Zero-consistency root emulation for unprivileged container image build

Reid Priedhorsky, Michael Jennings, Megan Phinney
High Performance Computing Division
Los Alamos National Laboratory
Los Alamos, NM, USA
{reidpr,mej,mphinney}@lanl.gov

***Abstract*—Do Linux distribution package managers need the privileged operations they request to actually happen? Apparently not, at least when building container images for HPC applications. We use this observation to implement a root emulation mode using a Linux seccomp filter that intercepts some privileged system calls, does nothing, and returns success to the calling program. This approach provides no consistency whatsoever but appears sufficient to build a wide selection of Dockerfiles, including one that Docker itself cannot build, simplifying fully-unprivileged workflows needed for HPC application containers.**

I. Introduction

Scientific software for high-performance computing (HPC) is increasingly deployed using Linux containers, which is a technology to package an application along with all its dependencies as a single unit called an *image*. A critical requirement for many HPC centers, including Los Alamos, is that user workflows must be fully unprivileged [1]; i.e., HPC users cannot be given elevated access of any kind to production resources. Within an unprivileged container, processes can have an effective user ID (EUID) of 0 (i.e., root) in a container and/or arbitrary capabilities, as well as access to some normally-privileged system calls, but this greater privilege is an illusion. Only unprivileged operations are actually available.

HPC container performance and reliability are typically best served by building images on the same supercomputer(s) targeted for deployment, due to their tightly specified architectures. This demands that building images, not just running them, must be fully unprivileged. However, these builds almost always use traditional Linux distribution package managers such as `dpkg(8)`[1] or `rpm(8)`, which assume they are running privileged, an assumption that has held for many years. While future package managers may relax this assumption, a build solution is needed for current distributions, which are in use now and may persist well beyond end-of-support when containerized.

This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy (DOE) Office of Science and the National Nuclear Security Administration (NNSA); the Advanced Simulation and Computing Program (ASC); and the LANL Institutional Computing Program, which is supported by the U.S. DOE's NNSA under contract 89233218CNA000001. LA-UR 24-24056.

[1]Notation `foo(n)` indicates the thing named `foo` in man pages section `n`. §1 is user shell commands, §2 is system calls, §8 is administrator commands, and there are others [2].

One way to bridge this gap is *root emulation*, which replaces key privileged operations (e.g. `chown(2)` to change file ownership) with similar-enough unprivileged ones, thus fooling package managers into believing they are privileged.

Existing approaches maximize the consistency of the root-emulated environment. For example, `chown(2)` could be intercepted, and instead of making the actual system call, the request stored. Then, results of later `stat(2)` would be adjusted to be consistent, i.e., the process sees the same fake ownership that it set earlier. These tools must be either installed or bind-mounted into the container, and they add complexity and overhead while reducing compatibility.

Our insight is that *consistency is not actually required when building HPC application images*. We can tell processes simple lies instead of complex ones.

This paper describes a lightweight, non-consistent root emulation mode based on seccomp filters, which was recently introduced in Charliecloud, LANL's lightweight, fully unprivileged container implementation for HPC applications [3].[2] We install a seccomp kernel filter that intercepts privileged system calls and simply returns success, without invoking the syscall or any user-space emulation of it. This remarkably unsophisticated root emulation appears workable for all images we tried; it is compatible with all distributions and libc's as well as statically linked binaries; and it has no dependencies beyond a C compiler and the Linux kernel, not even libseccomp [4].

II. Fully unprivileged (Type III) image build without root emulation?

We previously proposed a tripartite classification of container implementations, based on the Linux namespaces used [5], [6] and level of privilege needed to set up the container [1]:

- **Type I** containers are the bare minimum, using the mount namespace but not the user namespace.[3] They require privileged setup (root or `CAP_SYS_ADMIN`).
- **Type II** containers use mount and privileged user namespaces. They also require privileged setup (root or `CAP_SETUID` and `CAP_SETGID`). Many implementations call this type *rootless* because the main container

[2]All authors are members of the Charliecloud team, and our argument should be considered in that context.

[3]The other namespaces do not affect our classification.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00023

```
$ cat Dockerfile
FROM alpine:3.19
RUN apk add sl
$ ch-image build -t win --force=none .
  1. FROM alpine:3.19
[...]
  2. RUN.N apk add sl
copying image from cache ...
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/ma
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/co
(1/3) Installing ncurses-terminfo-base (6.4_p2023112
(2/3) Installing libncursesw (6.4_p20231125-r0)
(3/3) Installing sl (5.02-r1)
Executing busybox-1.36.1-r19.trigger
OK: 8 MiB in 18 packages
grown in 2 instructions: win
```

```
$ cat Dockerfile
FROM centos:7
RUN yum install -y openssh
$ ch-image build -t win --force=none .
  1* FROM centos:7
  2. RUN.N yum install -y openssh
[...]
  Installing : openssh-7.4p1-23.el7_9.x86_64     3/3
Error unpacking rpm package openssh-7.4p1-23.el7_9.x
error: unpacking [...] failed [...]: cpio: chown
[...]
something went wrong, rolling back ...
[...]
error: build failed: RUN command exited with 1
```

Fig. 1: Example Dockerfiles built with a Type III (fully unprivileged) implementation and no root emulation. Terminal output is right-truncated. (a) succeeded because no privileged system calls were used, while (b) failed because `rpm(8)` tried to change a file's owner with privileged `chown(2)`.

processes are unprivileged, but we believe this is a misnomer because privileged helper programs (typically `newuidmap(1)` and `newgidmap(1)`) are needed to set up the container namespaces.

