ORNL/TM-2015/210

Review of Enabling Technologies to Facilitate Secure Compute Customization

Ferrol Aderholdt
Blake Caldwell
Susan Hicks
Scott Koch
Thomas Naughton
Daniel Pelfrey
James Pogge
Stephen L. Scott
Galen Shipman
Lawrence Sorrillo

December 2014

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2015/210

Computing & Computational Sciences Directorate

DoD-HPC Program

Review of Enabling Technologies to Facilitate Secure Compute Customization

Ferrol Aderholdt[2], Blake Caldwell[1], Susan Hicks[1], Scott Koch[1],
Thomas Naughton[1], Daniel Pelfrey[1], James Pogge[2],
Stephen L. Scott[1,2], Galen Shipman[2] and Lawrence Sorrillo[1]

[1] Oak Ridge National Laboratory
Oak Ridge, TN 37831

[2] Tennessee Technological University
Cookeville, TN, 38501

Date Published: December 2014

Prepared by
OAK RIDGE NATIONAL LABORATORY
P.O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Executive Summary

Review of Enabling Technologies to Facilitate Secure Compute Customization

High performance computing environments are often used for a wide variety of workloads ranging from simulation, data transformation and analysis, and complex workflows to name just a few. These systems may process data for a variety of users, often requiring strong separation between job allocations. There are many challenges to establishing these secure enclaves within the shared infrastructure of high-performance computing (HPC) environments.

The isolation mechanisms in the system software are the basic building blocks for enabling secure compute enclaves. There are a variety of approaches and the focus of this report is to review the different virtualization technologies that facilitate the creation of secure compute enclaves. The report reviews current operating system (OS) protection mechanisms and modern virtualization technologies to better understand the performance/isolation properties. We also examine the feasibility of running "virtualized" computing resources as non-privileged users, and providing controlled administrative permissions for standard users running within a virtualized context. Our examination includes technologies such as Linux containers (LXC [32], Docker [15]) and full virtualization (KVM [26], Xen [5]).

We categorize these different approaches to virtualization into two broad groups: OS-level virtualization and system-level virtualization. The OS-level virtualization uses *containers* to allow a single OS kernel to be partitioned to create *Virtual Environments (VE)*, e.g., LXC. The resources within the host's kernel are only virtualized in the sense of separate namespaces. In contrast, system-level virtualization uses *hypervisors* to manage multiple OS kernels and virtualize the physical resources (hardware) to create *Virtual Machines (VM)*, e.g., Xen, KVM. This terminology of VE and VM, detailed in Section 2, is used throughout the report to distinguish between the two different approaches to providing virtualized execution environments.

As part of our technology review we analyzed several current virtualization solutions to assess their vulnerabilities. This included a review of common vulnerabilities and exposures (CVEs) for Xen, KVM, LXC and Docker to gauge their susceptibility to different attacks. The complete details are provided in Section 5 on page 33. Based on this review we concluded that system-level virtualization solutions have many more vulnerabilities than OS level virtualization solutions. As such, security mechanisms like sVirt (Section 3.3) should be considered when using system-level virtualization solutions in order to protect the host against exploits. The majority of vulnerabilities related to KVM, LXC, and Docker are in specific regions of the system. Therefore, future "zero day attacks" are likely to be in the same regions, which suggests that protecting these areas can simplify the protection of the host and maintain the isolation between users.

The evaluations of virtualization technologies done thus far are discussed in Section 4. This includes experiments with `user` namespaces in VEs, which provides the ability to isolate user privileges and allow a user to run with different UIDs within the container while mapping them to non-privileged UIDs in the host. We have identified Linux namespaces as a promising mechanism to isolate shared resources, while maintaining good performance. In Section 4.1 we describe our tests with LXC as a non-root user and leveraging namespaces to control UID/GID mappings and support controlled sharing of parallel file-systems. We highlight several of these namespace capabilities in Section 6.2.3.

The other evaluations that were performed during this initial phase of work provide baseline performance data for comparing VEs and VMs to purely native execution. In Section 4.2 we performed

tests using the High-Performance Computing Conjugate Gradient (HPCCG) benchmark to establish baseline performance for a scientific application when run on the Native (host) machine in contrast with execution under Docker and KVM. Our tests verified prior studies showing roughly 2-4% overheads in application execution time & MFlops when running in hypervisor-base environments (VMs) as compared to near native performance with VEs. For more details, see Figures 4.5 (page 28), 4.6 (page 28), and 4.7 (page 29). Additionally, in Section 4.3 we include network measurements for TCP bandwidth performance over the 10GigE interface in our testbed. The Native and Docker based tests achieved $\geq$ 9Gbits/sec, while the KVM configuration only achieved 2.5Gbits/sec (Table 4.6 on page 32). This may be a configuration issue with our KVM installation, and is a point for further testing as we refine the network settings in the testbed. The initial network tests were done using a bridged networking configuration.

The report outline is as follows:

- Section 1 introduces the report and clarifies the scope of the project, to include working assumptions about the threat model for customizable computing resources.
- Section 2 provides background and defines terminology used throughout the report. This section also includes details on security and virtualization classifications.
- Section 3 reviews isolation mechanisms for container and hypervisor based solutions. This section provides information on security frameworks for use with virtualization. A synopsis of management platforms for virtualization technologies, which brings these various pieces together is also included in this section.
- Section 4 provides details on evaluations done thus far, to include: namespace tests with LXC, memory/compute benchmarking with HPCCG [34], and TCP Bandwidth tests with `iperf` [24].
- Section 5 discusses a vulnerability assessment that reviewed common vulnerabilities and exposures (CVEs) for Xen, KVM, LXC and Docker CVEs Xen, KVM, LXC, Docker and the Linux kernel.
- Section 6 summarizes the report and highlights a observations from this initial phase as well as points of interest for subsequent phases of the project.

Chapter 1

Introduction

The ability to support “on-demand self-service” computing resources is a major asset of Cloud Computing [35]. These customizable environments are made possible by modern operating system (OS) mechanisms and virtualization technology, which allow for decoupling the physical and virtual resources. This separation enables users to customize “virtualized” computing resources, while maintaining appropriate protections to ensure control is maintained at the hosting (physical) level.

There are many factors that influence the degree of control and in this report we review these factors to gain insights into the current state of practice. The objectives of the report are to: (i) review current OS protection mechanisms and virtualization technologies to better understand the performance/isolation properties; (ii) examine the feasibility of running “virtualized” computing resources as non-privileged users, i.e., non-root users; (iii) analyze the vulnerabilities of current virtualization based environments. Additionally, the report provides insights into ways that available protection mechanisms may be used to better isolate use of shared resources in a multi-tenant environment[1]. Specifically we examine the use case of sharing common distributed parallel filesystems and a high speed network infrastructure.

1.1 Project Scope

The overall goal of this project is to evaluate the current technologies used to create user customizable computing platforms with respect to their security and isolation qualities. The basic assumption is that virtualization plays a central role in enabling secure compute customization. As such a thorough review of the current state of relevant virtualization technology is important to better understand the challenges and opportunities for deploying user customizable computing resources.

1.1.1 Customizable Computing Resources

In multi-user computing environments the access to common computing, network and storage resources are shared among many users. In contrast, this project is focused on a multi-tenant environment that allows users to customize the computing platform specific to their needs, while still sharing some physical network and storage resources. The customizability in a multi-user computing environment is limited to unprivileged changes such as altering the shell and environment variables while using the shared system in the context of a particular user. In a multi-user computing environment users are aware of other

[1]In subsequent months of the project, we anticipate leveraging these insights for node-level isolation mechanisms to explore ways to leverage them for protection with shared-storage and network aspects.

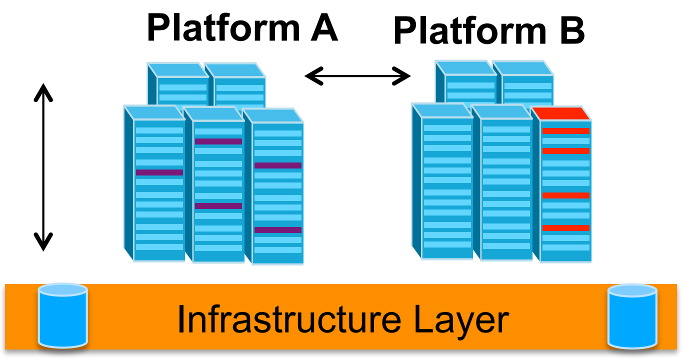


Figure 1.1. Illustration of two axes of administrative control (platform and infrastructure) in a customizable computing environment.

users on the system and of all processes running on the system. The reason for limiting users to unprivileged operations on the system is so that one user cannot bypass system security controls and affect the quality of service of another user. However in a multi-tenant environment, each tenant's computing platform is isolated from each other, meaning that they can be allowed a high degree of control as to how their platform is customized. For example a tenant may run a completely different Linux distribution with customized system-level functionality such as logging authorization mechanisms that have no bearing on other tenants. This can be done while still sharing physical resources such as network and storage. In this project a customizable computing platform is in the context of a multi-tenant model where even system-level changes are permissible. The isolation of computing platforms between each other and also the underlying infrastructure enables a distinction to be made between platform admins and infrastructure admins who manage the physical resources and services on top of which secure computing platforms are deployed (Figure 1.1).

1.1.2 Threat Model

Virtualization technologies that enable customizable environments involve a deep "software stack", from the infrastructure and system layers to the application layer. The security at each of these levels must be considered when evaluating the system as a whole. Furthermore, in this study we use isolation mechanisms to limit the exposure of particular levels in the software stack that might have weak security controls. The reinforcement of these controls through isolation mechanisms (e.g., virtualization) allows for more fine-grained resource marshalling, whereas relying purely on strict controls might limit the usefulness of a customized compute environment. Thus, to establish a framework for our evaluation and to identify remaining gaps for future work we developed the following threat models assumed for this study.

We assume a multi-tenant model where many users make use of shared infrastructure for separate tasks. Isolation must be maintained between users and their data. When virtualization involves sharing of system level resources, (e.g. memory, CPU caches, I/O devices), there is a potential attack vector of side-channel attacks between environments co-located on the same system. We assume granularity at the node level, such that a single user has ownership of a compute node where memory, CPU cores, or PCI devices are not shared.

While applications for different users do not share the same node, they may share physical network infrastructure and storage resources. The shared storage could be at the block-level granularity, such that data belonging to each user does not share filesystem data structures, or at the granularity of a subset of a shared filesystem. Our use case with respect to a customizable secure compute environment focuses on sharing a distributed or parallel filesystem such as Lustre or GPFS. A shared filesystem raises several challenges in a multi-tenant environment, and of those challenges, we address those related to configuring customizable compute environments for isolating segments of a shared filesystem and only providing a user access to explicitly approved segments.

In summary, the working assumptions/requirements for a prototypical system are:

- Granularity is at the node level, i.e., single user per node (Therefore not concerned with on-node, cross-user security attacks or snooping, i.e., memory of neighbor in co-hosted VM/VE)
- Must support controlled user permission escalation, i.e., user may obtain `root`, but only in VE/VM.
- Users can not escalate beyond set permissions/access granted to VE/VM (e.g., maintaining only limited access rights on a shared filesystem)
- Maintain "acceptable" performance levels, where "acceptable" will be defined as some percentage of native performance balanced with added protection capabilities. Performance implications are addressed in Section 4.2.
- Incorporate deployment/maintenance overheads when considering viability of different technologies (i.e., practicality factor). This is discussed in detail in Section 3.4.

1.2 Report Outline

The report is structured as follows, in Chapter 2 we define important terminology, delineating protection and security, and review security and virtualization classifications. Chapter 3 focuses on key virtualization technologies that are relevant to this project. Testing and evaluations related to these technologies that have been performed thus far in the project are described in Chapter 4. A vulnerability assessment for different virtualization technologies is given in Chapter 5. In Chapter 6 we discuss our observations and conclusions.

Chapter 2

Background

There are several terms that get used somewhat interchangeably and have different connotations depending on your background and area of expertise. To avoid these ambiguities, we define important terminology and review classifications that will help to structure the remainder of the report.

2.1 Terminology

Protection vs. Security: The title of this project includes the term "secure". Therefore we begin by clarifying our distinction between *protection* and *security*. The topic of security is important and a set of security classifications are discussed in Section 2.2. However, the focus of this project is on specific protection and isolation mechanisms, which can be used to create and enforce security policies. The underlying mechanisms provide the building blocks to create security.

Virtualization Variants: In Section 2.3 we provide details about different container and hypervisor based virtualization classifications. Generally speaking, throughout the report we distinguish between hypervisor and container-based virtualization configurations using the terms, Virtual Machine (*VM*) and Virtual Environment (*VE*), respectively.

Virtual Machine (VM) – type-I/type-II virtualization (hypervisors); VMs may include multiple OS kernels and the virtualization layer extends below the kernel to virtualization of the "hardware."

Virtual Environment (VE) – OS-level virtualization (containers); VEs share a single OS kernel with the host and include virtualization of the "environment." Resources within the host's kernel are only virtualized in the sense of separate namespaces.

2.2 Security Classifications

The "Orange Book" [14] is a requirements guideline published by the Department of Defense (DoD) in 1985. This publication defined both the fundamental requirements for a computer system to be considered secure and multiple classification levels in order to describe the security of a given system.

The fundamental requirements for secure computing systems are:

1. Security Policy. The security policy defines the mandatory security policy for the system. This may include separation of privilege levels and the implementation of a need-to-know access list.

2. Marking. Marking includes the ability for access control labels to be associated with system objects as well as have the ability to assign sensitivity levels to objects and subjects.

3. Identification. Within a secure system, each subject must be identified in order to properly assert the security policy.

4. Accountability. Logging must be used in order to properly hold each subject accountable for their actions while logged into the system.

5. Assurance. Each component of the security policy implementation must reside within independent mechanisms.

6. Continuous Protection. During the execution of the system, each component must retain its integrity in order to be considered trusted.

The criteria of the security for a system ranges from division *D* to division *A*. Division *D* represents systems with minimal security, while division *A* represents systems with a high level of verifiable security. A system that is labeled as a division *D* system is considered to provide minimal protection and is reserved for any system that does not meet the criteria for any higher rated system.

