about victim’s activities
• Strong isolation between different customer’s virtual machines
• VMM is ‘small’ compared to the kernel… less LoC means less bugs means (≈)more security.
• High degree of flexibility… but did most customers really need it?
Cloud Computing (Gen 2)

- **PaaS**: Platform as a Service
- **SaaS**: Software as a Service

<table>
<thead>
<tr>
<th>Hosted applications/apps</th>
<th>Development tools, database management, business analytics</th>
<th>Operating systems</th>
<th>Servers and storage</th>
<th>Networking firewalls/security</th>
<th>Data center physical plant/building</th>
</tr>
</thead>
</table>
New Gen of Cloud Computing

- **FaaS: Function as a Service**
• Microservices

A monolithic application puts all its functionality into a single process...

... and scales by replicating the monolith on multiple servers

A microservices architecture puts each element of functionality into a separate service...

... and scales by distributing these services across servers, replicating as needed.
Enter Containers

• Rather than virtualize both user space and kernel space… why not just ‘virtualize’ user space?
• Meets the needs of most customers, who don’t require significant customization of the OS.
• Sometimes called ‘OS virtualization,’ which is highly misleading given our existing taxonomy of virtualization techniques
• Running natively on host, containers enjoy bare metal performance without reliance on advanced virtualization support from hardware.
Enter Containers

Host OS

Container
separate
user
space

Container
separate
user
space

Container
separate
user
space

Kernel of the Host

Hardware
Enter Containers

Host OS

Container
separate
user
space

Container
separate
user
space

 Kernel of the Host

Hardware

Acts like a process from the outside!

Looks like a VM from the inside!
Containers are processes

Containers vs VMs

Containers occupy less RAM

This is because they share a single Linux kernel.

I can easily run thousands of small containers!

Containers start faster

because they're processes and process start fast

Done!

container

Um my operating system is still booting

VM

Containers are more complicated to secure

I'm totally isolated from other VMs on this computer!

VM

Container

Um it really depends how you configured me...

VM

Container

It's harder to figure out what you can do in a container

Just pretend I'm a computer! It's easy!

VM

I act like a VM kinda but there are exceptions...

VM

Container
Container is an old idea

- You didn’t hear of it cause it was not called “containers.”
  - Linux containers
  - BSD Jails
  - Solaris Zones
Docker’s Big Idea

- Build, Ship, and Run App, Anywhere
- Debug your app, not your environment -- Securely build and share any application, anywhere
Docker's Big Idea

Julia Evans @bork

The big idea: include EVERY dependency

Containers package EVERY dependency together
to make sure this program will run on your laptop, I'm going to send you every single file on my computer
exaggeration but it's the basic idea

A container image is a tarball of a filesystem
Here's what's in a typical Rails app's container:

- Your app's code
- Libc + other system libraries
- Rails + other Ruby gems
- Ubuntu 18.04 base OS
- Ruby interpreter

How images are built
0. Start with a base OS
1. Install program + dependencies
2. Configure it however you want
3. Make a tarball of the WHOLE FILESYSTEM
(this is what 'docker build' does)

Running an image
1. Download the tarball
2. Unpack it into a directory
3. Run a program and pretend that directory is its whole filesystem
(this is what 'docker run' does)

Images let you "install" programs really easily

Wow, I can get a Postgres test database running in 45 seconds!
OS Support for Containers

- Linux Containers (LXC):
  - `chroot`
  - `namespace`
    - PID, Network, User, IPC, uts, mount
  - `cgroups` for HW isolation
  - Security profiles and policies
    - Apparmor, SELinux, Seccomp
containers = chroot on steroids

- **chroot** changes the apparent root directory for a given process and all of its children
- An old idea! POSIX call dating back to 1979
- Not intended to defend against privileged attackers… they still have root access and can do all sorts of things to break out (like chroot’ing again)
- Hiding the true root FS isolates a lot; in *nix, file abstraction used extensively
- Does not completely hide processes, network, etc., though!
Chroot