- **Type III** containers use mount and unprivileged user namespaces; setup is unprivileged. *Only Type III containers are fully unprivileged throughout the container lifetime*. Conversely, the benefit of Type II over Type III is greater flexibility of users and groups within the container.

It would be convenient for HPC if images could be built in a Type III container naïvely, with no root emulation or other special measures. This does sometimes work. Figure 1a shows an example Dockerfile built with no root emulation; `apk(8)` can install `sl(1)` [sic] with no privileged system calls. On the other hand, in Figure 1b, `rpm(8)` failed to change a file's ownership with `chown(2)`, a privileged operation disallowed in an unprivileged container despite being container root.

This is why we need root emulation. What if `chown(2)` was not really `chown(2)` but rather an unprivileged substitute?

```
$ fakeroot ./fakeroot.sh
+ touch _file
+ chown nobody _file
+ mknod _dev c 1 3
+ ls -lh _*
crw-r----- 1 root   root 1, 3 Feb 10 18:09 _dev
-rw-r----- 1 nobody root    0 Feb 10 18:09 _file
$ ls -lh _*
-rw-r----- 1 reidpr reidpr  0 Feb 10 18:09 _dev
-rw-r----- 1 reidpr reidpr  0 Feb 10 18:09 _file
```

Fig. 2: Script that changes file ownership and then creates a device file, both of which are privileged operations, under `fakeroot(1)`. These "succeed" because `fakeroot(1)` intercepts the system calls and substitutes unprivileged userspace emulations. Under `fakeroot(1)`, `ls(1)` displays as expected (device and nobody-owned file, lines 5–7) because the `stat(2)` result is altered to match the prior emulations, but the subsequent unwrapped `ls(1)` exposes the lies (lines 8–10).

III. Related work

Charliecloud is not the first to implement root emulation, whether complex or simple. However, to our knowledge, in 2020 Charliecloud was the first to provide complex root emulation emulation in the context of container image builders [1], and in 2023 it was the first to provide simple emulation, as described in this paper. Here we detail existing root emulation work, with a focus on image build.

A. *fakeroot(1)*

`fakeroot(1)` is a program to run a command in a root-emulated environment. It is not a perfect simulation but rather just enough to work for its intended purpose, which is building distribution packages, allowing "users to create archives (tar, ar, `.deb` etc.) with files in them with root permissions/ownership" [7]. There are at least three `fakeroot(1)` implementations [1, Table 1] that hook processes in two different ways. `LD_PRELOAD` is a userspace mechanism that lets `fakeroot(1)` intercept shared library function calls (not syscalls); this is architecture-independent but cannot wrap statically linked executables. `ptrace(2)` is a kernel mechanism that can intercept system calls (in addition to many other things). It is architecture-dependent but can wrap statically linked executables. We have encountered packages that one implementation can install but others cannot.

All `fakeroot(1)`s maintain state in order to provide a consistent emulated environment, e.g. so `stat(2)` is consistent with prior `chown(2)`, with a daemon and/or disk files. See Figure 2 for an example of `fakeroot(1)` use.

Charliecloud was the first to implement `fakeroot(1)` injection into container image builds. It does this by installing `fakeroot(1)` into the image from package repositories of the containerized distribution, which requires detailed configuration for each supported distribution.

Figure 3 shows abbreviated output of Charliecloud in this emulation mode: `ch-image(1)` detects a likely-privileged command (`apt-get(8)`), installs the `fakeroot` package, and

```
$ ch-image build -t win --force=fakeroot .
  1* FROM debian:bookworm
  2. RUN.F apt-get install -y openssh-server
--force=fakeroot: will use: debderiv: Debian 9+, Ub
--force=fakeroot: init step 1: checking: $ apt-conf
--force=fakeroot: init step 1: $ echo 'APT::Sandbox
--force=fakeroot: init step 2: checking: $ command
--force=fakeroot: init step 2: $ apt-get update &&
[...]
Setting up libfakeroot:amd64 (1.31-1.2) ...
Setting up fakeroot (1.31-1.2) ...
[...]
--force: RUN: new command: ['fakeroot', '/bin/sh',
Reading package lists... Done
[...]
--force=fakeroot: modified 1 RUN instructions
grown in 2 instructions: win
```

Fig. 3: Charliecloud's `ch-image(1)` installing OpenSSH with auto-injected `fakeroot(1)`. Note the complex diagnostics, including installing a package (`fakeroot`) and its dependencies that the user did not request.

then prepends "`fakeroot`" to `RUN`'s command. See our prior paper [1] for further details on this mode.