Division *C* systems are broken into two classes: *C1* and *C2*. A system meeting the criteria for class *C1* provides the following fundamental requirements: (i) security policy, (ii) accountability, and (iii) assurance. With respect to requirement (i), a system must have discretionary access control in which access controls are used between objects and users with the ability to share objects amongst users or groups. Requirement (ii) states that the system should provide the user with the ability to login with the use of a password. Requirement (iii) directs the configuration to provide system integrity as well as a separation of privilege for user applications and the OS kernel. Additionally, the system should be tested with respect to the security mechanisms used to provide each of these requirements.

A *C2* system provides all of the requirements with some additional requirements. The kernel should provide sanitized objects before use or reuse. Additionally, the kernel should maintain logs of the following events: use of authentication mechanisms, file open operations, process initiation, object deletion, and user actions.

Division *B* criteria moves from *discretionary* to *mandatory* protection for the system. This division contains multiple classes of increasing protection from class *B1* through *B3*. In addition to providing an increased amount of protection, the system developer must also provide a security policy model as well as the specification for the model.

Class *B1* requirements are the same as *C2* with the addition of sensitivity labels, mandatory access control, and design specification and verification. The sensitivity labels allow for the ability to have multiple trust levels per user, per object, or both. The mandatory access control will leverage the security labels in order to enforce the security policy. The sensitivity policy that will be deployed in a class *B1* system, and from hence forth, is one in which a subject can read at their sensitivity level or lower and write at their sensitivity level or higher. This prevents objects from becoming untrusted and removes the possibility of users observing data outside of their sensitivity level. The design specification and verification of the system may be done as an informal or formal model and must be maintained over the life span of the system. Additionally, all claims made about the system must be verified or verifiable.

Class *B2* provides structured protection as well as a verifiable security policy model. The majority of requirements for a system to be considered of class *B2* are also required as a class *B1*. The additions from the *B1* requirements are within the fundamental requirements areas such as the security policy, accountability, and assurance. The security policy is extended to support device sensitivity labels and users have the ability to query their sensitivity level at runtime. The accountability requirement is extended to supported "trusted path" communication between the user and the kernel. The assurance requirements are considerably extended from the *B1* to the *B2* requirements. In a *B2* class, the kernel must execute within its own domain, maintains process isolation through address space provisioning, and is structured into independent modules with a separation between protection-critical and non-protection critical elements. Additionally, covert channels are searched for and analyzed.

Class *B3* requires a complete implementation of a reference monitor, which will provide mediation on all access of objects by subjects, be tamper-proof, and be small enough to be verifiable. Additionally, the system must contain recovery procedures in the case of faults or attacks as well as be highly resistant to penetration. With respect to the fundamental requirements for the system, minor additions are required within the accountability and assurance requirements when compared to class *B2*. In a class *B3* system, the trusted path is required like class *B2*, but the path should be isolated and distinguishable from any other path. The auditing system should be able to track the various security events and determine if any predefined thresholds have been exceeded. If the events are to continue, the system should terminate the event in the least disruptive manner. The system architecture should have minimal complexity with simple protection mechanisms while providing significant abstraction.

Division *A* provides the most secure systems. However, these systems require complete verification, which is not feasible with the average size and complexity of modern systems.

2.3 Virtualization Classification

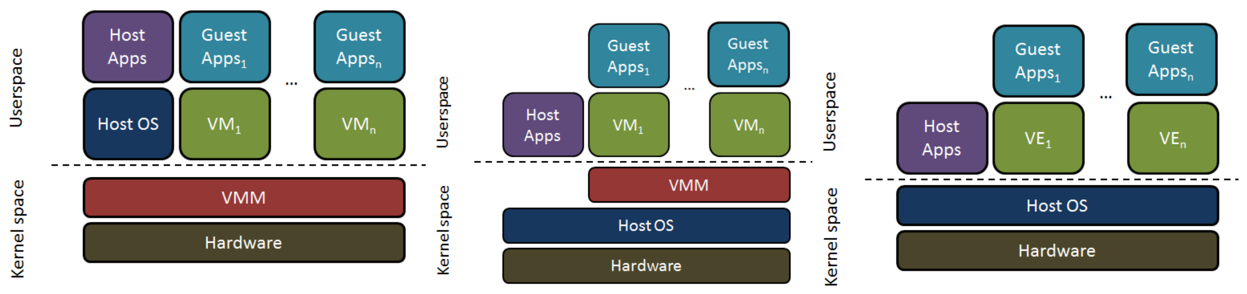
Virtualization is the abstraction of the system layer in order to achieve various goals including isolated execution, compute customization, and environment portability. A benefit enjoyed by cloud computing environments is making use of the abstraction for resource sharing, allowing for higher resource utilization through statistical multiplexing and oversubscription. Of more direct interest to this report, the virtualization layer provides a demarcation point, above which distinct VMs or VEs can be customized in a portable fashion, while layers below enforce inter-VM or VE isolation and act as a trusted arbitrator for system resources and hardware.

2.3.1 OS Level Virtualization

OS level virtualization, or container-based virtualization, is the abstraction of the OS such that processes and libraries are isolated within virtual environments (VE). These VEs are owned and created by a user on the host system via a set of user-level tools. The user-level tools leverage the kernel's isolation capabilities in order to provide the abstraction with respect to a VE's unique set of processes, users, and file system. A VE employs a single kernel instance, which manages the isolation for the "containers" where processes execute. This VE architecture is illustrated in Figure 2.1(c).

2.3.2 System-level Virtualization

System-level virtualization is the abstraction of the hardware such that execution environments are isolated within virtual machines (VM). Each VM is isolated and managed by a virtual machine monitor (VMM), also known as a hypervisor. The VMM is a thin software layer that may reside on top of the hardware, or on top of (or beside) an administrative OS known as a host OS. An example of both types can



(a) Type-1 Virtualization (b) Type-2 Virtualization (c) OS-level Virtualization

Figure 2.1. Overview of various virtualization architectures

be seen in Figure 2.1. A VM configuration employs multiple kernel instances, one per VM, which are managed by the VMM (hypervisor).

There two general architectures used to describe where the VMM exists within a system architecture [18]: *type-1* and *type-2* VMM. As shown in Figure 2.1(a), a type-1 VMM exists above the hardware. A type-2 VMM, Figure 2.1(b), exists on top of (or beside) the host OS. In a type-1 VMM, the host OS is often implemented as a privileged VM, which contains many of the hardware device drivers used by the system. The host OS for a type-2 VMM executes natively. An example of a type-1 VMM is the Xen hypervisor [5], while the kernel-based virtual machine (KVM) [21, 30] is an example of a type-2 VMM.

Chapter 3

Virtualization

Virtualization dates back to the 1960's with research performed at IBM in conjunction with their large time-shared systems [12, 19]. In these environments the resources were prohibitively expensive, such that the resources needed to be shared among users. As noted by Goldberg [18], virtual machines enhanced system multiplexing (e.g., "multi-access, multi-programming, multi-processing") to include the entire platform ("multi-environments").

Interestingly, many of the initial motivating factors that led to the use of virtual machines (machine costs, user accessibility, development on production environments, security, reliability, etc.) are true of today's large-scale computing environments [23, 28, 46, 47]. The early IBM VM/370 systems included additional hardware support for virtual machines [12]. The recent resurgence of interest in virtualization [17] has led to hardware enhancements to support virtualization on commodity architectures (e.g., Intel [45], AMD [1]).

Virtualization has re-emerged as a building block to aid in the construction of systems software. This infrastructure technology has been used in the past to support system multiplexing and to assist with compatibility during system evolution (e.g., IBM 360 virtual machines [12]). The approach has received renewed interest to enhance security (trusted computing base), improve utilisation (over-subscription), and assist system management (snapshots/migration). In this chapter we review relevant virtualization technologies that will be used in our feasibility study.

3.1 OS level virtualization

With respect to this work, all user-level tools will be leveraging mechanisms present within the Linux kernel. The Linux kernel has two primary mechanisms that are used to implement isolation for container-based, single-OS kernel virtualization: (i) namespaces and (ii) control groups (cgroups).

3.1.1 Namespaces

Namespaces provide isolations for various resources as well as users. Currently, there are six namespaces present in the Linux kernel[1]. These namespaces are summarized in Table 3.1.

The first namespace to be supported by the Linux kernel was for controlling file system mounts. The `mnt` namespace allows for isolating one namespace instance from another instance. This feature dates back to Linux version 2.4.19 and allows mounts within a namespace to be invisible outside the context of the

[1]As of Nov-2014, Linux v3.18.

Kernel	Namespace	Description
$\geq$2.4.19	`mnt`	mount points & file systems to be isolated, (i.e., file system mounts in one namespace are hidden from another namespace)
$\geq$2.6.19	`ipc`	Inter-Process Communication mechanisms within namespace
$\geq$2.6.19	`uts`	hostname and domain name separate from values at host
$\geq$2.6.24	`pid`	process isolation between namespaces
$\geq$2.6.29	`net`	isolates the network devices and network stack
$\geq$3.8	`user`	separate lists of users per namespace; allows for separation of privileges between the host and the guest.

Table 3.1. Available Linux namespaces and required kernel version.

namespace. Subsequently, inter-process communication (IPC) and hostname/domainname isolation mechanisms were introduced in Linux version 2.6.19 with the `ipc` and `uts` namespaces, respectively.

The isolation of entire processes between namespaces was added with the `pid` namespace in Linux version 2.6.24. This allows for two processes running on the same machine to be visible from the host but entirely invisible to each other. For example, a process listing (`ps`) from the host shell will show Process-A in Container-A and Process-B in Container-B. However, a process listing within Container-A will not show Process-B and vise versa.

The isolation of network devices, and the network stack as a whole, on a per-container basis was introduced in Linux version 2.6.29 with the `net` namespace. This provides a logical copy of the network stack, including: routes, firewall rules, and network devices, loopback device, SNMP statistics, all sockets, and network related *procfs* and *sysfs* entries. When using the `net` mechanism for devices and sockets, the network device belongs to exactly 1 network namespace, and the socket belong to exactly 1 network namespace.

The most recent addition was the `user` namespace, which establishes per-namespace contexts for user ID's (UIDs) and group ID's (GIDs). UIDs and GIDs when combined with capability sets (discussed in 3.3.4) are the basic security attributes in Linux for defining allowed operations on files, processes, or system resources. `User` namespaces, as an isolation mechanism, work in conjunction with these attributes to perform security enforcement specific to the context of a `user` namespace and only to the resources within that namespace. When applied to containers, a container runs within the context of a child `user` namespace distinct from the host OS's parent `user` namespace. In this scenario, a user in the host context can be mapped to a different user within the container, even the container's root user. This user is able perform administrative functions within the container, such as installing packages, and system operations such that: (i) the user possesses the capability set to do so, and (ii) the resource on which it is acting on is owned by the `user` namespace. For example, an user who is privileged within the child `user` namespace who attempts operations on `net` and `mnt` namespaces of a parent `user` namespace would be denied if the user is not privileged with the necessary capabilities in the parent namespace. However, new `network` and `mnt` namespaces that are created within the child `user` namespace may be modified. Launching an LXC container (as evaluated in Section 4.1) will cause new `net` and `mnt` namespaces to be created inside the child `user` namespace so that unprivileged users may modify them.

Any user on the system can create a nested namespace, such that the nesting level does not exceed up to 32 levels. When a process in the parent namespace creates a new user namespace, the processes's effective user becomes the child namespace's owner and inherits all capabilities in the new namespace by default. Other processes can be placed within the same child user namespace and return to their parent namespace, but processes may only exist in a single namespace at any point in time.

Access to system resources within `user` namespaces are controlled by the host OS kernel in the following way. When any user performs a system call (e.g. *open()*, *mount()*, *write()*), the host kernel will evaluate whether to allow the operation by mapping the UID in the calling namespace to the UID in the namespace where the target resource resides (and check the capabilities set within the target's namespace). In this way the root user within a container, which is mapped to an unprivileged user may not un-mount a filesystem on the host, because it only possesses the capabilities to un-mount a filesystem within the context of the container's namespace. System calls such as *getuid()* return the UID within the context of the calling process, meaning applications executing within the container are unaffected by the existence of different mappings on the host.

There is a significant degree of flexibility in how UIDs can be mapped between the host OS and a container. An unprivileged user on the host OS can create a child namespace, but by default only their own UID is mapped within the container (as root). To extend this behavior, the root user on the host OS can configure the allowed mappings such that unprivileged users can map ranges of UIDs on the host to within the container. A typical usage is to allow UIDs within the container to be mapped to very high UIDs on the host (e.g. 1,000,000), such that they remain unprivileged on the host, but the full range of 65k users can exist within the container (up to 1,065,534).

Having introduced the use of the namespaces such as `net`, `mnt`, and `user` namespaces related to kernel isolation mechanisms, it is also necessary to distinguish these from another type of namespace used in the context of filesystems. A filesystem namespace involves the hierarchical naming scheme of directories and files, where a file is uniquely identified by its path. Filesystem namespaces can be nested, where one filesystem's namespace is rooted at a mount point within another filesystem. There will be some overlap of these definitions when discussing isolating filesystem namespaces in Section 4.1. There we restrict the files a container may access to those files rooted at a particular point in the filesystem directory hierarchy. The two filesystem namespaces in that discussion are the global directory structure and the chroot'd namespace making up the files which a container may access.

3.1.2 Cgroups

Linux Control Groups (cgroups) provide a mechanism to manage resources used by sets of tasks [10]. This mechanism partitions sets of tasks into hierarchical groups allowing for these sets of processes, and all future child processes, to be allocated a specific amount of the given resources, e.g., CPU, memory. The Linux subsystems that implement the cgroups are called *resource controllers* (or simply *controllers*). These resource controllers are responsible for scheduling the resource to enforce the cgroup restrictions. A list of available controllers is shown in Table 3.2. The cgroups are arranged into a *hierarchy* that contains the processes in the system, with each task residing in exactly one cgroup. The cgroup mechanism can be used to provide a generic method to support task aggregation (grouping). For example, the grouping of CPUs and memory can be linked to a set of tasks via cpusets, which uses the cgroups subsystem [10].

To simplify the usage of cgroups, the designers created a virtual file system for creating, managing, and removing cgroups. The file system of type `cgroup` can be mounted to make changes and view details about a given cgroup hierarchy [10]. All query and modify operations are done via this `cgroup` file system [10], with each cgroup shown as a separate directory with meta-data contained in files in the directory, e.g., `tasks` list of PIDs in group. Figure 3.1 shows an example taken from [10] that details how create a cgroup named "Charlie" that contains just CPUs 2 and 3 and Memory Node 1, and starts a subshell 'sh' in that cgroup.

