ORNL/TM-2016/710

Secure Enclaves: An Isolation-centric Approach for Creating Secure High Performance Computing Environments

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

June 2015

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

ORNL/TM-2016/710

Computing & Computational Sciences Directorate

DoD-HPC Program

Secure Enclaves: An Isolation-centric Approach for Creating Secure High Performance Computing Environments

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: June 2015

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

Secure Enclaves: An Isolation-centric Approach for Creating Secure High Performance Computing Environments

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 at various security levels but in so doing are often enclaved at the highest security posture. This approach places significant restrictions on the users of the system even when processing data at a lower security level and exposes data at higher levels of confidentiality to a much broader population than otherwise necessary. The traditional approach of isolation, while effective in establishing security enclaves poses significant challenges for the use of shared infrastructure in HPC environments. This report details current state-of-the-art in virtualization, reconfigurable network enclaving via Software Defined Networking (SDN), and storage architectures and bridging techniques for creating secure enclaves in 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 to virtualization. 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.

We consider both VE and VM approaches and review 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 evaluations were focused primarily on the use of KVM for hypervisor-based experiments, and LXD/LXC for container-based tests. The literature reviews and technology examinations included a variety of approaches, such as Linux containers (LXC [71], Docker [35]) and full virtualization (KVM [57], Xen [8]).

The evaluations using the different virtualization technologies are discussed in Section 9. 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 9.2 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 12.2.2.

The other evaluations that were performed during Year-1 of the project provide baseline performance data for comparing VEs and VMs to purely native execution. In Section 9.3 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 9.9 (page 72), 9.10 (page 72), and 9.11 (page 73).

The two most complete experiments focused on controlling access to storage, while maintaining good performance, when using a VM or VE to access a parallel filesystem (Lustre). In Section 9.7, two different IO-Forwarding methods were compared to evaluate the performance of re-exporting a host mounted Lustre filesystem to a guest VM. The VirtFs/9pfs showed very good performance for IOR and FIO benchmark tests. The other tests (Section 9.8) looked at the VE instance and showed that a combination of bind mounts and user namespaces could provide secure near native performance for accessing Lustre from the VE.

SDN and NFV methods are based on a solid foundation of system wide virtualization. The purpose of which is very straight forward, the system administrator can deploy networks that are more amenable to customer needs, and at the same time achieve increased scalability making it easier to increase overall capacity as needed without negatively affecting functionality. The network administration of both the server system and the virtual sub-systems is simplified allowing control of the infrastructure through well-defined APIs (Application Programming Interface). While SDN and NFV technologies offer significant promise in meeting these goals, they also provide the ability to address a significant component of the multi-tenant challenge in HPC environments, namely resource isolation. Traditional HPC systems are built upon scalable high-performance networking technologies designed to meet specific application requirements. Dynamic isolation of resources within these environments has remained difficult to achieve. SDN and NFV methodology provide us with relevant concepts and available open standards based APIs that isolate compute and storage resources within an otherwise common networking infrastructure. Additionally, the integration of the networking APIs within larger system frameworks such as OpenStack provide the tools necessary to establish isolated enclaves dynamically allowing the benefits of HPC while providing a controlled security structure surrounding these systems.

Key Points The container-based virtualization shows great promise for providing efficient and secure compute enclaving. The hypervisor-based virtualization can achieve good performance with IO-Forwarding based on VirtFS (9pfs). We are able to leverage OpenStack and SDN to achieve on-demand network enclaving, as demonstrated on SE testbed deployed at ORNL.

SDN and NFV provides the functionality necessary to configure distributed networking components on-demand, while at the same time providing desired performance, security, and reliability goals. The requirements of these open standards are largely driven by the cloud computing community. Adapting these standards to HPC systems can provide an increased level of flexibility with significantly higher performance than that of a typical cloud computing infrastructure. Reconfigurable networks are a key component of this flexibility providing a unique opportunity to achieve the performance and application scalability of leading edge HPC platforms while providing the ability to isolate applications within a shared infrastructure.