Singularity [8], like Charliecloud, is a container implementation targeting HPC applications that supports image build. The project forked in 2021 [9], [10]. One of the forks, Apptainer, also supports root emulation via `fakeroot(1)`, based on Charliecloud's implementation but with a key difference: the host's `fakeroot(1)` is bind-mounted into the container [11]. This trades the need to install it in the image for tighter dependence between the host and container libc, but it does not address the other drawbacks of `fakeroot(1)`.

B. PRoot

Another stand-alone root emulator is PRoot, which uses `ptrace(2)` to intercept system calls [12], avoiding libc compatibility issues and allowing the tool to wrap static executables [13]. PRoot can in fact use seccomp filters, but for a different purpose than Charliecloud: it is a performance optimization that lets PRoot avoid being notified about system calls it doesn't need to intercept. However, the fundamental constraints of a complex, state-maintaining tool remain.

SingularityCE [10], the other fork of Singularity, bind-mounts the host's `proot(1)` to provide root emulation for image build, building on the lessons of `fakeroot(1)` by Charliecloud and Apptainer [13] for an arguably better implementation of complex root emulation.

C. *fakechroot(1)*

Like some `fakeroot(1)` implementations, `fakechroot(1)` uses `LD_PRELOAD` to intercept libc function calls, the main goal being to provide an unprivileged `chroot(2)` [14]. It also provides a simple root emulation by substituting `/bin/true` for a configurable set of executables. This is sufficient to e.g. bootstrap a Debian distribution, but this executables-only emulation surface is not enough for general image building.

```
1  void install_filter(void) {
2   struct sock_filter filter[] = {
3    // note: critical architecture code omitted
4    BPF_STMT(BPF_LD|BPF_W|BPF_ABS,
5    offsetof(struct seccomp_data, nr)),
6    BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, __NR_open, 2,0),
7    BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, __NR_openat, 1,0),
8    BPF_STMT(BPF_RET|BPF_K, SECCOMP_RET_ALLOW),
9    BPF_STMT(BPF_RET|BPF_K, SECCOMP_RET_KILL_PROCESS)
10  };
11
12  struct sock_fprog prog = {
13   .len = sizeof(filter) / sizeof(filter[0]),
14   .filter = filter,
15  };
16
17  seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog);
18 }
```

Fig. 4: A skeleton seccomp filter to deny `open(2)` and `openat(2)` to a process and its children. Adapted from [15].

IV. Seccomp (filter mode)

In this paper we are concerned with "filter mode" seccomp,[4] introduced in Linux 3.5 in 2012 [15]. This lets a process install a filter to manipulate the system calls of itself and its children. This filter is a Berkeley Packet Filter (BPF)[5] program run by the kernel, and once installed it cannot be removed, i.e., it binds program children whether they like it or not. Notably, BPF does not have loops, so it can be verified for completion by the kernel.

The BPF filter is run by the kernel upon each system call. It has four inputs: (1) the system call number (not name!), which varies by architecture; (2) the syscall's arguments, (3) the current architecture, which can vary even within a process, and (4) the instruction pointer. An important limitation is that BPF filters cannot dereference pointers. After its computation, the filter returns the disposition of the system call, which falls into three classes:

1) **Do not execute the syscall**, and one of (a) kill the thread (Linux 3.5), (b) kill the process (4.14), (c) send SIGSYS to the thread (3.5), or (d) return a specified `errno` (3.5).
2) **Execute the syscall**, and (e) log it first (4.14) or simply (f) execute it normally (3.5).
3) **Delegate the decision to a userspace process**, either with (g) `ptrace(2)` (3.5) or (h) a file descriptor (5.0), which then chooses disposition a–f.

Figure 4 shows a BPF filter that prevents a process from calling `open(2)` or `openat(2)`, built directly as a C struct using kernel macros, rather than e.g. libseccomp. The program first loads the current syscall number info the accumulator (line 4). It then compares that to the syscall numbers for `open(2)` (line 5) and `openat(2)` (line 6); if either match, it jumps to line 8, which directs the kernel to kill the calling process (instead of executing the syscall). Otherwise, the

[4]Seccomp is short for *secure computing*, though its scope has expanded considerably since naming.

[5]As the name implies, BPF was originally designed to manipulate network packets but likewise has been expanded in scope.

program falls through to line 7 and the syscall executes normally. Lines 11–16 install the filter program.

Several container implementations use seccomp filters. Docker and Podman/Buildah have a filter specification feature, apparently intended as a simple allow/denylist for syscalls [16]. distrobuilder, part of the LXC/LXD package, has a filter configuration language that could likely implement root emulation like Charliecloud's, but because users "must be root in order to run the distrobuilder tool" [17], that potential capability is moot. Firejail is a tool to sandbox processes using container-like technologies such as namespaces; it does filter system calls using seccomp but not for root emulation [18]. NsJail is a "light-weight process isolation tool" that uses namespaces [19].[6] It provides a seccomp configuration language that appears flexible enough to implement root emulation, but we are unaware of anyone having done so.

Finally and notably, Enroot is a small container runtime that *does* provide a lightweight root-emulation seccomp filter similar to Charliecloud's: "[w]e use a seccomp filter to trap all setuid-related syscalls, to make them succeed" [20]. However, the filter is less complete than Charliecloud's, and Enroot does not provide a build capability, which is where the main root emulation challenge lies.