Controller	Description
blkio	sets limits on input/output access to and from block devices such as physical drives (disk, solid state, USB, etc.).
cpu	uses the scheduler to provide cgroup tasks access to the CPU. It is mounted together with cpuacct on the same mount.
cpuacct	automatic reports on CPU resources used by tasks in a cgroup. It is mounted together with cpu on the same mount.
cpuset	assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
devices	allows or denies access to devices by tasks in a cgroup.
freezer	suspends or resumes tasks in a cgroup.
memory	sets limits on memory use by tasks in a cgroup, and generates automatic reports on memory resources used by those tasks.
net_cls	tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.
perf_event	allows to monitor cgroups with the `perf` tool.
hugetlb	allows to use virtual memory pages of large sizes, and to enforce resource limits on these pages.

Table 3.2. Available Resource Controllers in Red Hat Enterprise Linux 7 [39].

```
mount -t tmpfs cgroup_root /sys/fs/cgroup
mkdir /sys/fs/cgroup/cpuset
mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
cd /sys/fs/cgroup/cpuset
mkdir Charlie
cd Charlie
/bin/echo 2-3 > cpuset.cpus
/bin/echo 1 > cpuset.mems
/bin/echo $$ > tasks
sh
# The subshell 'sh' is now running in cgroup Charlie
# The next line should display '/Charlie'
cat /proc/self/cgroup
```

Figure 3.1. Example showing how to create a cgroup ("Charlie") containing CPUs 2 and 3, and Memory Node 1, and starting a process ('sh') in the new cgroup. (Example taken from [10].)

3.1.3 Linux-VServer

Linux-VServer [31] is a patch to the Linux kernel that allows for VEs to be created and isolated from each other as well as the host system. The patch specifically modifies the process, network, and file system data structures of the kernel.

With respect to processes, each process is given a unique PID regardless of VE. This means there is a global PID space. In order to isolate VEs, a VE is given a range of possible PIDs and any process with a PID within that range is considered within that VE.

The file systems of each VE are isolated through the use of `chroot`. Chroot changes the root directory (e.g., “/”) for the execution context to the directory associated with the VE. When this occurs, the user of the VE should not be able to locate any file not associated with the VE unless there is a shared file system between VEs (e.g., NFS).

There is little isolation with respect to the network subsystems of the kernel. More clearly, there is no performance isolation between VEs, but packets are tagged with a VE ID in order to determine the delivery location of the packet.

The scheduling of a VE to the CPU is completed with the combination of two approaches. The first approach is the use of the default Linux scheduler. However, simply using the Linux scheduler could result in an unfair scheduling of specific VEs. To remedy this, the Linux-VServer uses a token bucket filter (TBF) in order to schedule VEs. Each VE is assigned a TBF. While the TBF is not full, every process associated with the VE is removed from the scheduler’s list of “runable” processes (i.e., run queue). When the TBF is full, a process from the VE is scheduled and the TBF is emptied accordingly.

Unfortunately, due to the implementation of Linux-VServer (i.e. a global PID space), a VE executing in this environment cannot be checkpointed or migrated. This is because it is impossible to guarantee the same PID space originally assigned to a VE to be available on restart. As we will be dealing with clusters and scientific computing, there will be failures and the inability to overcome failures reduces the desirability of this virtualization system. Another potential detraction of Linux-VServers is that the support is through external patches, i.e., the code is not in the main line of the Linux kernel. Therefore, the integration and deployment of Linux-VServers with existing environments may be less streamlined.

3.1.4 OpenVZ

OpenVZ [38] is a container-based virtualization solution. The system is made possible by creating a custom kernel that supports the underlying functionality including process and resource isolation. The custom kernel commonly used is a modification of a Linux kernel. It is possible to make use of a unmodified Linux kernel of version 3.x or higher, but this will result in limited functionality.

OpenVZ leverages the namespace functionality resident in the Linux kernel in order to provide process, file system, I/O, and user isolation. The isolation is provided on a per VE basis. This allows for the safe execution of multiple VEs per system.

Resource isolation with respect to the CPU and disk resources are accomplished using two-level schedulers. For the CPU, beancounters are used to represent a VE executing on the system. These beancounters keep track of the VEs CPU usage over time and allow the scheduler to fairly select a schedulable VE. At the second level of scheduling, the default Linux scheduler is used to select a process to execute from within the VE. Similarly, beancounters are used to keep track of a VEs disk I/O usage as well. The first level scheduler for disk I/O examines the beancounters for each VE and fairly selects a VE. After the VE has been chosen, the default Linux disk I/O scheduler will be used as the second level scheduler.

OpenVZ provides several options for the use of the networking systems. There are route-based, bridge-based, and real-based networking options that may be assigned to a specific VE. The route-based approach is the routing of Layer 3 packets (e.g. TCP) to a VE. The bridge-based approach routes Layer 2 packets (e.g. Ethernet) to a VE. Finally, the real-based approach is simply the assignment of a NIC to a VE.

Checkpoint/restart and container migration is supported for OpenVZ. Checkpoint/restart may be accomplished using CRIU [13], which is able to leverage existing Linux kernel functionality in order to save the container’s state to disk. The same mechanism is used to provide migration functionality, however, this is a stop-and-copy approach rather than a live migration approach.

3.1.5 LXC

Linux containers (LXC) [32] is a collection of user-level tools that assist in the creation, management, and termination of containers. The tools leverage the feature-set presented by the Linux kernel including namespaces and cgroups. By leveraging these features, it is possible for LXC to remove the burden of knowledge with respect to virtual environment creation from the user. Instead, customization of the environment is eased and can be the primary goal of the user.

Because LXC leverages features present in Linux, the scheduling of CPU resources is provided by the default scheduler and the use of cgroups. Likewise, user, filesystem, and process isolation is provided through the use of namespaces.

3.1.6 Docker

Docker is a user-level tool to support in the creation, management, and termination of containers in Linux environments. This tool may leverage either the underlying Linux container-based features or LXC in order to easily create and maintain containers. As an alternative to LXC containers can be run through the libcontainer execution driver, which is aimed at standardizing the API that programmers use to create and manage containers. Docker has switched to make libcontainer the default execution driver in Docker, so it is likely that future development efforts from within Docker will be focused on libcontainer rather than LXC. Regardless of which is leveraged, Docker provides both resource isolation and resource management through Linux's namespaces and cgroups respectively.

The distinguishing features of Docker from LXC are higher-level features such as an image-based filesystem capable of support snapshotting, and an API that can be used locally by the docker daemon or remotely, if the socket is exported, to control the management of containers. LXC exposes many very granular configuration options, whereas Docker's configuration is much more limited and contained within a standardized "Dockerfile" format (see Appendix A.1 for Dockerfile examples). The image-based management of Docker images greatly simplifies the distribution of applications, where they can be stored in a repository, from which a user can pull the image, run the container, save the image, and push it back to the repository. For our evaluation, we set up a private Docker repository on a test bed node, which is an alternative to using *http://hub.docker.com*.

3.2 System level virtualization

A hypervisor based approach to virtualization allows for running multiple OS kernels, which run in the virtual machine. The following subsections describe the Xen (type-I) and KVM (type-II) hypervisor-based virtualization platforms.

3.2.1 Xen

The Xen hypervisor [5] is a commonly used hypervisor in Enterprise and Cloud environments. The reason for this is due to its free and open-source nature as well as the use of paravirtualization.

Paravirtualization is the modification of both the host OS and guest OS in order to make use of hypercalls from the guest to the host. Hypercalls are akin to system calls in both usage and implementation. For the Xen implementation, a hypercall table is used containing function pointers to the various functions. These functions are meant to perform some privileged operation on behalf of a guest without the requirement of a trap-and-emulate architecture commonly found in full virtualization environments.

Interrupts in the guest are delivered using an event-based interrupt delivery system in Xen. Upon interrupt delivery, the guest makes use of the corresponding interrupt service routine specified by the guest. During the boot process, the guest registers the interrupt descriptor table (IDT) and, thus, each interrupt service routine with Xen. Xen validates each routine before allowing it to handle interrupts. The majority of faults cause Xen to rewrite the extended stack frame prior to redirecting execution to the guest. An exception to this rule is system call exceptions as these are the most common interrupt. After validation, these exceptions are handled directly by the guest without redirection.

With respect to memory management, a guest OS is allocated a specified amount of RAM by the user during VM creation. As the guest boots, each page used is registered with Xen after it is initialized. At this point, the guest will relinquish write privileges to Xen and only have read privileges. Any update will be performed by Xen via a hypercall. This allows Xen to provide verification of page updates prior to them actually occurring.

Xen schedules VMs using its credit scheduler. With this scheduler, all VMs are given a certain amount of credits that are debited each time the VM is scheduled. Debits occur periodically every 10 milliseconds the guest is allowed to run.

The credit scheduler uses two states to describe the schedulability of a VM. At any given point, a VM is either in the UNDER state or the OVER state. The UNDER state means the VM still has credits to use and the OVER state is for VMs who have used all of their credits. When scheduling occurs, a VM in the UNDER state will be chosen first unless none are runnable. In this case, a VM from the OVER state will be chosen to execute.

3.2.2 KVM

The kernel-based virtual machine (KVM) [21, 30] is a hypervisor, which extends the Linux kernel. This is often implemented as a loadable kernel module (LKM) but may also reside in the kernel directly. The extension provides support for modern processor extensions for virtualization known as Intel VT-x [45] and AMD-v [1].

KVM operates in conjunction with supporting user-level tools found within QEMU [6]. QEMU is responsible for multiple tasks including allocating the memory associated with a guest, emulating the guest devices, and performing redirection of execution back to the hypervisor during execution. Any guest that is created by the user will have the memory for the guest allocated using `malloc` by QEMU and an `ioctl` is used to inform KVM of the initial address space that may be associated with the VM. Because `malloc` is used, KVM's VMs do not use the amount of assigned memory until each page is touched. Each emulated device is handled in userspace by QEMU after receiving notification from KVM that work is pending. The majority of execution by QEMU is within a loop that handles the pending I/O, as noted earlier, and will return execution to KVM at the end of the loop.

Emulating each device adds a significant amount of overhead due to the VM exits caused by the I/O operations from the executing VM. Because of this reason, Rusty Russell developed virtio [40]. Virtio is a standard for PCI device as well as block device paravirtualization.

While virtio is simply a standardized interface, it requires the usage of hypercalls between the host and the guest. Hypercalls are similar to system calls in implementation and allows for a layer of isolation to be removed in order to reduce the amount of VM exits and, thus, improve performance. Currently, KVM supports five hypercalls, of which only four are active.

With virtio, there is a frontend and backend driver. The frontend driver exists within the VM and communicates via hypercalls with the backend driver found within the host. In more detail, the steps for the KVM virtio frontend/backend communication between guest/host are:

Mechanism	Linux-VServer	OpenVZ	LXC	Docker	Xen	KVM
Namespaces	No	Yes	Yes	Yes	No	No
Cgroups	No	Yes	Yes	Yes	No	No
SELinux/sVirt	No	No	Yes	Yes	Yes	Yes
Hypervisor	No	No	No	No	Yes	Yes

Table 3.3. This table shows the relationship between the security/isolation mechanisms and the virtualization solutions (i.e., which mechanisms are present in which solutions).

1. A guest needs to perform an operation on the device.
2. The function corresponding to the operation is called by the guest on the frontend device.
3. A hypercall is issued between the frontend device and the backend device.
4. The backend device sends the operation to the hardware device and returns the result to the frontend device.

3.3 Virtualization and Security Mechanisms

In this section, we present relevant security mechanisms. These security mechanisms and the isolation mechanisms from Sections 3.1 & 3.2 are summarized in conjunction with relevant virtualization solutions in Table 3.3.

3.3.1 sVirt

The Secure Virtualization (*sVirt*) project extends the generic virtualization interface *libvirt* [7] to include a pluggable security framework [36]. sVirt can be used to put a "security boundary around each virtual machine" [37, Ch.15]. VM or VE processes and disk images are labelled by sVirt so that the kernel can enforce a MAC policy. The initial implementation used SELinux for the labeling and policy enforcement and addressed the threat of a guest that escapes the virtualization mechanism and and then use the host as a platform for attacks on other guests or escalation attacks on the host itself. As of libvirt 0.7.2, there is also support for using *AppArmor* with sVirt to restrict virtual machines [2].

3.3.2 SELinux

SELinux [41] is an implementation of the Flask [42] architecture for the Linux kernel. The Flask architecture was the result of the NSA's and Secure Computing Corporation's (SCC) research to develop a strong, flexible mandatory access control mechanism being transferred to Utah University's Fluke OS. While being implemented for Fluke, the mechanism was enhanced becoming the Flask architecture.

The Flask architecture is comprised of two components: (i) the security server and (ii) the access vector. The *security server* contains the security policy for the system. A security policy is a list of possible subjects and objects. Each subject is a user or role, while everything else is considered an object. With respect to kernel-space, the kernel subsystems are considered object managers. The *access vector* is simply a bitmap with the results obtained by the security server whenever access to a file or device is requested by a process. By storing the results of the security server in the access vector, it is possible to provide mandatory access control with little overhead.

Initially, the development of SELinux was completed as a series of patches to the Linux kernel that provide the services found within the Flask architecture. These patches focus on providing security labels for the various resources controlled by the kernel and the users that may use the system.

3.3.3 AppArmor

The AppArmor security project [4] is derived from the SubDomain project that dates back to 1998/1999 [3], and was rebranded as AppArmor after Novell acquired the work in 2005 [3]. The code extends the Linux kernel to support mandatory access controls (MAC). In 2009 Canonical took over maintenance and development of AppArmor and the core functionality was accepted into the main Linux source in kernel version 2.6.36 [3]. AppArmor uses the Linux Security Module (LSM) interface [11].

AppArmor places restrictions on resources that individual applications can access, which defines the "AppArmor policy" for the program. These controls include access to files, Linux capabilities, network resources and resource limits (rlimits) [4]. The program profiles are path-based. The system is intented to have a lower learning curve than some other security tools. This is in part due to a "learning mode" where policy offenses are logged to help identify the behavior of the program [4]. These learned elements can then be added to the restrictions ("enforced mode") or ignored depending on the security objectives. The intent is to reduce the overhead in developing the security policies for a platform. The various releases and re-packaged versions of AppArmor also include additional policy defaults for standard services, e.g., *ntpd*.