There are a few existing and in-progress protection features in Lustre related to secure storage, which are discussed in (Chapter 5.1). These include authentication capabilities like GSSAPI/Kerberos and the in-progress work for GSSAPI/Host-keys. The GPFS filesystem provides native support for encryption, which is not directly available in Lustre. Additionally, GPFS includes authentication/authorization mechanisms for inter-cluster sharing of filesystems (Chapter 5.2). The limitations of key importance for secure storage/filesystems are: (i) restricting sub-tree mounts for parallel filesystem (which is not directly supported in Lustre or GPFS), and (ii) segregation of hosts on the storage network* and practical complications with dynamic additions to the storage network, e.g., LNET. A challenge for VM based use cases will be to provide efficient IO forwarding of the parallel filessytem from the host to the guest (VM). There are promising options like para-virtualized filesystems to help with this issue, which are a particular

*The network that connects the storage subsystem and users, e.g., Lustre's LNET.

instances of the more general challenge of efficient host/guest IO that is the focus of interfaces like `virtio`. A collection of bridging technologies have been identified in Chapter 6, which can be helpful to overcome the limitations and challenges of supporting efficient storage for secure enclaves. The synthesis of native filesystem security mechanisms and bridging technologies led to an isolation-centric storage architecture that is proposed in Chapter 8, which leverages isolation mechanisms from different layers to facilitate secure storage for an enclave.

Recommendations 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 10 on page 90. 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.

Additional research into the application of SDN and NFV technologies within an HPC context is required. Leveraging large-scale orchestration frameworks such as OpenStack to manage HPC system components will broaden the applicability and improve the security of HPC systems. While our initial work focuses on leveraging SDN and NFV capabilities of Ethernet based networks for secure enclaves, the proposed techniques are readily adaptable to high-performance networking technologies utilized within HPC. Adopting SDN, NFV and broader orchestration technologies such as OpenStack for on-demand network reconfiguration will require further development including scalable low-overhead tools that provide monitoring and auditing of networking components (including endpoints). All this development should be within the scope of compliance with applicable and necessary security policies.

In the context of storage, the Lustre filesystem offers excellent performance but does not support some security related features, e.g., encryption, that are included in GPFS. If encryption is of paramount importance, then GPFS may be a more suitable choice. There are several possible Lustre related enhancements that may provide functionality of use for secure-enclaves. However, since these features are not currently integrated, the use of Lustre as a secure storage system may require more direct involvement (support). The use of OpenStack with GPFS will be more streamlined than with Lustre, as there are available drivers for GPFS. The Manila project offers "Filesystem as a Service" for OpenStack and is worth further investigation. Manila has some support for GPFS. The proposed Lustre enhancement of Dynamic-LNET should be further investigated to provide more dynamic changes to the storage network which could be used to isolate hosts and their tenants. The Linux namespaces offer a good solution for creating efficient restrictions to shared HPC filesystems. However, we still need to conduct a thorough round of storage/filesystem benchmarks.

Outline The remainder of this report is structured as follows:

- Section 1: Introduces the topic of secure enclaves and clarifies the scope of the project, to include working assumptions about the threat model for secure enclaves.
- Section 2: Provides background and terminology used throughout the report. This section includes details on Software Defined Networking (SDN), storage architectures, system mangement tools, and

security & virtualization classifications.