V. Charliecloud's seccomp filter

A. Filter program structure

Charliecloud's zero-consistency root emulation installs a seccomp filter to intercept certain privileged system calls and fake their success. In pseudocode, our filter is:

```
1 if (privileged system call):
2    do nothing
3    return success
```

That is, from the point of view of a filtered process, these system calls always succeed, but if the process does anything to verify the actions requested, it will see that nothing happened.

The 29 privileged syscalls we filter fall into four classes:

1) **File ownership** (7 syscalls): `chown(2)`, `fchownat(2)`, etc.
2) **User, group, or capability manipulation** (19): `setresuid(2)`, `capset(2)`, etc.
3) **File creation** (2): `mknod(2)` and `mknodat(2)` can be privileged or not. We examine the file type argument before faking success (device file) or allowing the syscall (other file types).
4) **Self-test** (1): `kexec_load(2)` reboots into a new kernel and is unlikely to ever be needed by HPC applications, so we use it to validate the filter after installation.

Charliecloud's source code has a table listing the numbers for each syscall on each of the six supported architectures.[7] We translate this into a BPF program and install it with two C functions totalling about 150 lines of code, including comments.

[6]Though it resides in Google's GitHub organization, NsJail's readme states that it "is NOT an official Google product" [emphasis in original].

[7]Some syscalls are not implemented on all architectures; for example, arm64 lacks `chown(2)`, relying on user-space code to translate its calls to `fchownat(2)` instead.

```
1  $ cat Dockerfile
2  FROM alpine:3.19
3  RUN mknod _fifo p
4  RUN mknod _chardev c 1 3
5  RUN ls _*
6  $ ch-image build -t win .
7    1. FROM alpine:3.19
8    2. RUN.S mknod _fifo p
9    3. RUN.S mknod _chardev c 1 3
10   4. RUN.S ls _*
11 _fifo
12 --force=seccomp: modified 0 RUN instructions
13 grown in 4 instructions: win
```

Fig. 5: Successful seccomp root-emulation build of a Dockerfile that creates a FIFO (named pipe) and character device, one that Docker itself is unable to build. Importantly, note that while the build succeeded, the device file `_chardev` was not actually created.

B. Image build example

Figure 5 shows a successful Charliecloud build using the zero-consistency root emulation mode. We highlight this image because it creates a device file — an operation that is absolutely inappropriate for unprivileged users because it allows direct hardware access. Docker cannot build this image, failing eagerly at the device file creation attempt. Charliecloud is instead lazy: any later attempt to access the non-existent device will be prevented by the kernel. This fits our use case because all device files HPC users need should already exist, i.e., we assume the device file here is being created by some incidental dependency, rather than something the application actually cares about. Therefore, wait until an actual attempt at violating the security boundary occurs before failing.

Specifically, "`mknod _chardev c 1 3`" succeeded (line 5). We see on lines 6–7, however, that no file of any type called `_chardev` was created. In our experience, neither package managers nor HPC applications are affected by inconsistencies like these.

To understand the filter in more detail, we can examine a disassembly of the BPF filter, provided by the Ruby gem `seccomp-tools` and shown in part in Figure 6. We can use this to trace program flow of the two `mknod(2)` calls.

The program first loads the architecture code into the accumulator (instruction 000), then compares it to arm64 (001). For this example, we are on x86-64, so the program repeatedly tests architectures until arriving at instruction 128, where it finally matches and we fall through to loading the system call number (129).[8] We then enter a jump table for system calls; most of these analyze the call no further and jump directly to instruction 149, which returns an "error" of zero instead of executing the syscall, i.e., fake do-nothing success. If the syscall is not one we intercept, we fall through the whole table and finish at instruction 148, which allows the syscall to execute normally.

[8]This program does have optimizations available. For example, we could avoid the repeated architecture loads at the cost of slightly more complex arithmetic during translation.

```
000  A = arch
001  if (A != ARCH_AARCH64) goto 19
[...]
019  A = arch
020  if (A != 0x40000028) goto 52
[... goto 85 ... 106 ... 127 ...]
127  A = arch
128  if (A != ARCH_X86_64) goto 148
129  A = sys_number
130  if (A == capset) goto 149
131  if (A == chown) goto 149
[... other system calls ...]
145  if (A == mknod) goto 150
146  if (A == mknodat) goto 0152
147  goto 0148
148  return ALLOW
149  return ERRNO(0)
150  A = args[1]
151  goto 0153
152  A = args[2]
153  A &= 0xf000
154  if (A == 8192) goto 0157
155  if (A == 24576) goto 0157
156  return ALLOW
157  return ERRNO(0)
```

Fig. 6: Excerpt of Charliecloud's seccomp program relevant to Figure 5, disassembled using the `seccomp-tools` Ruby gem.

In this case, the syscall is `mknod(2)`, so we jump from instruction 145 to 150, which loads the second argument of the system call, which is a constant specifying what type of file to created. Instructions 153–155 test whether it's a character (`8192 == S_IFCHR`) or block (`24576 == S_IFBLK`) device; if so, we fake success on instruction 157; if not, the system call proceeds on instruction 156.