AppArmor is available on many modern Linux distributions, e.g., Debian, openSUSE, Ubuntu. Note, there does not appear to be direct support for AppArmor in the lastest Red Hat release (RHEL7) but the RPMS from openSUSE may be usable. AppArmor has also been integrated with libvirt [2] to provide another security backend for the sVirt framework.

3.3.4 Capabilities

Linux capabilities were introduced in version 2.2 as a mechanism for dividing up the privileges of the root user into distinct units [9]. As of Linux 3.17 the kernel has 37 such units. A thread posseses capability bounding sets which are subsets of the 37 capabilities, one is the effective set, which is used for permission checking by the kernel. Particular capabilities can be individually added or dropped using the *capset()* syscall. For example, a process just needing to modify the kernel's logging behavior (e.g. clear the ring buffer), could have all other capabilities dropped except for CAP_SYSLOG. This is an example of a narrowly-scoped capability that can be granted with a low likelihood of allowing that process to escalate to full root privileges. However, another capability CAP_SYS_ADMIN accounts for over 30% of all uses of capabilities within the 3.2 kernel [9]. The implication is that CAP_SYS_ADMIN has become the catchall for privileged operations in the kernel and due to legitimate privilege escalation vectors, it is no better at limiting the scope of privilege than the full set of capabilities. Some other examples that can lead to privilege escalation when given to a process unconstrained by kernel namespaces are CAP_SYS_MODULE, CAP_SYS_RAWIO, CAP_SYS_PTRACE, CAP_CHOWN [44] The first one would allow arbitrary code to be loaded as a kernel module, and the second one would allow processes to directly control system devices. Interestingly, only the first two are removed from capability bounding set granted to a Docker container by default. Namespace and chroot isolation mechanisms limit the attack surface and in the last two, the isolation prevents specific root escalation vectors. The capability CAP_SYS_PTRACE allows a process to control the execution of another through the *ptrace()* syscall, but when constrained to a *pid* namespace, the processes which can be traced are very few. Likewise, from within a chroot environment, the CAP_CHOWN capability (as root) allows the files to have ownership bits changed within the chroot but sensitive files like */etc/passwd* on the host are not accessible. However, in the last example, additional isolation techniques, such *user* namespaces are needed to prevent a chroot breakout. So even though capabilities distinguish units of root privileges, dropping capabilities must be combined with other isolation techniques to prevent a process from expanding its effective capabilities beyond what was granted.

3.4 Management Platforms

This section primarily addresses the “practicality factor” of the virtualization technologies evaluated in this work. As virtualization technologies enable the portability and lifecycle management (save/start/stop) of user-customizable secure computing environments, tools that operate at a higher level of abstrction become necessary to facilitate rapid deployment and management of resources. Should it be desirable, they allow for compute resources to be instantiated without privilege on the physical hardware they are deployed on. We believe OpenStack may be a string option for managing both VMs and VEs. Since OS-level virtualization doesn’t necessarily involve a hypervisor, an alternative would be to manage VE hosts through standalone configuration management tools such as Puppet. Also entering into the VE management picture is a new project called LXD. The following section is a brief overview of the management platforms that we are evaluating for improving the “practicality factor” of the virtualization mechanisms employed to meet our goal of a customizable secure compute platform.

3.4.1 OpenStack

OpenStack is an open-source cloud framework primarily aimed at deployed private cloud platforms. Numerous sub-projects are each responsive for providing services to the OpenStack cloud. For example the *neutron* project provides the networking services, where the *nova* project provides the computing resources. Each project has an API for admins or tenants to interact with. Only a subset of the OpenStack projects apply to the use case in this work and each component can be configured to meet specific customer demands. For instance, *neutron* has several plugins for providing different types of network services (e.g. *vlan*, *flat*, *gre*). We are interested in using the plugins that facilitate isolation through various mechanisms, including VLANs, and SDN plugins that allow the configuration of network devices to be automated.

Our use case of a multi-tenant cloud, providing system-level or OS-level customizable computing platforms, while supporting separate platform and infrastructure admins fits well within the OpenStack framework. Over the coming months we will be exploring the use of OpenStack with regard to how it works with the secure customizable compute platforms that are the subject of this repot.

OpenStack deployments typically allow users to log into a web-based dashboard to view and manage tenant-specific resources, or alternatively allow them to authenticate with API’s providing similar functionality. From the dashboard, a user can launch an instance selecting from a list of available images, and upon successfully deployment, view the IP address that can be used to SSH to the deployed VE or VM. Resource limits such as the number of instances or CPUs, or GB of memory that can be used are specified per-tenant. In summary OpenStack provides significant ease of use benefits to both types of administrators, but it also expands the functionality exposed to tenants who are unprivileged on the actual hosts providing either VE or VM compute resources.

3.4.2 Puppet

As a very flexible and full-featured configuration management system, Puppet [27] can be a useful tool for an automated infrastructure deployment. In the test bed, Puppet was used to initially configure the RHEL 7 VM or VE hosts. In the typical deployment scenario there were one-time tasks first run by the infrastructure admin, before the system is accessible by other users. After the initial run, Puppet can be run via an agent process in the bacground to periodically sync configurations. The agent ensured consistency across the various machines in a deployment. Beyond use by infrastructure admins, Puppet could also be employed to configure tenant VMs or VEs based on templates. An option for further customization by tenants could be to set up a separate version-controlled repository per-tenant. When Puppet runs on the VMs or VEs, it would use the configuration parameters and templates from the committed repository. This achieves a high standard of consistency and reproducibility where a VM or VE that fails can easily be recreated from the data in configuration management. This model works well for infrastructure admins who have a large number of systems to manage and complex configuration requirements. However, the check-in and Puppet agent model can be a burden for tenants to keep up with, where configured environments might only be serving a temporary purpose.

An alternative to periodic runs of configuration management is to leverage virtualization feautures such as snapshotting to save the changes to the base image to persistent storage. This is a common feature for system-level virtualization technologies, but further testing is needed with OS-level virtualization to explore tools such as CRUI [13] when used with LXC. This tool would provide the means for live migration of containers between physical hosts. A hybrid approach to this problem is likely where Puppet is used for infrastructure administration, while tenant configuration may rely on other higher level tools.

3.4.3 LXD

LXD [33] was recently announced in November 2014 as a project that may fill the gap for container-based virtualization and where cloud management tools sit today. It promises a management platform for unprivileged containers making use of user namespaces, AppArmor, and seccomp by default. Other feautres typically only found in VM hypervisors are support for live-migration and snapshotting. The latter of which comes from LXC's prior integration with the CRIU tool. Additional features promised are improved networking options beyond NAT or bridging modes and REST API-based image storage. The development effort is backed by Canonical and the first project code has been posted but it is evident that signficant development remains before this tool is operationally ready.

Regarding future OpenStack integration for LXD, a driver for Nova called *nova-compute-lxd* will be developed to interface between the Nova API and the LXD API. It will bear a strong resemblence to an existing project *nova-compute-flex* that can create VEs with `user` namespaces on an OpenStack cloud today. Our evaluation efforts in the short term in this area will be using *nova-compute-flex* while we wait for LXD to mature.

Chapter 4

Evaluation

4.1 User namespaces

From our review of prevalent virtualization technologies, which we have discussed in Chapter 3 above, we observed that `user` namespaces have a unique benefit with respect to VEs and multi-tenant shared filesystems. The existence of a kernel-enforced isolation mechanism between the user mappings on the host and guest meant that the root user could be prevented from gaining access to certain areas of the filesystem. Since a shared filesystem client, typically sits in kernel-space where it handles VFS calls it implicitly trusts the supplied UID and GID, as that of an authenticated user. However, the root user local to that machine is capable of supplying an UID or GID with a POSIX system calls (e.g. *read()*, *write()*, *stat()*), so there are no mechanisms preventing root from accessing any users data on the filesystem. Root-squash techniques limit the power of the actual root user, but they are powerless to distinguish between a real user and root posing as that user.

However, with the introduction of the `user` namespace abstraction, root in the VE is a new level in the privilege hierarchy where the filesystem client can be protected behind UID and GID mapping where root in the container is just a normal user on the host. With respect to the security of a shared filesystem, the consequence of allowing the end-user to have root credentials within the container is no different than granting them an unprivileged user account on the filesystem. A container root user can be restricted to a separate view of the shared filesystem as defined by POSIX file and directory permissions (also referred to as a filesystem namespace).

4.1.1 Shared-storage use case

Building on the technical feasibility of securely isolating the filesystem namespace that a container may access, we are evaluating the use of customizable VEs (containers) accessing these isolated segments of two parallel distributed filesystes, Lustre and GPFS. Security is achieved through a combination of filesystem namespace isolation, and existing POSIX permission-based access controls. For a proof-of-concept demonstration, we used a single node in our test bed infrastructure running Red Hat Linux version 7 for the host OS, and a LXC container as the VE guest, also running Red Hat Linux 7. Red Hat disables `user` namespace support by default, so a 3.13.11 kernel was built with `user` namespaces enabled and additional upstream patches for supporting unprivileged `user` namespaces. Shadow-utils 4.2 was used instead of the Red Hat-provided version to include new features relating to `user` namespaces. Specifically, 4.2 enabled the host's root user to control of the allowed UID/GID mappings with usermod utility or the */etc/subuid* and */etc/subgid* files. While the kernel and shadow-utils version were not the Red Hat-provided versions, there is precedent in other distributions, namely Ubuntu to support these features out of the box. A last important requirement for this proof-of-concept that is relevant in a production deployment was centralized LDAP for consistent UIDs between the RHEL7 host OS and Lustre servers. LDAP is not a requirement in the VE guest for shared storage isolation.

On the Lustre side, the server was KVM-virtualized for rapid deployment running Lustre 2.5 on Red Hat Linux 6. Work is currently underway to migrate the filesystem to dedicated hardware and storage controllers in the testbed to facilitate performance evaluations. The RHEL7 host ran a Lustre 2.6 client, which is installed as a kernel module and activated with the mount command. The mount command below run as root mounts the filesystem at */lustre* by initiated a TCP connection to the Lustre MGS server. This environment uses the TCP lustre networking driver instead of the InfiniBand driver.

```
mount -t lustre 192.168.122.5@tcp:/lustre /lustre/
```

When */lustre* is viewed on the host by an unprivileged user alice, the directory ownership is dictated by the LDAP server. Three users on the host: alice, bob, and root have three directories each, with user, group, and world writable bits set.

```
[alice@or-c45 lustre]$ ls -l
total 36
drwxrwx--- 3 alice  users 4096 Oct 14 09:23 alice-group
drwx------ 3 alice  users 4096 Oct 14 09:24 alice-user
drwxrwxrwx 3 alice  users 4096 Oct 14 08:27 alice-world
drwxrwx--- 3 bob  users 4096 Oct 14 08:28 bob-group
drwx------ 2 bob  users 4096 Oct 14 08:23 bob-user
drwxrwxrwx 3 bob  users 4096 Oct 14 08:28 bob-world
drwxrwx--- 2 root root  4096 Oct 14 08:15 root-group
drwx------ 2 root root  4096 Oct 14 08:15 root-user
drwxrwxrwx 3 root root  4096 Oct 14 08:27 root-world
```

This is the typical case where a cluster compute node has the filesystem mounted where any user can access files as the ownership and permissions settings allow. Both alice and bob can access directories owned by themselves, where access to the other directories depends on whether the group r/w/x bits are set and whether they are a group owning the directory. For this example, note that alice can access root-world, bob-world, and bob-group, but not root-user, root-group, or bob-user. Next we will expose this Lustre filesystem through to an LXC container by bind-mounting it to a path that the container has access to.

Since the container runs in a chroot inside the host's global directory hierarchy, the host han access the container's filesystem. As such it can perform a mount command on behalf of the container, where the mount point is relative to the host's directory structure. The command below will cause */lustre* on the host to be bind-mounted in the container at emph/lustre on startup.

```
lxc.mount.entry=/lustre \
                /home/alice/.local/share/lxc/lxc_lustre/rootfs/lustre \
                none defaults,bind 0 0
```

The new `user` namespace will attempt to set its mappings on startup, but the host kernel will consult the */etc/subuid* and */etc/subgid* files to see that the requested mappings are allowed. Since those files are on the host filesystem and managed by the host root user, they are trusted. In this demonstration, we want to allow alice to map her own UID 6000. The mapping also allows 65533 continguos other UIDs starting at 100000 on the host. Since UID 100000 and above on the host are unprivileged, all of the allowed mappings within the container will be unprivileged as well. Bob's UID is excluded from this list, so his UID cannot be mapped into the container. */etc/subuid*:

```
alice:100000:65533
alice:6000:1
```

The file */etc/sugid* is configured in an analogous way, except the users group has GID 100:

```
alice:100000:65533
alice:100:1
```

LXC needs to be aware of the allowed mapping as well. This makes up what the container will attempt to write to */proc/CONTAINER_PID/uid_map* and */proc/CONTAINER_PID/gid_map* on startup. The kernel consults /etc/subuid and /etc/subgid and the write will succeed since the mappings were defined above.

```
lxc.id_map = u 0 6000 1
lxc.id_map = g 0 100 1
lxc.id_map = u 1 100000 65534
lxc.id_map = g 1 100000 65534
```

To allow a specific user to modify the cgroups for the container, the following scriptable commands were needed:

```
for controller in /sys/fs/cgroup/*; do
  sudo mkdir -p $controller/$USER/lxc
  sudo chown -R $USER $controller/$USER
  echo $$ > $controller/$USER/lxc/tasks
done
```

After starting the container with *lxc-start –name lxc_lustre* and and gaining a prompt either with *lxc-attach –name lxc_lustre /bin/bash* or ssh, the expected mappings are visible in */proc/CONTAINER_PID/uid_map* and */proc/CONTAINER_PID/gid_map*:

```
[root@lxc_lustre lustre]# cat /proc/1299/uid_map
         0       6000          1
         1     100000      65533
[root@lxc_lustre lustre]# cat /proc/1299/gid_map
         0        100          1
         1     100000      65533

[root@lxc_lustre lustre]#ls -l
total 36
drwxrwx--- 3 root  root  4096 Oct 14 08:27 alice-group
drwx------ 3 root  root  4096 Oct 14 08:27 alice-user
drwxrwxrwx 3 root  root  4096 Oct 14 08:27 alice-world
drwxrwx--- 2 65534 65534 4096 Oct 14 08:15 root-group
```

```
drwx------ 2 65534 65534 4096 Oct 14 08:15 root-user
drwxrwxrwx 3 65534 65534 4096 Oct 14 08:27 root-world
drwxrwx--- 3 65534 root  4096 Oct 14 08:28 bob-group
drwx------ 2 65534 root  4096 Oct 14 08:23 bob-user
drwxrwxrwx 3 65534 root  4096 Oct 14 08:28 bob-world
```

The output from *ls* confirm that alice's UID of 6000 was mapped to 0 (root) on the host and GID 100 (users) was mapped to the root group. Notice how unmapped UIDs and GIDs become 65534 inside the `user` namespace, which is UIDMAX. While user 65534 can have a name assigned to it (e.g. nfsnobody), it has no real permissions on the system. This prevents alice from being able to access root-user even if she maps 65534 within the container or sets her effective UID to 65534 (this is allowed since she has root privileges within the container).