- Section 3: A review of isolation mechanisms for container and hypervisor based solutions. This section provides information on security frameworks for use with virtualization.
- Section 4: Details alternative architectures and available methods for implementing dynamically reconfiguring networks.
- Section 5: A review of protection mechanisms in two HPC filesystems; details about available isolation, authentication/authorization and performance capabilities are discussed.
- Section 6: Describe technologies that can be used to bridge gaps in HPC storage and filesystems to facilitate more secure storage.
- Section 7: A brief overview of key implementation details for the OpenStack cloud software stack. This chapter also includes details on emerging components and services in the OpenStack project that are particular interest for the secure enclaves effort.
- Section 8: We describe an isolation-centric approach for a secure high-performance environment, and provide example instances to clarify applications in the secure enclaves effort.
- Section 9: Details on evaluations related to compute, network and storage experiments for secure enclaves. This chapter includes an overview of our secure enclave testbed, which is a prototype for the architecture described in Chapter 8.
- Section 10: Discusses a secure compute 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 11.1: An overview of a number of SDN and Network Function Virtualization (NFV) vendor technologies and their capabilities. Also, a brief analysis of storage vendors with products that could potentially be of use for creating secure enclaves.
- Section 12: A brief synopsis of the report and highlight key observations from Year-1 of the project as well as points of interest for further investigations.

Chapter 1

Introduction

High performance computing environments are 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 at various security levels but in so doing are often enclaved at the highest security posture. This approach places significant restrictions on the users of the system even when processing data at a lower security level and exposes data at higher levels of confidentiality to a much broader population than otherwise necessary. The traditional approach of isolation, while effective in establishing security enclaves poses significant challenges for the use of shared infrastructure in HPC environments.

The ability to support "on-demand self-service" computing resources is a major asset of Cloud Computing [79]. 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.

As the use of virtualization becomes more ubiquitous, additional hardware support is emerging to assist with the multiplexing of the physical resources. Many hardware specific services such as data storage and networking were not initially easily realizable with available virtual machine technologies. However, newer hardware functionality is helping to improve performance when virtualizing these critical I/O services. In the specific case of HPC workloads, latency and bandwidth requirements place a higher performance demand on these virtualized services and the hardware used to realize them. The adapting of virtualization methods within the HPC community requires a more narrowly focused approach to virtualization. Streamlined techniques such as the use of Linux containers provides a virtualized environment rather than a complete virtual machine, enabling the flexibility desired within the HPC community without sacrificing performance system performance.

This report reviews current state-of-the-art in reconfigurable secure networking, secure high-performance compute customization, and secure access to shared HPC storage resources. There are numerous factors that influence the degree of control and isolation that can be achieved in current HPC environments. The key technologies associated with secure enclaves are detailed. An architecture for a high-performance secure enclave is presented along with details for a prototype implementation and evaluations that that illustrate the capabilities and highlight areas for refinement. Therefore, this report focuses on these three areas: (i) secure HPC compute, (ii) secure HPC network, and (iii) secure HPC storage. 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 (Chapter 8). Specifically we examine the use case of sharing common distributed parallel filesystems and a high speed network infrastructure.

In the area of secure compute, we review current OS protection mechanisms and virtualization technologies to better understand the performance/isolation properties. We examine the feasibility of using "virtualized" computing contexts to enable improved compute customization, which includes controlled support for users having increased administrative permissions (within the virtualized context). We also analyze the vulnerabilities of current virtualization based environments.

The growth of server virtualization is spurring increased interest in technologies that can be leveraged to aid with network virtualization. A key element of modern networking with virtualized resources is the combination of SDN and NFV. As industry standard APIs are developed using a common open source standard, network appliance operation moves seamlessly within the compute infrastructure.

In the area of secure network, we present results from our investigation into mechanisms that can be used to implement reconfigurable networks. The intent is to leverage these networking technologies to facilitate isolation in multi-tenant environments. The report focuses on SDN to gain insights into its use in a high-performance computing (HPC) context. This includes a review of methods and technologies for implementing reconfigurable networks and a snapshot of key vendors that are providing products that support SDN.

Lastly, in the area of secure storage, we provide background information on two major HPC storage architectures: Lustre and GPFS. We focus on their security profiles; highlighting strengths, discussing weaknesses and possible workarounds. Particular attention is paid to areas where security gaps can be bridged by using other technologies. The resulting SE architecture offers an isolation-centric approach to secure high-performance storage.