Julia Evans @b0rk

a container image is a tarball of a filesystem (or several tarballs: 1 per layer)
if someone sends me a tarball of their filesystem how do I use that though?

very basic way to "run" a Redis container tarball of filesystem with Redis installed
$ mkdir redis; cd /redis
$ tar -xzf redis.tar
$ chroot $PWD /usr/bin/redis
# done! redis is running!

chroot: trick a program into thinking it has a different root directory

$ ls /path/to/container_filesystem
bin/ etc/ usr/ var/
$ sudo chroot /path/to/container_filesystem /bin/bash
(inside chroot now)
$ ls /
bin/ etc/ usr/ var/ that's our new fake root directory!
we tricked ls!

problems with just using chroot
→ no CPU/memory limits
→ other running processes are still visible
→ can't use the same network port as another process
→ LOTS of security issues

Docke uses pivot-root + extra isolation features to run containers
pivot-root is like chroot but harder to escape from
Namespaces

• The key feature enabling containerization!
• Partition practically all OS functionalities so that different process domains see different things
  • **Mount** (*mnt*): Controls mount points
  • **Process ID** (*pid*): Exposes a new set of process IDs distinct from other namespaces (i.e., the hosts)
  • **Network** (*net*): Dedicated network stack per container; each interface present in exactly one namespace at a time.
  • ....
Namespaces

• The key feature enabling containerization!

• Partition practically all OS functionalities so that different process domains see different things

• **Interprocess Comm. (IPC):** Isolate processes from various methods of POSIX IPC.
  
  • e.g., no shared memory between containers!

• **UTS:** Allows the host to present different host/domain names to different containers.

• There’s also a **User ID (user)** and **cgroup** namespace
• Like others, can provide a unique UID space to the container.

• More nuanced though — we can map UID 0 inside the container to UID 1000 outside; allows processes inside of container to think they’re root.

• Enables containers to perform administration actions, e.g., adding more users, while remaining confined to their namespace.
Namespace

inside a container, things look different

- I only see 4 processes in `ps aux`, that's weird...

Commands that will look different:

- `ps aux` (less processes!)
- `mount & df`
- `netstat -tulpn` (different open ports!)
- `hostname`... and LOTS more

Why those commands look different:

I'm in a different PID namespace so `ps aux` shows different processes!

every process has 7 kinds of namespaces

- network
- mount
- user
- PID
- cgroup
- UTS
- IPC

there's a default ("host") namespace

"outside a container" just means "using the default namespaces"

processes can have any combination of namespaces

I'm using the host network namespace but my own mount namespace!

❤️ this? more at wizardzines.com
cgroups

- Limit, track, and isolate utilization of hardware resources including CPU, memory, and disk.

- Important for ensuring QoS between customers! Protects against bad neighbors

- Features:
  - Resource limitation
  - Prioritization
  - Accounting (for billing customers!)
  - Control, e.g., freezing groups

- The cgroup namespace prevents containers from viewing or modifying their own group assignment
**cgroups**

**Julia Evans @boork**

**Processes can use a lot of memory**

- I want 10 GB of memory
- Me too!
- Guys I only have 16 GB total

**A cgroup is a group of processes**

- Usually you'll assign the same cgroup to every process in a container.

**Cgroups have memory/CPU limits**

- You three get 500 MB of RAM to share, okay?