We next attempt to perform some filesystem operations from within the container to directories on the bind-mount:

```
[root@lxc_lustre lustre]# mkdir root-world/test
[root@lxc_lustre lustre]# ls -l root-world/
total 4
drwxr-xr-x 2 root root 4096 Oct 14 09:24 test
[root@lxc_lustre lustre]#mkdir root-group/test
mkdir: cannot create directory 'root-group/test': Permission denied
[root@lxc_lustre lustre]# ls -l bob-group/
total 4
drwxr-xr-x 2 root root 4096 Oct 14 08:28 test
[root@lxc_lustre lustre]# chown root bob-group/
chown: changing ownership of 'bob-group/': Operation not permitted
```

We can see that alice can create a directory in root-world since it has the 'other' w bit set. However, creating the directory */lustre/root-group/test* is disallowed because even though alice is a member of the group root in the container, she is not a member of root on the host, which is GID 65534 in the container. Also note how the group for bob-user, bob-group, and bob-test is root within the container and alice can read files in bob-group. This means bob can share files with alice even as alice accesses the directories from within the `user` namespace. Attempts to change ownerships of bob's directories fail, because the host kernel will map root within the container to alice's UID of 6000 and the Lustre filesystem will not allow UID 6000 to change directories owned by UID 3000 (bob).

Since the check whether a particular UID is allowed to access or change a file are done on the Lustre server, it must be the case that Lustre is only supplied with UIDs from a trusted source. The host kernel, running the Lustre client kernel module is trusted, but alice's container is not. Since `user` namespaces ensure that the UIDs from the container are mapped to allowed UIDs before being sent to the Lustre client, the Lustre server can trust the supplied UID. In the absence of `user` namespaces, since the host must be trusted, it was not possible to give tenants root access to a customized compute environment with similarly configured shared filesystems.

4.2 HPCCG

4.2.1 Description

We performed a set of tests using the High-Performance Computing Conjugate Gradient (HPCCG) benchmark to establish baseline performance for basic application execution. The tests gather data from execution of the benchmark on the *Native* (host) machine, and when run under *Docker* and *KVM*.

HPCCG was developed by Michael Heroux from Sandia National Laboratories and is included in the Mantevo mini-apps [34]. The code is written in C++ and support serial and parallel (MPI & OpenMP) execution. The benchmark performs an iterative refinement until reaching a solution within a given threshold, or until a maximum number of iterations are performed.

This application is relevant for high-performance computing (HPC) from a few perspectives. Firstly, it provides use case for metrics regarding application memory and compute usage. Also, previous studies have found iterative algorithms to be resilient to some errors [8], possibly at the cost of taking longer to converge on an appropriate value, which is relevant for HPC resilience purposes. Additionally, the HPCCG benchmark has been identified as a more representative metric for current scientific applications and was identified by Heroux and Dongarra as a candidate alternative metric for future Top 500 indexes [22].

4.2.2 Setup

The software configuration for the test used HPCCG v1.0 compiled with the GNU g++ v4.8.2 compiler. The host and guest Linux kernel was the same (version 3.10.0-123.8.1.el7.x86_64), with the host (Native) running Red Hat Enterprise Linux (RHEL) v7.0 and the guests running CentOS v7.0 (free alternative that is binary compatible with RHEL v7.0). The parallel tests used Open MPI version 1.6.4 that is shipped with CentOS v7.

The tests were done on the project's testbed at ORNL[1]. The machines have dual Intel(R) Xeon(R) CPU E5-2650 processors running at 2.8 GHz, with 32 cores and a total of 65 GiB of physical memory per node. The virtualization based tests used KVM v1.5.3-60.el7_0.7 with the VM allocated resources fixed at 31 CPUs and 48G of memory. The VE tests used Docker v0.11.1-22.el7 configured with libcontainer and the VE was allocated resources fixed at 31 CPUs and 48G of memory.

HPCCG accepts three parameters that define the dimensions for the problem, nx, ny, and nz. The serial tests were run using $nx = ny = nz = N$ for increasing values of N up to the max memory available[2]. The parallel tests were run using the same dimensions, which results in a larger overal problem size based on the number processors used, i.e., overall problem size is $nx * ny * (NumProcs * nz)$ [34]. The parallel tests used two nodes ($NumProcs = 2$) with one VM (VE) per node. The tests were run with `max_iterations=150` and `tolerance=0.0`, which results in all tests running to the maximum number of iterations every time. The benchmark was run 20 times, with dimensions of 100, 200, 300, 400, and 430[3], using the loops shown in Figure 4.2 for serial and parallel (MPI) tests, respectively. The output from a serial run of the benchmark is shown in Figure 4.1, with the *Total execution time* and *Total MFLOPS* highlighted in green.

[1]For reference purposes, the tests were done on nodes `or-c46` and `or-c46` of the testbed.

[2]The max memory was the maximum that would be available in the Native/Docker/KVM configurations such that all could have the same max, even though our Native tests could have had a bit more memory than that used in VE/VM configurations.

[3]The selection of $N = 430$ was emperically determined through testing to see what was max value usable with a single processor for given memory resources allocated to VE/VM. In parallel case, the increase problem size exceeds the available memory and results in a max dimension for tests of $N = 300$.

```
bash:$ time -p ./test_HPCCG 100 100 100
Initial Residual = 2647.23
Iteration = 15   Residual = 35.0277
        …<cut>…
Iteration = 149   Residual = 7.9949e-21
Mini-Application Name: hpccg
Mini-Application Version: 1.0
Parallelism:
  MPI not enabled:
  OpenMP not enabled:
Dimensions:
  nx: 100
  ny: 100
  nz: 100
Number of iterations: : 149
Final residual: : 7.9949e-21
```

```
********** Performance Summary (times in sec)
***********:
Time Summary:
  Total   : 6.28416
  DDOT    : 0.366029
  WAXPBY  : 0.56881
  SPARSEMV: 5.34828
FLOPS Summary:
  Total   : 9.536e+09
  DDOT    : 5.96e+08
  WAXPBY  : 8.94e+08
  SPARSEMV: 8.046e+09
MFLOPS Summary:
  Total   : 1517.47
  DDOT    : 1628.29
  WAXPBY  : 1571.7
  SPARSEMV: 1504.41
real 6.45
user 6.37
sys 0.08
```

Figure 4.1. Example output from HPCCG benchmark.

```
  # Serial test
for count in {1..20} ; do
   for dim in 100 200 300 400 430 ; do
      time -p ./test_HPCCG $dim $dim $dim
   done
done

  # Parallel test
for count in {1..20} ; do
   for dim in 100 200 300 ; do
      time -p mpirun -np $numproc --hostfile hosts \
                ./test_HPCCG.mpi $dim $dim $dim
   done
done
```

Figure 4.2. Example showing how the HPCCG serial and parallel (MPI) tests were run.

The startup for the VM based test with KVM is shown in Figure 4.3. The startup for the VE based tests with Docker are shown in Figure 4.4. In the serial case, all tests were run on a single host inside a single VE. In the parallel test case, a "master" VE was started and is where the `mpirun` executes and connects to the "slave" VE(s). The MPI process launch uses SSH to start the remote processes in the "slave" VE(s). Note, the master and slave VE(s) run on separate hosts and are connected via the 10Gig network.

Notice:
The current data was run using 'sudo' for KVM and Docker bootstrap, but we plan to run as a non-privledged user in future to determine if there is any change in performance.

```
  # Edit 48GiB, 31CPUs
sudo virsh edit centos7kvm
  # Start VM
sudo virsh start centos7kvm
  # Login to vm an run benchmark test
ssh centos@<vm_ip_addr>
```

Figure 4.3. Example showing the commands used for KVM/libvirt VM startup.

```
 # -- Single VE for Serial Tests --
 # Limit to 48GiB, not explicitly restricting Num cpus
 # Run benchmark test from shell in container
sudo docker run -m 48g -t -i naughtont3/centos7cxx /bin/bash


 # -- (Part-1) MASTER VE for Parallel Tests --
 # Limit to 48GiB, not explicitly restricting Num cpus
 # Run benchmark test from shell in container
sudo docker run -m 48g --name master --privileged \
           -ti -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
           or-c46.ornl.gov:5000/blakec/centos7cxx-sshd-mpi /bin/bash

 # -- (Part-2) SLAVE VE for Parallel Tests --
 # Limit to 48GiB, not explicitly restricting Num cpus
 # Run benchmark test from shell in container
sudo docker run -m 48g -d --name slave-1 --privileged \
      -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
      or-c46.ornl.gov:5000/blakec/centos7cxx-sshd-mpi
```

Figure 4.4. Example showing the commands used for Docker VE startup. The serial test runs in a single VE, while the parallel version run across two VEs (master/slave).

4.2.3 Discussion & Observations

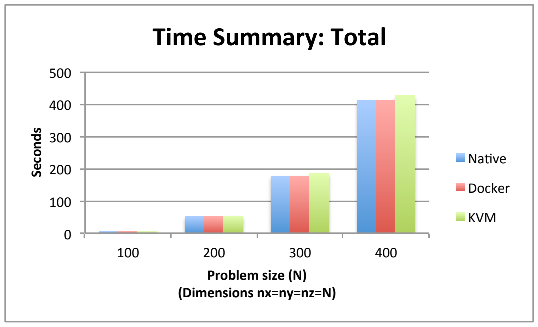
4.2.3.1 HPCCG (serial & parallel)

All results in Figure 4.5 are averaged over 20 runs of HPCCG (serial), e.g., `test_HPCCG nx ny nz`. These results are consistent with previous studies that reported roughly 2-4% overheads in hypervisor-based virtualization environments. The HPCCG (serial) application execution time & MFLOPS shown in Figure 4.5, with details in Tables 4.2 & 4.3, reflect this moderate overhead for the VM case and show near-native performance for the VE case.

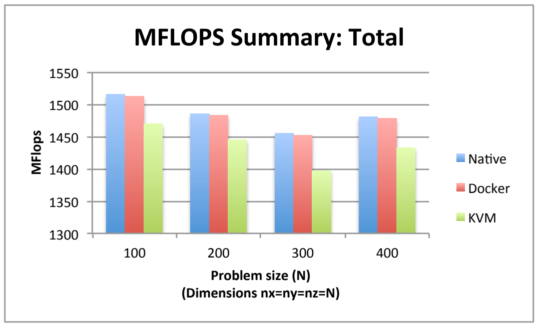
As shown in Tables 4.4 & 4.5, the serial tests had more consistent MFLOPS performance (except in 1 instance) with Docker runs of HPCCG (serial) than with Native runs of HPCCG (serial), i.e., lower standard deviation over 20 runs. (It is currently unclear why this was the case.) However, the actual Docker vs. Native values were almost the same, with Native achieving slightly better performance (lower Time and higher MFLOPS).

Table 4.5 also shows that over 20 runs the standard deviation in KVM based execution of HPCCG (serial) was very high ($\sigma = 15$ to $\sigma = 30$). Further testing will be needed to determine the cause of this fluctuation but it may be due to a lack of resource pinning when the benchmark was run. Overall, the Time tests with HPCCG (serial) showed very consistent values for application runtime over the 20 runs, with the exception of two instances with KVM (kvm-300 and kvm-400).

While these baselines were for single node (HPCCG serial mode), they provide a basis for future

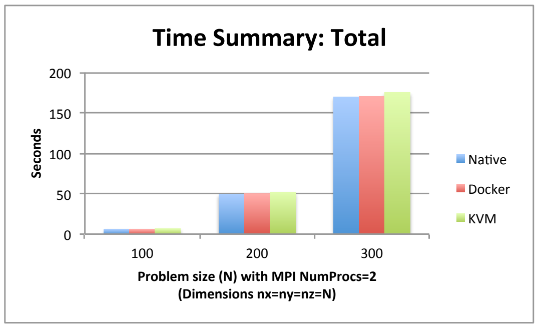


(a) HPCCG (serial) Time in seconds

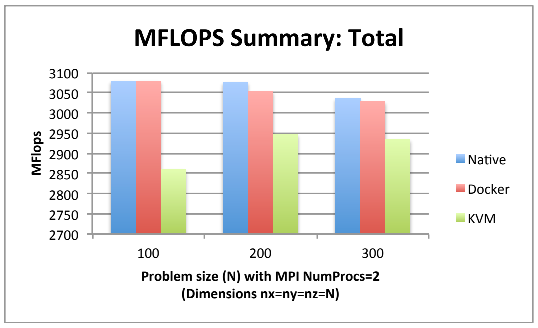


(b) HPCCG (serial) MFLOPS

Figure 4.5. HPCCG (serial) under Native, Docker and KVM at different problem sizes N.



(a) HPCCG (parallel) Time in seconds



(b) HPCCG (parallel) MFLOPS

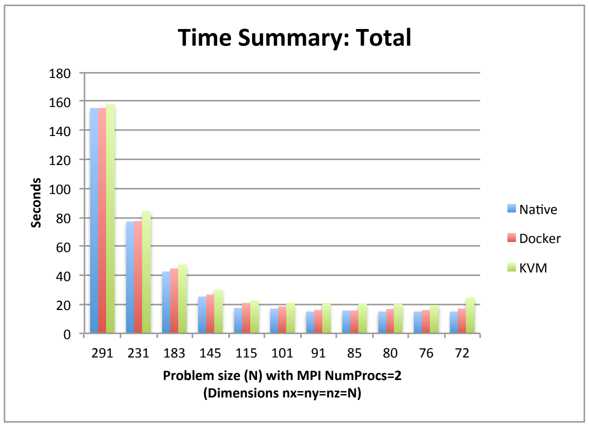
Figure 4.6. HPCCG (parallel) under Native, Docker and KVM at different problem sizes N.

comparisons later in the project. The near native performance of Docker and fast launch times make it a very interesting candidate for further tests with workloads that do not require multiple kernels. Additionally, the tools for Docker launch and execution environment customization show great promise. The integration of more advanced capabilities, e.g., `user` namespace isolation, will also enhance the viability of this approach to virtualization.