C. Assumption failure: Debian's `apt(8)` is more careful

An exception to the assumption that package managers don't care about consistency is Debian's `apt(8)`, which by default drops privileges for downloading packages over HTTP(S) and *also verifies that they were dropped correctly*. This validation fails under our seccomp filter. We work around the problem awkwardly by detecting `apt(8)` and `apt-get(8)` in `RUN` instructions and injecting `-o APT::Sandbox::User=root` into their command lines, which disables privilege dropping for download.

D. Performance evaluation

To properly evaluate the utility of our seccomp-based root emulation, users and sysadmins will need to understand its performance trade-offs, if any.

Using Charliecloud commit `ac2190f`, we built ten images from Charliecloud's examples and test suite with each of Charliecloud's three root emulation modes (none, fakeroot, and seccomp) as well as Docker 25.0.1 privileged (Type I) build, all with build cache turned off. We used a virtual machine in LANL's VMware vSphere cloud with 12 Intel Xeon cores and 32.0 GiB of RAM. Storage was a 4.00 TiB BTRFS filesystem backed by vSphere block devices. We ran each build 13 times and report median build times.

Table I summarizes our results. Mean speedup of seccomp mode over fakeroot was 1.12, no root emulation 1.02, and Docker 1.05. Excluding image `mpihello`, detailed below, speedup over fakeroot was 1.13, no root emulation 1.03, and Docker 1.14. That is, (1) Charliecloud's seccomp-based root emulation mode appears as fast or faster than a mature privileged alternative (Docker), and (2) directly comparable builds with and without seccomp filters, i.e. Charliecloud's none vs. seccomp, show that the seccomp filter imposes no meaningful performance penalty.

There is one image that only seccomp mode can build. Similarly to Figure 5, `seccomp` uses `mknod(2)` to create a device file; this is rejected by the other three modes. That is, *Charliecloud's seccomp root emulation mode is both comparably performant and more capable*.

The MPI-related images build on each other. The OpenMPI base image sequence is `almalinux_8ch` → `libfabric` → `openmpi`; `lammps` and `mpihello` are then based on `openmpi`. This raises some quirks in the experiment. First, one would expect that because none mode failed to build `almalinux_8ch`, the beginning of the chain, the other four MPI images should also fail to build, but they succeed. This is because when Charliecloud's cache is disabled, an image in the process of building is simply abandoned in place if the build fails, and then later builds with that as base image copy the partially built image as a starting point without validation. Apparently in this case, the failed base image was good enough for its descendants to build, though we did not test running. Second, this full copy of the base image explains the bad performance of Charliecloud compared to Docker on `mpihello`. This is a small Hello World application, so Charliecloud spends most of its build time copying the base image, rather than simply setting up an overlay like Docker. We expect enabling the cache would alleviate this deficit.

VI. Discussion

We present a novel root emulation mode based on the observation that distribution package managers and similar tools rarely need their privileged requests to be actually carried out (for HPC application container image build), but rather are satisfied to be simply told what they want to hear. We have implemented a seccomp filter for some privileged system calls that, instead of executing the syscall, does nothing and returns success to the userspace program. While limited to identity and files, this simple, zero-consistency root emulation performs well and is sufficient to build all the container images we tested, including one that Docker could not.

We do know of exceptions — builds that call `unminimize(8)` or trigger certain systemd scripts — but these both seem to be implementation hassles rather than something fundamental about our approach.

Alternately, one can use a complex, consistent root emulation using `fakeroot(1)` or `proot(1)`. Simple and complex both allow image build with a fully unprivileged Type III container implementation, a critical requirement for HPC

image	description	Docker	none	fakeroot	seccomp
`distroless`	Python interpreter plus a hello world program	1.4	0.99	0.93	1.1
`quick`	Alpine 3.17 base plus one package and dependencies	2.8	2.1	2.6	1.7
`seccomp`	Alpine 3.17 base plus C program that exercises mknod(2)				3.4
`debian_11ch`	Debian 11 base plus one package and dependencies	9.1		13	8.9
`mpihello`	Hello World MPI C program	3.1	9.9	10	9.8
`almalinux_8ch`	AlmaLinux 8 base plus various OS and compiled packages	150		150	130
`nvidia`	Ubuntu 20.04 plus CUDA from nVidia's repo; two stages	220		220	210
`lammps`	LAMMPS MPI-based MD code	320	340	330	330
`libfabric`	libfabric and various related libraries	320	310	340	330
`openmpi`	OpenMPI 4.1.4	350	340	340	340

TABLE I: Time in seconds to build each of the ten test images under Docker and Charliecloud's three root emulation modes, to two significant figures. Cell color indicates performance relative to the seccomp mode: red is faster while blue is slower, i.e. Charliecloud's seccomp mode prefers blue. Missing numbers mean the build could not complete. Seccomp mode is comparable to or faster than the alternatives, with one exception (`mpihello` under Docker) discussed in the text.

application containers. In our view, however, the simple, inconsistent seccomp method of root emulation has a number of advantages:

1) **Overhead.** The seccomp method imposes a relatively light overhead [21], [22] of its filter on every system call (not just those filtered), while the consistent method requires user-space emulation of system calls, making an extra program and possibly its shared libraries available to the container build, and state maintenance using a daemon process. In the case of Charliecloud, the `fakeroot(1)`-based root emulation imposes around 10–12% performance penalty.
2) **Simplicity.** The seccomp method has no user-space component and does nothing to actually emulate any system calls; further, "emulation" is complete once the filter is installed (though see `apt(8)` workaround above). Also, because it does not maintain state, the seccomp method intercepts fewer system calls.
3) **Compatibility.** The seccomp method is agnostic to libc and static/dynamic linking, and mostly agnostic to distribution, the exception being `apt(8)` above, though these properties are shared by PRoot and `fakeroot(1)` implementations based on `ptrace(2)`. Fewer intercepted system calls and no syscalls actually emulated has compatibility benefit as well.