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 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

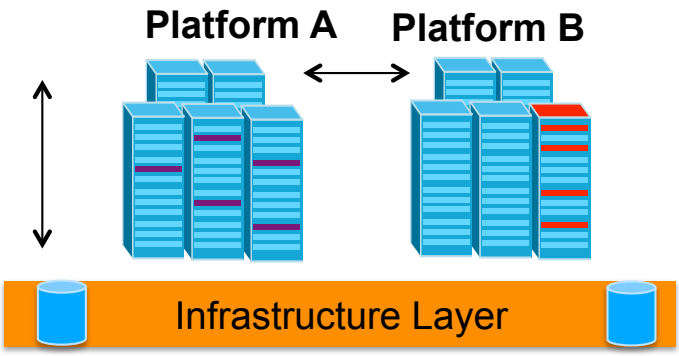


Figure 1.1. Illustration of two axes of administrative control (platform and infrastructure) in a customizable computing environment.

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 presented in Chapter 9.

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. Additionally, we provide background on resource management and orchestration capabilities available through SDN and NFV. We also include information related to general security technologies and system management tools.

Chapter 3 focuses on key virtualization technologies that are relevant to this project. In Chapter 4 details are given about alternative architectures and available methods for implementing dynamically reconfiguring networks. Chapter 5 provides details on security related features of selected HPC filesystems, namely Lustre and GPFS, and contrasts some of their differences with respect to secure storage. Chapter 6 discusses technologies that can help to bridge gaps in secure HPC storage, which were identified in Chapter 5.

Chapter 7 reviews OpenStack implementation details, to include the core components as well as emerging components, e.g., Manila Filesystem-As-A-Service and Magnum Containers-As-A-Service. Chapter 8 presents our view of an isolation-centric system architecture. This describes our approach to secure enclaves in HPC. The prototype SE testbed from Year-1 leverages OpenStack.

The evaluations and demonstrations from Year-1 are provided in Chapter 9. This includes testing in both virtual machine and container based environments to show the performance when employing different isolation mechanisms for secure enclaves.

A vulnerability assessment for different virtualization technologies is given in Chapter 10. Chapter 11 provides an overview of a number of SDN vendor technologies and their capabilities. We also include a review of vendor offerings in the area of secure storage.

Finally, in Chapter 12 we discuss our observations and conclude the report. An appendix is included with auxiliary details on security protection levels (Appx. A), Docker (Appx. B) and libvirt (Appx. C).

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.

Note, we generalize virtualization technologies into two groups: hypervisor-based and container-based. Throughout the report we use the term **virtual machine (VM)** when referring to hypervisor-based virtualization, e.g., KVM. We use the term **virtual environment (VE)** when referring to container-based virtualization, e.g., Docker/LXC.

2.1 Terminology

2.1.1 Virtualization

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 2.5. 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 2.2 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.

2.1.2 Networking

This section reviews relevant terminology and background concepts. Standardizing of terminology is still being worked out in the network virtualization community and inconsistencies exist in the literature.

For example, the available vendor documentation and associated research on both SDN and OpenFlow discuss basic capabilities and functions in application specific terminology, often focusing on specific use cases, rather than a generic capability.

Software Defined Networking (SDN) In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications. As a result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs [39].

OpenFlow OpenFlow is an open standard that enables researchers to run experimental protocols in the campus networks we use every day. OpenFlow is added as a feature to commercial Ethernet switches, routers and wireless access points and provides a standardized hook to allow researchers to run experiments, without requiring vendors to expose the internal workings of their network devices. OpenFlow is currently being implemented by major vendors, with OpenFlow-enabled switches now commercially available [78].

DirectFlow Arista's DirectFlow is an enhanced version of OpenFlow. As per [43], "*DirectFlow extends the capabilities of OpenFlow with controller-less operation and enables per-flow pattern-matching with full control.*" The DirectFlow Control product enables a user to perform operations on the switch via the Command Line Interface (CLI) and EOS API (eAPI)[1] interface [43].