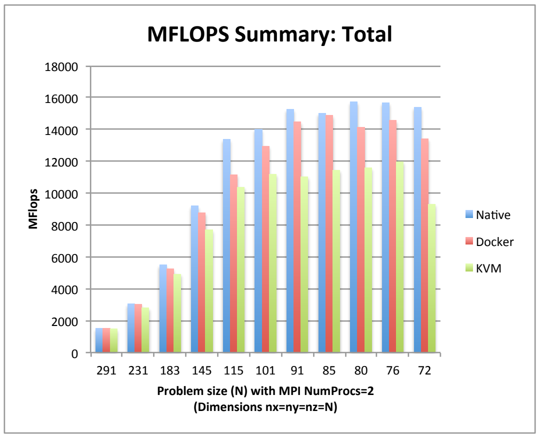
We also repeated these tests same tests with a parallel build of HPCCG, which used MPI and two compute nodes. These HPCCG (parallel) tests were using just two ranks, each on separate hosts/VE/VM. The HPCCG problem size was varied as with HPCCG (serial) tests up to the maximum available memory. The parallel version factors in the number of processors (ranks) to scale the problem up accordingly and therefore the $N = 400$ case exceeded the available memory for the HPCCG (parallel) tests. The results for Time and MFlops are shown in Figure 4.6.

4.2.3.2 HPCCG MPI scale-up

In addition to repeating MPI based runs to match the serial HPCCG test we also ran some very small scale-up tests with HPCCG. While the testbed is still being setup, we had two nodes available so did the scale-up based on the number of cores-per-node (32). The test includes more variation in the number of ranks used and illustrates increased number of ranks per node with roughly fixed problem size. The results from these tests are shown in Figure 4.7, which show the average value over 5 runs at each $NumProc$.



(a) Scale-up test with HPCCG MPI Time in seconds



(b) Scale-up test with HPCCG MPI MFLOPS

Figure 4.7. MPI scale-up test of HPCCG (parallel) under Native, Docker and KVM with a problem sized for approximately 50% of 48GiB/per node. (See details in Table 4.6)

The values for N try to keep the problem size at roughly 50% of available memory. The VE and VM are allocated 48G of memory and the Native system has 64G, so the Native tests are slightly lower percentage of total memory but it is reasonably close for our purposes. As with the earlier tests, the value of $nx = ny = nz = N$ and the same number of nodes (2) are used with these tests for a total of 64 cores (32 per node). The number of ranks used in the tests ranged between 1 to 64 (`mpirun -np X ...`) to show scale-up as detailed in Table 4.1. The N value was calculated so that $nx * ny * (NumProcs * nz)$ is roughly equivelent to 50% ($PctMem = 50\%$) of the memory ($TtlMem = 48GiB$).

$$MemPerRank = \frac{TtlMem \times PctMem}{NumProcs}$$

$$N = \left\lfloor \sqrt[3]{MemPerRank} \right\rfloor$$

For example, with 64 MPI ranks $NumProcs = 64$, each getting $MemPerRank$ amount of data, we set the value of the dimension parameter for HPCCG to $N = 72$, assuming $TtlMem \approx 48G \rightarrow 49747160$.

$$388649.68 = \frac{49747160 \times 0.50}{64}$$

$$72 = \left\lfloor \sqrt[3]{388649.68} \right\rfloor$$

NumProcs	**MemPerRank**	N **(dim)**
1	24873580.00	291
2	12436790.00	231
4	6218395.00	183
8	3109197.50	145
16	1554598.75	115
24	1036399.17	101
32	777299.38	91
40	621839.50	85
48	518199.58	80
56	444171.07	76
64	388649.69	72

Table 4.1. HPCCG MPI ranks and associated dimension parameter $nx = ny = nz = N$, and amount of per-rank memory for problem sized for approximately 50$\%$ of 48GiB memory per node.

Dimensions	Native	Docker	KVM
100	6.293491	6.304308	6.4892095
200	51.36137	51.43617	52.79015
300	176.93065	177.30215	184.34125
400	412.2705	412.85375	426.21125

Table 4.2. HPCCG (serial) Times in seconds averaged over 20 runs under Native, Docker and KVM at different problem sizes N (Dimensions $N = nx = ny = nz$).

Dimensions	Native	Docker	KVM
100	1515.2195	1512.6185	1469.752
200	1485.33	1483.3495	1445.288
300	1455.2405	1452.1705	1397.405
400	1480.3515	1478.2645	1432.4365

Table 4.3. HPCCG (serial) MFLOPS averaged over 20 runs under Native, Docker and KVM at different problem sizes N (Dimensions $N = nx = ny = nz$).

Dimensions	Native	Docker	KVM
100	0.00931357	0.005463481	0.085735511
200	0.146079607	0.610579775	0.598340949
300	0.751579199	0.31564575	4.333634987
400	0.62727787	0.978753443	8.432111397

Table 4.4. Standard Deviation of HPCCG (serial) Times in seconds averaged over 20 runs under Native, Docker and KVM at different problem sizes N (Dimensions $N = nx = ny = nz$). Large values are highlighted in red.

Dimensions	Native	Docker	KVM
100	2.240769311	1.311653336	18.71753824
200	4.218194924	16.89799536	15.75161232
300	6.168290282	2.584735505	30.945298
400	2.250406455	3.498747859	27.07518737

Table 4.5. Standard Deviation of HPCCG (serial) MFLOPS averaged over 20 runs under Native, Docker and KVM at different problem sizes N (Dimensions $N = nx = ny = nz$). Large values are highlighted in red.

4.3 iperf: TCP Bandwidth

4.3.1 Description

We ran basic network performance measurements using the `iperf` benchmarking / tuning utility [24]. The tests provide data about the *Native* (host) performance and the comparison when running under *Docker* and *KVM*.

4.3.2 Setup

The tests were limited to the 10GigE interface in the testbed. The tests focused on the TCP bandwidth between two nodes in the testbed. The tests used `iperf` version 2.0.5-2 for x86-64. The tests were run as shown below in Figures 4.8 using the standard client/server setup for to the tool. The The host and guest Linux kernel was the same (version 3.10.0-123.8.1.el7.x86_64), with the host (Native) running Red Hat Enterprise Linux (RHEL) v7.0 and the guests running CentOS v7.0. This is the same hardware and software configuration used for the HPCCG tests discussed in Section 4.2 on page 25.

The virtualization based tests used KVM v1.5.3-60.el7_0.7 with the VM allocated resources fixed at 31 CPUs and 48G of memory. The VE tests used Docker v0.11.1-22.el7 configured with libcontainer and the VE was allocated resources fixed at 31 CPUs and 48G of memory. The default TCP window size was used for all tests, which was 95.8 KBytes for the Native and KVM client. The Docker client defaulted to a TCP window size of 22.5 KBytes, with one exceptional case that defaulted to 49.6 KBytes.

```
 # Start server, binding to 10Gig interface
[mpiuser@160d5aae2f91 ~]$ ./iperf_2.0.5-2_amd64 -s -B 10.255.1.10

 # Start client TCP test
[mpiuser@160d5aae2f91 .ssh]$ ./iperf_2.0.5-2_amd64 -c 10.255.1.10
```

Figure 4.8. Example showing the commands for staring the server and client of iperf test.

4.3.3 Discussion & Observations

The tests were each run 10 times and the averages are shown in Table 4.6. The Native tests achieve most of the 10GigE bandwith, and Docker achieved near native performance. The KVM configuration did much worse than Native and Docker, which is an issue we plan to look into further as we proceed with the networking tasks. This initial testing was simply to gain a baseline for a basic bridged networking configuration. One possible point for further testing will be to see how the KVM performance changes if we use a ‘virtio’ interface instead of the ‘e1000’ interfaces. Further details about the KVM configuration used in this round of testing are given in Appendix B.

Platform	Transfer (Gbytes)	TCP Bandwidth (Gbits/sec)
Native	11.5	9.89
Docker	10.88	9.331
KVM	3.086	2.651

Table 4.6. Network Bandwidth for TCP tests with `iperf` between two nodes on the 10GigE interface.

Chapter 5

Vulnerability Assessment

5.1 Introduction

This work consists of many components interconnected via some form of interconnect. These components include secure compute, network, and shared storage. In order to provide isolation for each user, the scope of the secure compute component will be reviewed to assess the existing vulnerabilities that are present.

The components that will be evaluated include the Xen hypervisor, the Kernel-based virtual machine (KVM), Linux containers (LXC), Docker, and a brief examination of the Linux kernel.

5.2 Evaluation

To perform this assessment, we compiled relevant common vulnerabilities and exposures (CVEs) and characterized these vulnerabilities based on the type of exploit and the different regions of the platform that are vulnerable to those attacks.

In order to fully clarify the operating model upon which our assessment is based, we made several assumptions of the environment. The following assumptions were made:

1. *Users are allocated resources on a per node basis.* Specifically, this assumes that any resources allocated on a physical node should be given solely to that user. This is a single tenant environment. By making this assumption, we are restricting the amount of possible attacks by removing denial of service (DoS) attacks against the hosting node.

2. *A user may only access to a VM or VE.* In other words, the user should never be given access to the host directly.

3. *The host and the guest will likely be running some version of the Linux kernel.* The use of OS level virtualization solutions forces this assumption to be true as the same kernel must be used on both the host and guest. However, with the use of system-level virtualization, this assumption may still hold but it is not necessarily true that both the host and guest will use the same kernel.

These assumptions limit the goals of the attackers to three types: (i) privilege escalation of the attacker, (ii) unauthorized access to memory and storage including shared storage, and (iii) arbitrary code execution on the host.

Solution	Rel. CVEs	Privilege Escalation	Unauthorized Access	Arbitrary Code Exec.
Xen	53	43.4%	37.4%	18.9%
KVM	26	34.6%	23.1%	42.3%
LXC	2	50%	50%	0%
Docker	3	66.7%	33.3%	0%

Table 5.1. Virtualization solutions and their corresponding attack vulnerabilities.

Solution	Rel. CVEs	x86 Emu.	Devices	Userspace tools	Hardware	VMM/Kernel
Xen	53	22.6%	22.6%	20.8%	5.7%	28.3%
KVM	26	7.7%	69.2%	0%	0%	23.1%
LXC	2	0%	0%	50%	0%	50%
Docker	3	0%	0%	100%	0%	0%

Table 5.2. Virtualization solutions and their vulnerabilities' targeted region of the system.

A very serious type of attack is for the attacker to gain privileges on the host (i.e. type (i)). By doing this, the attacker can raise their privilege from a normal user to that of a superuser on the system allowing the installation of malicious software that could allow for long term superuser access (i.e. kernel-level rootkit). Additionally, the attacker will have the ability to subvert the network isolation provided by VLANs and may attempt to join other VLANs in order to sniff traffic or perform network-based attacks on other nodes. Shared-storage systems would also be vulnerable if a system becomes compromised as described, because the attacker with root privileges can assume any UID on the system. If mechanisms such as user namespaces, sVirt, 3rd party authentication (e.g. Kerberos), are either not used or subverted, the attacker is free to access all other users' data on the shared file system.

Unauthorized access to memory and storage, (ii), is similar to (i) with respect to the violations of isolation between users and users' data. This is evident with the ability of the attacker to be able to obtain sensitive data in memory or on the secondary storage. However, the largest threat of such an attack is to obtain sensitive information that is stored on the host. With the assumption of a single tenant environment, the threat is lessened but still present.

In (iii), an attacker may execute arbitrary code on the host machine. Allowing the attacker to perform such actions could compromise the integrity and trust of the host.

We have isolated our assessment to three areas: system-level virtualization solutions, OS level solutions, and the host and guest kernels. Within the scope of system-level virtualization, we will be examining both the Xen hypervisor and KVM. For OS level virtualization, LXC and Docker are assessed and the Linux kernel is reviewed as it will likely be used for both the host and guest kernel, though not necessarily the same kernel version for both.

The results of this analysis are summarized in Table 5.1 and Table 5.2.

5.2.1 System-level Virtualization

Within the scope of system-level virtualization, we will be performing a vulnerability assessment on two solutions: (i) Xen and (ii) KVM. For both of these hypervisors, we will characterize the potential attacks based on the region of the system targeted to perform the exploit, the type of exploit (i.e., the three types listed prior), and the operating dependency for the Xen hypervisor (e.g., an attack that is only successful when using full virtualization rather than para-virtualization). The regions of the systems that may be potentially targeted include the emulation of the x86 and AMD64 platforms, any emulated devices available for use by the VM including para-virtualized devices, user-level tools and libraries, the hypervisor and hypervisor related libraries, and the underlying hardware architecture.

5.2.1.1 The Xen Hypervisor

The Xen hypervisor [5] has been available for use from 2003 to present with versions 3.4 and 4.2-4.4 currently supported. The version limitation has restricted our assessment only to the these versions. While this does not include the full lifespan of the Xen hypervisor, it is sufficient to obtain a reasonable, modern assessment.

We have found that there are 124 total CVEs with respect to Xen. However, after limiting these CVEs based on our assumptions, this reduces the amount of valid CVEs to 53. Of these CVEs, almost an identical amount are based on Xen operating in either full virtualization, known as HVM, or para-virtualization with full virtualization required for 28.3% of the CVEs and para-virtualization required for 24.5%. The remaining CVEs had no specific operating dependency.

Privilege escalation provides for 43.4% of the CVEs for the Xen hypervisor. The attacks are present for privilege escalation of both unprivileged and privileged users in the guest environment with the escalation resulting in the potential to escape from the VM and access the host. Unauthorized access to memory or storage accounts for 37.7% of the CVEs suggesting that both privilege escalation and unauthorized access to sensitive information will be the primary attacks used in future zero-day exploits for Xen.