On the other hand, the complex emulation is consistent on dimensions relevant to package management: a process under emulation can make changes to identity or privileged file metadata and have the emulated changes reflected back later. When this does matter for image build, simpler workarounds are available, e.g. for `apt(8)` above.

Future work includes (1) an optional wider set of emulated syscalls, such as `setxattr(2)`, which may allow systemd to be installed;[9] (2) evaluate adding *just a little* consistency, for user and groups IDs only, to remove the workaround for `apt(8)` explained above; (3) identify and characterize images the approach does not build correctly, and fix it to the extent practical; (4) deeper analysis of failure modes, e.g. in Figure 1b it's actually `cpio(1)` failing, not `rpm(8)`; and (5) more detailed performance testing.

[9]You might ask: "Why do I want systemd in my containers?" Indeed, you probably don't, but it tends to be pulled in as a dependency.

References

[1] R. Priedhorsky, R. S. Canon, T. Randles, and A. J. Younge, "Minimizing privilege for building HPC containers," in *Proc. SC*, Nov. 2021.
[2] J. W. Eaton, R. Faith, G. WIlford, F. Polacco, and C. Waton, "man(1)," Man page, Sep. 2023. [Online]. Available: https://man7.org/linux/man-pages/man1/man.1.html
[3] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in HPC," in *Supercomputing*, 2017.
[4] P. Moore and others, "libseccomp," The libseccomp Project, Apr. 2024. [Online]. Available: https://github.com/seccomp/libseccomp
[5] M. Kerrisk, "Namespaces in operation, part 1: Namespaces overview," *Linux Weekly News*, Jan. 2013. [Online]. Available: https://lwn.net/Articles/531114/
[6] ——, "Namespaces in operation, part 5: User namespaces," *Linux Weekly News*, Feb. 2013. [Online]. Available: https://lwn.net/Articles/532593/
[7] J. Dassen, j. witteveen, and C. Adams, "fakeroot(1)," Man page, Aug. 2021. [Online]. Available: https://manpages.debian.org/bullseye/fakeroot/fakeroot.1.en.html
[8] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, May 2017.
[9] Apptainer project, "Community announcement," Nov. 2021. [Online]. Available: https://apptainer.org/news/community-announcement-20211130/
[10] Sylabs Inc., "SingularityCE is Singularity," Jun. 2022. [Online]. Available: https://sylabs.io/2022/06/singularityce-is-singularity/
[11] D. Dykstra, "Apptainer without Setuid," Aug. 2022.
[12] C. Vincent *et al.*, "PRoot — chroot, mount –bind, and binfmt_misc without privilege/setup," Jan. 2022. [Online]. Available: https://proot-me.github.io/
[13] D. Trudgian, "proot based non-root / non –fakeroot builds," Aug. 2022. [Online]. Available: https://github.com/sylabs/singularity/issues/880
[14] P. Roszatycki, "fakechroot," Mar. 2019. [Online]. Available: https://github.com/dex4er/fakechroot/blob/2.20.1/man/fakechroot.pod
[15] M. Kerrisk, "Seccomp," Jan. 2024. [Online]. Available: https://man7.org/training/download/splc_seccomp_slides-mkerrisk-man7.org.pdf
[16] Docker Inc., "Seccomp security profiles for Docker," May 2023. [Online]. Available: https://docs.docker.com/engine/security/seccomp/
[17] distrobuilder contributors, "distrobuilder documentation," Apr. 2023. [Online]. Available: https://linuxcontainers.org/distrobuilder/docs/latest/
[18] "Features," Dec. 2015. [Online]. Available: https://firejail.wordpress.com/features-3
[19] R. Swiecki *et al.*, "nsjail," Apr. 2024. [Online]. Available: https://github.com/google/nsjail
[20] F. Abecassis and J. Calmels, "Distributed HPC applications with unprivileged containers," Feb. 2020. [Online]. Available: https://archive.fosdem.org/2020/schedule/event/containers_hpc_unprivileged/
[21] M. Larabel, "Seccomp filters get a very nice speed-up with Linux 5.11," Dec. 2020. [Online]. Available: https://www.phoronix.com/news/Linux-5.11-SECCOMP-Performance
[22] Zatoichi, "Zatoichi's Engineering Blog," Nov. 2017. [Online]. Available: https://zatoichi-engineer.github.io/2017/11/06/seccomp-bpf.html

APPENDIX A
REPRODUCIBILITY

A. Software

1) Charliecloud commit `ac2190f`.[10]
2) Docker 25.0.1. Storage path will need to be configured in `/etc/docker/daemon.json`.
3) `seccomp-tools` Ruby gem version 1.6.1.
4) Python environment described in Listing 1. We used `micromamba` to install it.