OpenDaylight OpenDaylight is a collaborative, open source project to advance Software-Defined Networking (SDN). OpenDaylight is a community-led, open, industry-supported framework, consisting of code and blueprints, for accelerating adoption, fostering new innovation, reducing risk and creating a more transparent approach to Software-Defined Networking [90].

Control plane and Data Plane In traditional networking, the control plane and data plane traffic shares the same path. In SDN, the control and data are separated to facilitate an abstract network design. Control plane traffic consists of L2, and L3 protocols, management traffic such as Simple Network Management Protocol (SNMP) and Secure Shell (SSH). The data plane is the traffic containing the data exchanged between applications, i.e., application data.

Network Abstraction The concept of network abstraction is primarily focused on supporting network policy and controls rather than specific methods that can be used to deploy the controls through physical hardware. In the context of SDN, it refers to connections, ports and data flow policies rather than the physical connection descriptions such as VLANs, IP addresses, and physical networking devices. This network abstraction layer facilities APIs that can be used to configure details about the network.

Network Decoupling The separation of the control plane, and the data plane, allows the network to be abstracted. The control plane is defined in general terms and manages policies. The data plane is the physical interface that acts on these policies, thus abstracting the network design, from the planning and manipulation of physical connections.

[1] EOS is Arista's Extensible Operating System that runs on their network switch hardware.

Northbound and Southbound network interfaces The concept of North and Southbound traffic refers to the information exchanged between the decoupled control and data planes of the SDN. Northbound specifically refers to information from the data plane to the control plane, and Southbound refers to information from the control plane to the data plane. Restated, the Southbound interface involves the controller-to-switch interaction and is defined by protocols like OpenFlow [89]. Conversely, the Northbound interface involves the controller(s) and network services/application and the standards for this are less well defined [89].

In the context of OpenDaylight, a Northbound interface allows applications to gather information about the network used to modify the existing connection resources and capabilities such as bandwidth assigned to the network. The Southbound interface deals with the hardware, and network layers, control policies are translated to the data plane as instructions for connections, traffic management and security policies [91].

Agents and Controllers Tenants or applications interact with provisioning mechanisms that communicate with agents or controllers. This allows the administrator to reconfigure the network to meet tenant or application needs. An agent requests network resources through an API in the controller, which then provisions the requested resources. The available resources are reported back to the agent. This exchange allows the agents or tenants to dynamically provision resources during a heavy load, and then release resources when the demand is low.

2.2 Virtualization Classification

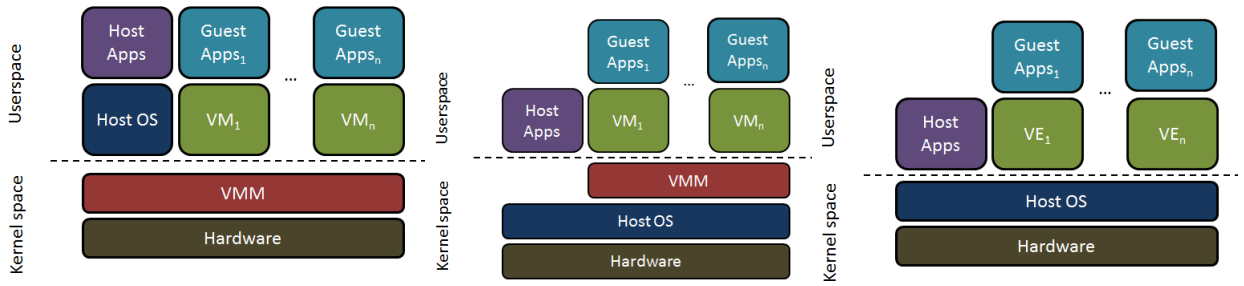
Virtualization is the abstraction of the system layer in order to achieve various goals including isolated execution, compute customization, and environment portability. A benefit enjoyed by cloud computing environments is making use of the abstraction for resource sharing, allowing for higher resource 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.