The regions of the systems that are targeted for these CVEs are primarily related to the hypervisor or hypervisor-level tools with 28.3%. Both the platform emulation and emulated devices are next with 22.6% each. User-level tools make up 20.8% of the CVEs and hardware related regions having the smallest amount with 5.7%. These results suggest that the regions of the system that require protection from future exploits are varied and not easily hardened with external mechanisms such as Xen's Xen security module (XSM), which implements a protection mechanism much like SELinux.

It should be noted that the CVEs we examined specific to Xen have all been fixed.

5.2.1.2 KVM

The KVM hypervisor [26] is a relatively new hypervisor in comparison to Xen and takes a more traditional approach to performing full system-level virtualization (i.e. trap-and-emulate style). Because it is only performing full system-level virtualization, the complexity of the hypervisor is less than that of Xen and it is evident with respect to this assessment.

Currently, there are 26 CVEs matching our assumptions related to KVM or the user-level supporting tools known as QEMU, which is used for VM initialization as well as device emulation. Of these CVEs, there was a relatively even distribution of CVEs among the three types of attacks we are focusing on for this work. Arbitrary code execution has the largest proportion of CVEs with 42.3% and privilege escalation accounting for 34.6%. Unauthorized memory access had a total of 23.1% of the CVEs.

There are only three regions of the system that have vulnerabilities. Device emulation and implementation errors within the hypervisor itself consume a combined 92.3% of the CVEs, while x86 and AMD64 emulation consume the rest.

These types of vulnerabilities and regions of the system effected suggest that the simplistic implementation of KVM provides far more benefits with respect to limiting vulnerabilities than Xen's more complex implementation. Additionally, many of the vulnerabilities are specific to device emulation, which is handled, primarily, in userspace on the host. This means an attacker would need to perform additional attacks in order to fully compromise the host.

Like Xen, all of the CVEs listed for KVM have been fixed.

5.2.2 OS level virtualization

With respect to LXC and Docker, both work are inter-related as Docker made use of LXC by default prior to the 0.9 version where Docker moved to libcontainer as the default virtualization solution. However, Docker can still make use of LXC rendering all vulnerabilities specific to LXC also potentially affecting Docker as well.

To evaluate this work, the same categories used in Section 5.2.1 will be applied here. Obviously changes must be made with respect to the regions of the system that may be exploited due to the different techniques involved in providing OS level virtualization as apposed to system-level virtualization. Thus, the categories related to the emulation of the x86 and AMD64 as well as device emulation will be dropped from this evaluation. However, the regions including userspace tools that leverage LXC (e.g., libvirt), hardware related, and the kernel.

5.2.2.1 LXC

With respect to LXC, there are few vulnerabilities as compared to the system-level virtualization approaches. Currently, there are four CVEs specific to LXC, i.e. CVE-2011-4080, CVE-2013-6436, CVE-2013-6441, and CVE-2013-6456. However, due to the single tenant environment, CVE-2013-6436 should not be considered as exploiting this vulnerability will result in a DoS attack. Likewise, CVE-2013-6456 is not a vulnerability within LXC but within Libvirt, which may be used to create VEs. Because the attacker would need access to the host, we are excluding this vulnerability from analysis.

In CVE-2011-4080, there was a logical implementation error with respect to permissions to read the kernel's ring buffer (i.e., `dmesg`). The error dealt with the `dmesg_restrict` system call that, when set to 0 allows unprivileged users to perform `dmesg`. When the system call is set to 1, the user must have the `CAP_SYS_ADMIN` capability set in order to perform `dmesg`. The vulnerability occurs when a unprivileged user becomes root within the container. At this point, the user is able to perform the `dmesg_restrict` system call and set it to 0 allowing the unprivileged view of the system log. This vulnerability has since been fixed.

The CVE-2013-6441 vulnerability is not actually related to the implementation of LXC, but to a template used by LXC to assist in the building of a VE. Templates are simply scripts used to build VEs. This template is the `lxc-sshd.in` template and the issue is during VE creation, the script performs a bind mount of the host's `/sbin/init` executable with r/w permission inside the guest. A malicious user could modify or replace the host's `init` with their own and escalate privilege by creating another guest that uses this template. Oddly, the vulnerability is not fixed in all distributions of Linux, though it only affects LXC versions prior to 1.0.0.beta2 and can be fixed by modifying one line of the template.

5.2.2.2 Docker

There are three vulnerabilities specific to Docker. These vulnerabilities include CVE-2014-3499, CVE-2014-5277, and vulnerability that was not assigned a CVE number.

CVE-2014-3499 is specific to version 1.0.0 of Docker. In this vulnerability, Docker failed to assign the correct permissions to the management sockets allowing them to be read or written by any user. This could allow a user to take control of the Docker service and its privilege. This vulnerability was has since been fixed.

The vulnerability described by CVE-2014-5277 is specific to a fallback from HTTPS to HTTP if an attempt to connect to the Docker registry fails. This presents the possibility for an attacker to force the Docker engine to fallback to HTTP if a man-in-the-middle attack was used. Because HTTP is used rather than HTTPS, unauthorized access to any information sent over the network will occur. In the worst case, authentication information may be leaked due to an exploit of this vulnerability. This vulnerability was fixed in Docker version 1.3.1.

The unassigned vulnerability effects Docker versions 0.11 and prior. In these versions, Docker failed to restrict all kernel capabilities to the guest and instead only restricted a specific set of capabilities. This could allow a malicious guest to walk the host file system by opening inode 2, which always refers to the root file system on the host. This was fixed in version 0.12.

5.2.3 The Linux Kernel

The Linux kernel is present in some form for all of the system-level and OS level virtualization solutions. Currently, the Linux kernel is known to have as many as 1199 CVEs from 1999 to present. Of these CVEs, 33.1% consist of the three types of attacks we have presented above with the majority of these related to the unprivileged access to memory and storage. The Linux kernel also contains a significant attack vector with as many as 339 system calls as of the Linux 3.17 kernel. This is in addition to many more kernel modules and subsystems with interfaces open to userspace.

5.3 Recommendations

From the assessment that was performed, there are conclusions that may be made. First, system-level virtualization solutions are more vulnerable than OS level virtualization solutions. Another conclusion that can be made focuses on the isolation of the majority of vulnerabilities to specific regions of the system. Based on these conclusions recommendations will be made for the possible design of a secure compute environment.

As stated prior, system-level virtualization solutions have many more vulnerabilities than OS level virtualization solutions. Based on the results of Section 5.2.1, this is likely due to the complexity of the implementation necessary to perform safe virtualization of the underlying architecture as well as the multiplexing of hardware through emulated devices. However, many of the vulnerabilities for KVM are due to emulated devices, which reside in userspace. Based on this, a recommendation for this work is to make use of a mechanism such as sVirt if system-level virtualization is used. The use of sVirt may limit the damage from the exploit of these vulnerabilities.

The majority of vulnerabilities for KVM, LXC, and Docker are in specific regions of the system. This is important because future zero-day vulnerabilities will likely be in the same regions. For KVM, the vulnerabilities are primarily found in device emulation. LXC and Docker primarily have vulnerabilities in userspace tools or scripts. The protection of these areas can simplify the protection of the host and maintain the isolation between users. It is recommended that these solutions be used to provide the virtualization layer for this work.

Chapter 6

Conclusion

6.1 Synopsis

The customization of multi-tenant environments is directly influenced by the underlying isolation mechanisms used to limit access and maintain control of the computing environment. In this report we reviewed terminology and relevant technologies, which helps to elucidate the topic of secure compute customization.

A brief review of relevant security classifications is discussed in Section 2.2. This was followed by a review of virtualization classifications in Section 2.3, which detailed the different types of OS-level and system-level virtualization. The main distinction between different virtualization technologies, which is used throughout the report, has to do with the degree of integration with a host kernel. The terms *virtual environment (VE)* and *virtual machine (VM)* are used to distinguish between container-based (single kernel) and hypervisor-based (multiple kernel) virtualization.

In Section 3, we reviewed current operating system protection mechanisms and virtualization technologies that provide the basis for customizable environments. This included various OS-level mechanisms like namespaces, cgroups and LXC/Docker. This was followed by a review of two hypervisor-based solutions Xen (type-I hypervisor) and KVM (type-II hypervisor). This included details about kernel versions when various capabilities were introduced, which provides information about dependencies when choosing different solutions for deployment. Security mechanisms and virtualization were discussed in Section 3.3, which included information on the sVirt framework that enhances the *libvirt* virtualization interface to support a security framework. The currently supported sVirt backends, SELinux and AppArmor are also described. This section finishes with a review of Linux Capabilities in Section 3.3.4.

The evaluations that have been carried out thus far in the project are described in Section 4. The first (Section 4.1) focuses on experiments using the Linux `user` namespace to differentiate guest/host user contexts and provide different access rights accordingly. The next evaluation (Section 4.2) provided a baseline for running a scientific application under native and virtualized settings. This included numbers for the HPCCG benchmark run natively and on Docker and KVM. Finally, in Section 5 we analyzed several current virtualization solutions to assess their vulnerabilities. This included a review of common vulnerabilities and exposures (CVEs) for Xen, KVM, LXC and Docker to gauge their susceptibility to different attacks.

6.2 Observations

We will now briefly discuss observations that were made as a result of this review. The intent is to briefly summarize, or highlight, important points that may be useful for realizing user-customizable secure computing environments.

6.2.1 Vulnerability Assessment

Based on the vulnerability assessment, system-level virtualization solutions have many more vulnerabilities than OS-level virtualization solutions. As such, we recommend that sVirt be used with system-level virtualization solutions in order protect the host against exploits. Also, the majority of vulnerabilities related to KVM, LXC, and Docker are in specific regions of the system. Therefore, future zero-day attacks are likely to be in the same regions. Also, protecting these areas can simplify the protection of the host and maintain the isolation between users.

6.2.2 Initial Benchmarks

Our testing verified prior studies showing roughly 2-4% overheads in application execution time & MFLOPS when running in hypervisor-based virtualization environment as compared to native execution. We observed near-native application performance (time & MFLOPS) when running under container-based virtualization as compared to native execution. Also, we observed more consistent MFLOPS performance (except in 1 instance) with Docker runs of HPCCG than with Native runs of HPCCG, i.e., lower standard deviation over 20 runs, which is currently unexplained. However, the actual Docker vs. Native values were mostly the same, with Native achieving slightly better performance (lower Time, higher MFLOPS).

The standard deviation (over 20 runs) of HPCCG running on KVM was very high ($\sigma = 15$ to $\sigma = 30$). This may be due to a lack of resource pinning (e.g., CPU pinning), but further tests will be needed to confirm this suggestion. Regarding Time tests with HPCCG, all runs except two with KVM (kvm-300 and kvm-400) had very consistent values over the 20 runs, i.e., low standard deviation in Time in seconds for HPCCG runtime.

6.2.3 User namespaces

We identified Linux namespaces as a promising mechanism to isolate shared resources. This includes the most recent kernel additions (Linux ≥ 3.13) that include the `user` namespace, which was evaluated in Section 4.1 to support shared-storage isolation. Those experiments confirmed that different UIDs could be mapped between the guest/host environments when accessing shared resources. The near-native performance results of HPCCG running in a Docker-based container, which uses namespaces to implement the isolation, is another reason we believe this to be an interesting avenue to pursue for secure compute customization.

Since `user` namespaces is a very recent addition to the kernel, its use cases are still evolving and higher-level tools like Docker are still working on formalizing an interface for users. The flexibility of user mappings is almost entirely inherited by LXC, but this interface used in the evaluation in Section 4.1 is still cumbersome and prone to mistakes. Discussions amongst developers engaged in the effort to bring `user` namespaces to Docker have converged on a model where a single username *docker-root* is designated to be always mapped to root within the container. Other users (besides root on the host) will be mapped one-to-one inside the container. This approach was taken in order to speculatively meet general user requirements, but it will not work for customizable computing environments where only a small subset of host users should be mapped into the VE. Otherwise, *docker-root* would be able to modify the files of all of those users. The second problem is with *docker-root* being shared between containers that access a shared-filesystem. If the chroot'ed environments of VEs overlap, then *docker-root* on one VE may be able to interfere with *docker-root* on the other container. If they could replace files with malicious binaries, and then convince the victim *docker-root* to run the binaries, they would then be able to access files of all users on the victim's VE.

LXD is a new open source project with the support of Canonical titled the "Linux Container Daemon" and and will provide an API for managing LXC containers with `user` namespaces enabled by default [33]. However, like Docker, it takes a simplified approach to the mappings, at least initially. It will only support a single contiguous range of users mapped into the container (e.g. UID 100000 on host maps to UID 0 in the container and UID 165534 maps to UID 65534 in the container).

For the use case presented in this work, we would desire the flexibility offered by bare LXC containers to set up (1) an unprivileged user on the host to map to root within the container and (2) specific ranges of UIDs that should be mapped one-to-one. It is possible that this functionality may be added to Docker or LXD after initial `user` namespace functionality has been merged.

6.2.4 Security Classifications

Lastly, as a point of clarification we note that based on the security classifications presented in Section 2.2, the technologies presented in this work belong to the class *C1* in their default configuration. This is because the technologies leverage Linux as their OS and Linux meets the three requirements for class *C1*. More clearly, Linux as used in this work by VMs and VEs satisfies these requirements by (1) using DAC, (2) having users with passwords, and (3) user applications are unprivileged while the kernel is privileged. For any of the virtualization platforms to be considered of class *C2*, they would need to both sanitize objects before use and re-use and provide logging for various actions performed on the system. Sanitization is an easier problem to solve if the VM or VE is just considering adding a further layer of isolation, where the VM or VE is not a multi-user environment itself. In this case, the host already ensures objects are sanitized before re-use by another process. With respect to the *B1* classification that requires use of label-based Mandatory Access Controls, VMs could use SELinux and Audit to meet the requirements, but SELinux cannot run inside the container. That being said, if the use case is limited to a single application running within a VE, SELinux can be enabled on the host and sVirt can be used to label the container's processes and enforce a MAC policy unique to that VE.

Since RHEL7 was released only in June 2014, and this is the first release where Red Hat has supported containers, we found mention of one bug in the audit framework and expect several other refinements will be needed to reach the classifications acquired by RHEL 5 and 6.