B. Figures

Most of the figures are terminal transcripts using the input given. Figures 2 and 4 are self-contained. Figure 6 is the result of the command "`seccomp-tools dump -c 'ch-run --seccomp alpine:3.19 -- true'`".

C. Performance experiment

The workflow uses two scripts, `buildem` (Listing 2) and `plotem` (Listing 3). Several paths within the scripts will need to be changed to match your system. Place them in a new directory; we'll refer to it as `perf-eval`.

Input Dockerfiles are from the Charliecloud source code. Several will need auxiliary files, also from Charliecloud. Place them in `perf-eval/examples`. Also create an empty directory `perf-eval/out`. See Listing 4.

Change directory to `perf-eval` and execute `./buildem foo.csv`; this took 60–90 minutes for us. We ran the 13 iterations described in a Bash `seq(1)` loop overnight.

Once this is complete, move the CSV files into subdirectory `out`. Then run `./plotem`, which analyzes the data (just a few seconds) and produces a LaTeX file `out.tex` that is included into the paper to produce Table I. It also prints the corresponding dataframes on stdout, which may be easier to compare to the paper.

Listing 1: `environment.yml`

```
name: spe
channels:
  - conda-forge
  - nodefaults
dependencies:
  # Python version
  - python==3.12.5
  # Conda packages
  - freezegun=1.5.0
  - matplotlib==3.9.1
  - pandas==2.2.2
  - seaborn==0.13.2
  # Non-explicit dependencies
```

Listing 2: `buildem`

```
#!/usr/bin/env python3

import csv
import datetime
import glob
import os.path
import subprocess
import sys


def main():
   out_path = sys.argv[1]
   if (len(sys.argv) > 2):
      df_paths = sys.argv[2:]
   else:
      df_paths = sorted(  glob.glob("**/*.dockerfile", recursive=True)
                        + glob.glob("**/Dockerfile*", recursive=True))

   out = Out_CSV(out_path)
   INFO("*** opened output: %s" % (out))
```

[10] https://github.com/hpc/charliecloud/tree/ac2190f

```
21
22    for df_path in df_paths:
23       df = Dockerfile(df_path)
24       INFO("*** dockerfile: %s" % df)
25       for name in globals():
26          if (name[:2] == "T_"):
27             i = globals()[name](df)
28             INFO("** builder: %s" % i)
29             out.write_build(i, *i.build())
30
31    out.close()
32    INFO("*** done")
33
34
35 class Dockerfile:
36
37    __slots__ = ("path",
38                 "name")
39
40    def __init__(self, path):
41       self.path = path
42       cs = self.path.split("/")
43       if (cs[-1] == "Dockerfile"):
44          self.name = cs[-2]
45       else:
46          self.name = os.path.splitext(cs[-1])[1][1:]
47
48    def __str__(self):
49       return "%s@%s" % (self.name, self.path)
50
51    @property
52    def context(self):
53       return os.path.split(self.path)[0]
54
55
56 class Out_CSV:
57
58    def __init__(self, path):
59       self.path = path
60       self.fp = open(path, "a", newline="")
61       self.csv = csv.writer(self.fp)
62
63    def close(self):
64       self.fp.close()
65
66    def write_build(self, test, exit_code, t_build):
67       INFO("writing result: %s, exit %d" % (t_build, exit_code))
68       time_str = datetime.datetime.now().isoformat(timespec="milliseconds")
69       self.csv.writerow([time_str, test.builder_name, test.df.name,
70                          exit_code, t_build.total_seconds()])
71       self.fp.flush()
72
73    def __str__(self):
74       return self.path
75
76
77 class Test:
78
79    __slots__ = ("builder_name",
80                 "df")
81
82    def __init__(self, df):
83       self.df = df
84
85    def __str__(self):
86       return "%s: %s" % (self.builder_name, self.df)
87
88    def build(self):
89       INFO("executing: %s" % self.cmd)
```

```
      t_start = datetime.datetime.now()
      cp = subprocess.run(self.cmd)
      t_end = datetime.datetime.now()
      return (cp.returncode, t_end - t_start)


class Charliecloud(Test):

   __slots__ = ("force_mode")

   @property
   def cmd(self):
      return ["ch-image", "build",
              "--storage", "/scratch/reidpr.ch_" + self.builder_name,
              "--no-cache", "--force", self.force_mode,
              "-t", self.df.name, "-f", self.df.path, self.df.context]

class T_Charliecloud_None(Charliecloud):
   builder_name = "ch.none"
   force_mode = "none"

class T_Charliecloud_Fakeroot(Charliecloud):
   builder_name = "ch.fakr"
   force_mode = "fakeroot"

class T_Charliecloud_Seccomp(Charliecloud):
   builder_name = "ch.seco"
   force_mode = "seccomp"

class T_Docker(Test):

   builder_name = "docker"

   @property
   def cmd(self):
      return ["sudo", "docker", "build",
              "--no-cache",
              "-t", self.df.name, "-f", self.df.path, self.df.context]

def INFO(msg):
   if (sys.stderr.isatty()):
      color_start = "\033[38;5;207m"
      color_stop = "\033[0m"
   else:
      color_start = ""
      color_stop = ""
   time_str = datetime.datetime.now().strftime("%m/%d %H:%M:%S.%f")[:-3]
   print("%s%s %s%s" % (color_start, time_str, msg, color_stop),
         file=sys.stderr)


if (__name__ == "__main__"):
   main()
```