**Use too much memory: get OOM killed**

- I want 1 GB of memory
- NOPE your limit was 500 MB you die now

**Use too much CPU: get slowed down**

- I want to use ALL THE CPU!
- You hit your quota this millisecond, you'll have to wait

**Cgroups track memory & CPU usage**

- That cgroup is using 412.3 MB of memory right now!
- See `/sys/fs/cgroup`!
“Containers do not contain.” - Dan Walsh (SELinux contributor)

- In a nutshell, it’s **real hard** to prove that every feature of the operating system is namespaced.
- Root access to any of these enables pwning the host
- Solution? Just don’t forget about MAC; at this point SELinux pretty good support for namespace labeling.
- SELinux and Namespaces actually synergize nicely; **much** easier to express a correct isolation policy over a coarse-grained namespace than, say, individual processes
# Capabilities

The root user can do *anything*:
- edit any file
- change network config
- spy on any program's memory

Capabilities let you grant specific permissions.
- CAP_SYS_ADMIN: basically root access. Try to use a more specific capability!
- CAP_NET_ADMIN: for changing network settings
- CAP_SYS_PTRACE: strace needs this
- CAP_NET_RAW: ping needs this to send raw ICMP packets

Sometimes containers need privileged access:
- I need to update the route for 10.1.93.4
- That container shouldn't have root access though!

Here's CAP_NET_ADMIN so you can manage the network:
- Yay!

Run this in a container to print its capabilities:
- `capsh --print`

Shows which capabilities ping is allowed to use:
- `getcap /usr/bin/ping`
Seccomp-bpf

**all programs use system calls**

- Read 2000 bytes from this file.
- Here you go!

**some programs have security vulnerabilities**

- I know ffmpeg codecs can be exploited but I really need to process these untrusted videos...

**rarely used syscalls can help an attacker**

- Process VM read
- Reboot
- Request key

**seccomp-BPF: make Linux run a tiny program before every system call**

- Reboot the computer!

**Docker blocks dozens of syscalls by default**

- Most programs don't need those system calls so I told Linux to block them for you

**2 ways to block scary system calls**

1. Limit a container's capabilities
2. Use a seccomp-BPF whitelist

Usually people do both!
Linux Security Modules

Figure 1: LSM Hook Architecture

- User Level process
- open system call
- Look up inode
- error checks
- DAC checks
- LSM hook
- Complete request
- Access
- inode

User space

Kernel space

LSM Module Policy Engine

Examine context
Does request pass policy?
Grant or deny

“OK with you?”
Yes or No
DIY container

Containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!
(download this script at bit.ly/containers-arent-magic)

```
wget bit.ly/fish-container -O fish.tar
mkdir container-root; cd container-root
tar -xf ./fish.tar
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)"
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"
cgset -r cpu.shares=512 "$cgroup_id"
cgset -r memory.limit_in_bytes=1000000000 \\ "$cgroup_id"
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \
  unshare --mount-proc \
  chroot "$PWD" \
  /bin/sh -c "
    /bin/mount -t proc proc /proc &&
    hostname container-fun-times &&
    /usr/bin/fish"
```

# 1. download the image
# 2. unpack image into a directory
# 3. generate random cgroup name
# 4. make a cgroup &
#    set CPU/memory limits
# 5. use the cgroup
# 6. make + use some namespaces
# 7. change root directory
# 8. use the right /proc
# 9. change the hostname
# 10. finally, start fish!
container kernel features

Containers are implemented using these Linux kernel features:
You can use any of these on their own. When we use them all we call it a "container".

- **pivot_root**
  Set a process’s root directory to a directory with the contents of the container image.

- **cgroups**
  Limit memory/CPU usage for a group of processes.

- **namespaces**
  Allow processes to have their own:
  - network
  - hostname
  - PIDs
  - mounts
  - users
  - more

- **seccomp-bpf**
  Security: prevent dangerous system calls.

- **capabilities**
  Security: avoid giving root access.

- **overlay filesystems**
  Optimization to reduce disk space used by containers which are using the same image.
• Container support has existing in Linux for many years
• Foundations of containerization has been around for decades!
• Automating LXC for portability (i.e., Docker) has revolutionized cloud computing
• Lasting legacy of containers may be enabling the Function-as-a-Service revolution… cloud customers can now pay by the method invocation without any idle costs.