6.3 Future Plans

We conclude with a few remarks about plans moving forward and possible avenues for further investigation. The current benchmarking efforts will continue and provide data to measure the performance as additional isolation mechanisms are introduced. We will also be performing more parallel application tests and file-system/storage tests. We have started to investigate prior work in paravirtualized filesystems (VirtFS) [25]. This looks like a promising approach for providing efficient sharing of host filesystem mounts with guests, in contrast to network filesystem such as NFS or CIFS. It is implemented using 9p filesystem support in Linux and the VirtIO PCI interface which can support zero-copy operations, thus reducing inefficiencies from extra copies and packetization for TCP transport. An enhanced security model offered by VirtFS called “mapped”, stores access credentials in the extended attributes of files that are stored on disk. The access credentials stored are relative to client-user accessing the file, so different guest VMs can have completely isolated filesystem views.

There are also indications that the “mapped” VirtFS security model could be used to share a parallel filesystem mount (e.g. GPFS), but since this relies on a backing filesystem format change by using extended attributes, this may break compatibility with Lustre. Nonetheless, the security model of VirtFS with 9p is very interesting and we are interested in investigating how it can isolate shared storage.

Another area that might provide interesting future avenues for investigation is the use of VM recording to perform security audits. There has been prior work to log all non-deterministic input, allowing execution replay [16] to aid in forensics and debugging. These capabilities might also be advantageous for performing audits and even attestation that specific code was executed in [20] user-customizable environments.

Finally, we anticipate that hardware extensions that were added to Intel and AMD processors to support system-level virtualization with KVM or Xen will be adapted for VE environments as well. Intel VT-d extensions allow a guest to directly communicate with a PCIe function and thus remove the host’s networking bridge from the path in VE network connectivity. Also Extended Page Tables (EPT) give a guest saccess to dedicated physical memory. Research in these areas as applied to VMs are bringing them closer to OS-level VEs to capitalize on the ability to access virtualized hardware resources without a KVM or Xen VMM [43] [29]. A new open-source project LXD is aiming to make use of hardware isolation capabilities as a hypervisor for LXC containers in addition to bringing management features such as live migration and OpenStack integration [33].

6.4 Acknowledgements

This work was supported by the United States Department of Defense (DoD) and used resources of the DoD-HPC Program and the Compute and Data Environment for Science (CADES) at Oak Ridge National Laboratory.

Bibliography

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, September 2007. Publication Number: 24593, Revision: 3.14.

[2] AppArmor Libvirt: Confining virtual machines in libvirt with AppArmor. URL: `http://wiki.apparmor.net/index.php/Libvirt` [cited 23-nov-2014].

[3] AppArmor Security Project History. URL: `http://wiki.apparmor.net/index.php/AppArmor_History` [cited 30-nov-2014].

[4] AppArmor Security Project Wiki. URL: `http://wiki.apparmor.net` [cited 29-nov-2014].

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating System s Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[6] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference*, Anaheim, CA, USA, April 2005.

[7] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 574–579, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. `doi:10.1109/DATE.2010.5457142`.

[8] Greg Bronevetsky and Bronis de Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 155–164, New York, NY, USA, 2008. ACM. `doi:10.1145/1375527.1375552`.

[9] CAP_SYS_ADMIN: the new root. URL: `http://lwn.net/Articles/486306/` [cited 30-nov-2014].

[10] cgroups: Linux Control Groups Documentation. URL: `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt` [cited 30-nov-2014].

[11] Crispin Cowan. Securing Linux Applications With AppArmor, August 2007. Presentation at DEFCON-15 in Las Vegas, NV, August, 2007. URL: `https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-cowan.pdf` [cited 30-nov-2014].

[12] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[13] CRIU: Checkpoint/Restore In Userspace. URL: `http://www.criu.org` [cited 29-nov-2014].

[14] Department of Defense. *Trusted Computer System Evaluation Criteria*. December 1985. Note, also referred to as the "Orange Book.".

[15] Docker: An open platform for distributed applications for developers and sysadmins. URL: `https://www.docker.com` [cited 05-dec-2014].

[16] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1346256.1346273`, `doi:10.1145/1346256.1346273`.

[17] Renato Figueiredo, Peter A. Dinda, and José Fortes. Resource Virtualization Renaissance (Guest Editors' Introduction). *IEEE Computer*, 38(5):28–31, May 2005.

[18] R. P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press. `doi:http://doi.acm.org/10.1145/800122.803950`.

[19] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[20] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1924943.1924952`.

[21] Red Hat. (Whitepaper) KVM - Kernel-based Virtual Machine, September 1, 2008. URL: `http://www.redhat.com/resourcelibrary/whitepapers/doc-kvm` [cited 29-nov-2014].

[22] Michael A. Heroux and Jack Dongarra. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, June 2013. URL: `http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf` [cited 29-nov-2014].

[23] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th International Conference on Supercomputing (ICS)*, pages 125–134, New York, NY, USA, 2006. ACM Press. `doi:http://doi.acm.org/10.1145/1183401.1183421`.

[24] *iperf*: A tools to measure network performance. URL: `https://iperf.fr` [cited 04-dec-2014].

[25] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. VirtFS—A virtualization aware File System pass-through. In *Ottawa Linux Symposium*, pages 1–14, December 2010.

[26] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[27] Puppet Labs. Puppet Documentation Index. URL: `https://docs.puppetlabs.com/puppet/` [cited 02-dec-2014].

[28] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Stephen Jaconette, Mike Levenhagen, Ron Brightwell, and Patrick Widener. Palacios and Kitten: High Performance Operating Systems For Scalable Virtualized and Native Supercomputing. Technical Report NWU-EECS-09-14, Northwestern University, July 20, 2009. URL: `http://v3vee.org/papers/NWU-EECS-09-14.pdf`.

[29] Ye Li, Richard West, and Eric Missimer. A virtualized separation kernel for mixed criticality systems. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 201–212, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2576195.2576206`, `doi:10.1145/2576195.2576206`.

[30] Linux Kernel-based Virtual Machine (KVM). URL: `http://www.linux-kvm.org` [cited 29-nov-2014].

[31] Linux VServer project. URL: `http://linux-vserver.org` [cited 19-nov-2014].

[32] LXC - Linux Containers: Userspace tools for the Linux kernel containment features. URL: `https://linuxcontainers.org` [cited 19-nov-2014].

[33] LXD: The Linux Container Daemon. URL: `http://www.ubuntu.com/cloud/tools/lxd` [cited 30-nov-2014].

[34] Mantevo mini-application downloads. URL: `http://www.mantevo.org/packages.php` [cited 29-nov-2014].

[35] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology, September 2011. URL: `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf` [cited 29-nov-2014].

[36] James Morris. sVirt: Hardening Linux virtualization with mandatory access control, 2009. Presentation at Linux Conference Australia (LCA). URL: `http://namei.org/presentations/svirt-lca-2009.pdf` [cited 23-nov-2014].

[37] OpenStack Security Guide, 2014. This book provides best practices and conceptual information about securing an OpenStack cloud. URL: `http://docs.openstack.org/security-guide/content/index.html` [cited 23-nov-2014].

[38] OpenVZ: Container-based virtualization for Linux. URL: `http://www.openvz.org` [cited 19-nov-2014].

[39] Red Hat. *Red Hat Enterprise Linux 7 Resource Management and Linux Containers Guide*, 2014. URL: `http://goo.gl/Y4rB5D` [cited 30-nov-2014].

[40] Rusty Russell. virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. `doi:10.1145/1400097.1400108`.

[41] SELinux: Security Enhanced Linux. URL: `http://selinuxproject.org` [cited 29-nov-2014].

[42] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th Conference on USENIX Security Symposium*, volume 8 of *SSYM'99*. USENIX Association, 1999.

[43] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 401–412, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2046707.2046754, doi:10.1145/2046707.2046754.

[44] Juliean Tinnes and Chris Evans. Security in-depth for Linux software, October 2009. URL: https://www.cr0.org/paper/jt-ce-sid_linux.pdf [cited 30-nov-2014].

[45] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel® Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.

[46] Geoffroy R. Vallée, Thomas Naughton, Christian Engelmann, Hong H. Ong, and Stephen L. Scott. System-level virtualization for high performance computing. In *Proceedings of the* 16^{th} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2008*, pages 636–643, Toulouse, France, February 13-15, 2008. IEEE Computer Society, Los Alamitos, CA, USA. URL: http://www.csm.ornl.gov/~engelman/publications/vallee08system.pdf, doi:http://doi.ieeecomputersociety.org/10.1109/PDP.2008.85.

[47] Lamia Youseff, Keith Seymour, Haihang You, Jack Dongarra, and Rich Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC'08)*, pages 141–152, New York, NY, USA, 2008. ACM. doi:http://doi.acm.org/10.1145/1383422.1383440.

Appendix A

Docker

A.1 Docker Files

The following source listings provide an example of a Docker image description file. These listings also provide details about the configurations used in our testing.

Listing A.1. Example Dockerfile for CentOS v7 image that includes the HPCCG benchmark and GNU compilers.

```
# $Id: Dockerfile 263 2014-12-02 15:00:18Z tjn3 $
# TJN adding G++ to CentOS7 for HPCCG testing
#
# To rebuild image:
# sudo docker build -t="naughtont3/centos7cxx" .
# sudo docker push naughtont3/centos7cxx

FROM centos:centos7
MAINTAINER "Thomas␣Naughton" <naughtont@ornl.gov>

ADD etc/yum.repos.d/epel-ornl.repo /etc/yum.repos.d/epel-ornl.repo
ADD etc/yum.repos.d/rhel7.repo /etc/yum.repos.d/rhel7.repo
CMD ["chmod","0644","/etc/yum.repos.d/epel-ornl.repo","/etc/yum.repos.d/rhel7.repo"]

RUN yum -y update; yum clean all
RUN yum -y install epel-release; yum clean all
RUN yum -y install libcgroup-tools; yum clean all
RUN yum -y install gcc gcc-c++ libstdc++ libstdc++-devel; yum clean all
RUN yum -y install wget net-tools; yum clean all

CMD ["mkdir","-p","/benchmarks"]
ADD benchmarks/HPCCG-1.0.tar.gz /benchmarks
ADD benchmarks/Makefile.HPCCG /benchmarks/HPCCG-1.0/Makefile.HPCCG
ADD benchmarks/Makefile.HPCCG+mpi /benchmarks/HPCCG-1.0/Makefile.HPCCG+mpi
ADD benchmarks/Makefile.HPCCG /benchmarks/HPCCG-1.0/Makefile
```

Listing A.2. Example Dockerfile that extends base CentOS7-HPCCG image to include Open MPI, an `mpiuser` and configurations for inter-container MPI launch via SSH.

```
# BC adding OpenMPI and SSHD to CentOS7/HPCCG bundle
#
# To rebuild image:
# sudo docker build -t="blakec/centos7-sshd-mpi" .
# sudo docker push blakec/centos7-sshd-mpi

FROM naughtont3/centos7cxx
MAINTAINER "Blake␣Caldwell" <blakec@ornl.gov>
ENV container docker

# Install the real systemd
RUN yum -y swap -- remove fakesystemd -- install systemd systemd-libs

# Install all packages and create mpiuser with authentication by ecdsa key
RUN yum -y install openssh-clients openssh-server openmpi openmpi-devel net-tools; \
 yum clean all; \
 adduser mpiuser; \
 su -c "echo␣'export␣LD_LIBRARY_PATH=/usr/lib64/openmpi/lib:\$LD_LIBRARY_PATH'␣>>␣~
     mpiuser/.bashrc" mpiuser; \
 su -c "echo␣'export␣PATH=/usr/lib64/openmpi/bin/:\$PATH'␣>>␣~mpiuser/.bashrc" mpiuser
     ; \
 su -c "/usr/bin/ssh-keygen␣-t␣ecdsa␣-f␣~/.ssh/id_ecdsa␣-q␣-N␣''" mpiuser; \
 su -c "cat␣~/.ssh/id_ecdsa.pub␣␣>>␣~/.ssh/authorized_keys" mpiuser; \
 ssh-keygen -t ecdsa -f /etc/ssh/ssh_host_ecdsa_key -q -N ""; \
 su -c "echo␣-n␣'*␣'␣>␣~/.ssh/known_hosts␣&&␣cat␣/etc/ssh/ssh_host_ecdsa_key.pub␣>>␣
     ~/.ssh/known_hosts" mpiuser

# Configure systemd removing unecessary unit files
RUN (cd /lib/systemd/system/sysinit.target.wants/; \
 for i in *; do [ $i == systemd-tmpfiles-setup.service ] || rm -f $i; done); \
 rm -f /lib/systemd/system/local-fs.target.wants/*; \
 rm -f /lib/systemd/system/systemd-remount-fs.service; \
 rm -f /lib/systemd/system/sockets.target.wants/*udev*; \
 rm -f /lib/systemd/system/sockets.target.wants/*initctl*; \
 rm -f /lib/systemd/system/basic.target.wants/*; \
 rm -f /lib/systemd/system/anaconda.target.wants/*; \
 rm -f /lib/systemd/system/console-getty.service; \
 rm -f /etc/systemd/system/getty.target.wants/*; \
 rm -f /lib/systemd/system/getty@.service; \
 rm -f /lib/systemd/system/multi-user.target.wants/getty.target; \
 /usr/bin/systemctl enable sshd.service
VOLUME [ "/sys/fs/cgroup" ]

# start systemd
CMD ["/usr/sbin/init"]
```

Appendix B

libvirt

B.1 libvirt Files

The following source listings provide an example of a libvirt configuration file. These files describe the "virtual hardware" configuration for the virtual machine. These listings also provide details about the configurations used in our testing.

Listing B.1. Example libvirt XML for CentOS v7 image, which is setup for bridged networking.

```
<domain type='kvm'>
  <name>centos7kvm</name>
  <uuid>70564581-691f-43b5-aeab-c0793fab9071</uuid>
  <memory unit='KiB'>50331648</memory>
  <currentMemory unit='KiB'>50331648</currentMemory>
  <vcpu placement='static'>31</vcpu>
  <os>
    <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/libexec/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' cache='none'/>
      <source file='/var/lib/libvirt/images/centos7-x86_64.qcow2'/>
      <target dev='vda' bus='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
    </disk>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='/var/lib/libvirt/images/user-data.img'/>
      <target dev='vdb' bus='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
    </disk>
    <controller type='usb' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
    </controller>
    <controller type='pci' index='0' model='pci-root'/>
    <interface type='bridge'>
      <mac address='52:54:00:17:bd:79'/>
      <source bridge='br-eth2'/>
      <model type='e1000'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='virtio'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
    </memballoon>
  </devices>
</domain>
```