Listing 3: `plotem`

```
#!/usr/bin/env python3

import glob
import os
import math

import freezegun
import matplotlib as mpl
import pandas as pd
import seaborn as sns

# Configure.
pd.options.mode.copy_on_write = True
```

```
#pd.options.display.max_columns = None

# Reproducible output.
os.environ["SOURCE_DATE_EPOCH"] = "0"
FREEZE_TIME = "1994-03-01"
freezer = freezegun.freeze_time(FREEZE_TIME)
freezer.start()

# Configure plots.

# Load the data.
print("loading data ...")
dfs = list()
for path in glob.glob("./out/*.csv",):
   df = pd.read_csv(path,
                    names=["t_end", "builder", "image", "fail", "t"],
                    parse_dates=["t_end"],
                    index_col=["image", "builder", "t_end"])
   dfs.append(df)
   #print(df)
builds_all = pd.concat(dfs)
print("loaded %d builds" % len(builds_all))

# delete builds that failed
builds = builds_all[builds_all["fail"] == 0].drop(columns=["fail"])
print("kept %d builds that succeeded" % len(builds))

# compute median and tidy
median = builds.groupby(["image", "builder"]) \
               .median() \
               .unstack() \
               .droplevel(0, axis="columns") \
               .sort_values("ch.seco") \
               .reindex(columns=["docker", "ch.none", "ch.fakr", "ch.seco"])
print(median)

# compute normalized median
median_norm = median.div(median.loc[:, "ch.seco"], axis="rows")
print(median_norm)

# mean speedup of seccomp
print("MEAN SPEEDUP:")
print(median_norm.mean())
print("MEAN SPEEDUP WITHOUT mpihello:")  # which is weird
print(median_norm.drop(index=["mpihello"]).mean())

# add description column
median.insert(0, "description",
              ["Python interpreter plus a hello world program",
               "Alpine 3.17 base plus one package and dependencies",
               "Alpine 3.17 base plus C program that exercises mknod(2)",
               "Debian 11 base plus one package and dependencies",
               "Hello World MPI C program",
               "AlmaLinux 8 base plus various OS and compiled packages",
               "Ubuntu 20.04 plus CUDA from nVidia's repo; two stages",
               "LAMMPS MPI-based MD code",
               "libfabric and various related libraries",
               "OpenMPI 4.1.4"])
#print(median)

# dump LaTeX table
norm=mpl.colors.Normalize(-1, 1)
cmap=mpl.colors.LinearSegmentedColormap.from_list(
   "a", sns.color_palette("RdBu", 13)[2:-2])#.reversed()
bg_map = mpl.cm.ScalarMappable(norm=norm, cmap=cmap)
def P(text):
   print(text, file=fp)
def BG(f):
   return "%s,%s,%s" % bg_map.to_rgba(math.log2(f))[:3]
```

```
with open("out.tex", "wt") as fp:
   # note: hard to \input body of table only; see:
   # https://tex.stackexchange.com/questions/641441
   # https://stackoverflow.com/questions/26212089
   P(r"""\begin{tabular}{llSSSS}
         \toprule
           \multicolumn{1}{c}{\textbf{image}}
         & \multicolumn{1}{c}{\textbf{description}}
         & \textbf{Docker}
         & \textbf{none}
         & \textbf{fakeroot}
         & \textbf{seccomp}
         \\
         \midrule""")
   for i in range(median.shape[0]):        # rows
      row = median.iloc[i, :]
      P(r"\code{%s} & %s" % (row.name, row.iat[0]))
      for j in range(1, median.shape[1]):  # columns
         t = row.iat[j]
         t_norm = median_norm.iat[i, j-1]
         if (math.isnan(t)):
            P(r"&")
         else:
            P(r"& \cellcolor[rgb]{%s} %f" % (BG(t_norm), t))
      P(r"\\")
   P(r"\bottomrule")
   P(r"\end{tabular}")
```

Listing 4: File locations

```
$ ls -R perf-eval
perf-eval:
buildem  examples  out

perf-eval/examples:
distroless                Dockerfile.libfabric  Dockerfile.quick  seccomp
Dockerfile.almalinux_8ch  Dockerfile.nvidia     lammps
Dockerfile.debian_11ch    Dockerfile.openmpi    mpihello

perf-eval/examples/distroless:
Dockerfile  hello.py

perf-eval/examples/lammps:
Dockerfile  melt.patch  simple.patch

perf-eval/examples/mpihello:
Dockerfile  hello.c  Makefile  slurm.sh

perf-eval/examples/seccomp:
Dockerfile  mknods.c
```