2.2.1 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).

2.2.2 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 [41]: *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 [8], while the kernel-based virtual machine (KVM) [47, 63] is an example of a type-2 VMM.

2.3 SDN and Network Function Virtualization

Prior work in *programmable networks* laid the foundation for the current efforts into Software Defined Networking (SDN) [89]. Fundamentally, the SDN architectural model is based on the notion of decoupling the control and data channels. This separation enables the control portion to be managed in a more flexible manner without binding it to the actual data forwarding layer [83, 89], i.e., the control and data may be managed (even implemented) separately.

This separation can be leveraged by virtualized environments to allow more dynamic configuration of the network to meet the needs of applications. Allowing tenants (customers) to provision and configure dynamic networks can be beneficial for testing applications, or scaling an existing production environment or specific application. Virtualization saves time for the tenants since they don't have to wait for network administrators to provision and configure additional network resources. This saves time for both the network engineers and systems engineers. The system engineers can focus on adding to resource capacity, leaving the virtualization controller to handle the tenant flexibility needs. SDN works by separating the control plane and data plane in the network environment. The control plane handles the configuration and use management of available network resources including routing and monitoring functions. The control plane is responsible for QoS and security policy enforcement on the network connections. The data plane handles the actual flow of data between applications with connections and port sharing under direct management of the control plane between tenant compute nodes and any external network connections.

In a traditional network the routers, switches, firewalls, and load balancers are dedicated to a physical configuration. Often these pieces of hardware are from different vendors. Organizational network topologies are typically centered on these functions. Virtualized networking is focused on commodity servers that can perform all of these functions to various extents. The standardization of software based

services as opposed to application specific physical appliances provides on-demand flexibility in provisioning the layout of the newly defined system. Routers, firewalls and load balancers can be rapidly deployed as needed in a virtualized environment. As network commodity servers improve in performance and lowered cost, additional network function virtualization can be realized. Advances in the switch fabric ASIC and corresponding controllers will allow MAC (media access control) functions such as layer 2 and layer 3 level control functions to be transferred to the virtualized network control resulting in a reduction in system cost coupled with increased deployment flexibility.

The large scale adaptation of SDN facilitates the dynamic reconfiguration of networks to meet the needs of both specific user requirements and applications. The incorporation of Network Function Virtualization (NFV) on the other hand is changing how networks are scaled, enabling dynamically configured functions such as firewalls, and load balancers to optimize deployment time. Systems can be deployed and realized based on available CPU, network, and memory capacity in the virtual server farms. This deployment model saves the network engineers from having to focus on rack space, cooling, and cabling requirements associated with specific system expansion needs. NFV deployment reduces custom hardware support costs, however server support costs will increase.

System performance requirements are more easily realized using SDN and NFV. The use of SDN allows the user to quickly spin up network functions such firewalls and load balancers based on specific needs. Additionally SDN has the potential to assist in optimizing traffic flows within the network to reduce latency and network hot spots. For example, in a SDN environment, software could detect that tenant traffic is spread out and is pushing heavy traffic among nodes causing potential hot spots and requiring QoS to be enabled. Rearranging the host servers and changing the network to meet that service level agreement is possible with reconfigurable networks. The software provisions the network in an underutilized area with respect to virtual server resources and available network resources providing the hypervisors with the necessary resources to move tenant traffic over to the newly created service. Likewise the software could detect that a virtualized load balancer, or firewall, is nearing capacity and can spin up and configure replacement virtual services with additional capability. The combination of SDN and NFV allows higher functionality, while being able to monitor the network, and modify configurations as needed.

2.4 Storage Architectures

Storage systems are integral components of HPC environments. Historically, HPC storage systems were mostly installed in government labs, large universities and a few commercial research and development labs. The emphasis was on achieving performance and scalability suitable for a HPC context and less on implementing strong security controls. Often very basic protections were judged sufficient. The shift toward more ubiquitous Internet access and the corresponding increase in shared storage across compute platforms, especially cloud-based platforms, has increased the demand for security in HPC storage. We focus on two HPC storage solutions, Lustre and GPFS, as we believe they collectively represent the more mature, feature-rich, performant, scalable, cost effective and actively developed architectures.

2.4.1 The Lustre Storage Architecture

Lustre is a popular storage architecture in the HPC world. The Lustre filesystem has been utilized today by 7 out of 10 of the TOP10 supercomputing sites and over 60% of the TOP100 [93]. It is open source (GPLv2) and is available for several Linux variants. It is a massively parallel filesystem, capable of

tremendous I/O performance. Its ability to easily scale capacity and performance by adding more servers is another reason why Lustre is so popular. Clients and servers on a Lustre network communicate via a special networking API called LNET, Lustre networking.

LNET supports a variety of transport protocols including TCP, Infiniband's o2ib, Cray's Seastar and Gemini, Myrinet's MX, RapidArray's *ra* and Quadrics' Elan [93]. LNET also supports routers which provide the capability of routing traffic between different IP networks whose underlying Layer 2 technology might differ. A common use of routers is to bridge between an Ethernet network and an Infiniband fabric [93], often to extend the storage resources on an Infiniband fabric to clients without Infiniband connectivity. Infiniband-to-Infiniband routers are also used to physically partition a large fabric, while still allowing Lustre filesystem access between the partitions. An example of a potential Lustre configuration for basic cluster installation is shown in Figure 2.2.

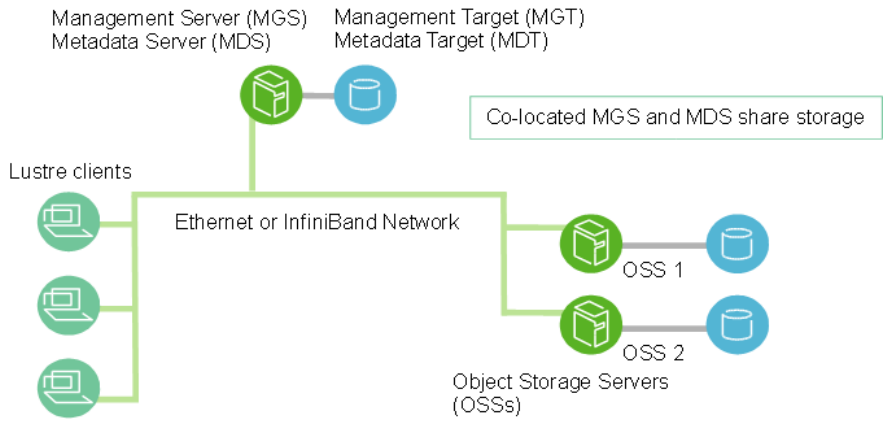


Figure 2.2. Illustration of Lustre configuration for basic cluster installation (figure source: [98]).

In Lustre, the management server (MGS) provides the clients with configuration information, such as the location of the metadata server (MDS), storage servers, and filesystem parameters. It also serves as the first point of contact for a client that wishes to mount the filesystem. To open a particular file, the client contacts a MDS server to receive the metadata for that file. This is stored in an inode similar to traditional filesystems such as *ext4*, but information on the layout of the file is stored in the extended attributes portion. The layout information consists of Object Storage Target (OST) indexes and object numbers for each chunk of the file, which reside throughout the cluster. The client can then contact the Object Storage Server (OSS) on which the specified OSTs reside and gather all chucks from the file. This interaction is illustrated in Figure 2.3, which shows the steps for a Lustre client involved in writing data to the filesystem.

Lustre Components

Metadata Server (MDS) The MDS provides the interface between Lustre clients and the Metadata Target (MDT). The metadata includes file and directory names and their associated inode (location) in the Lustre filesystem [98].

Management Server (MGS) The MGS provides an information service for Lustre, which includes configuration details about all Lustre filesystems in the cluster. The MGS is independent of an individual

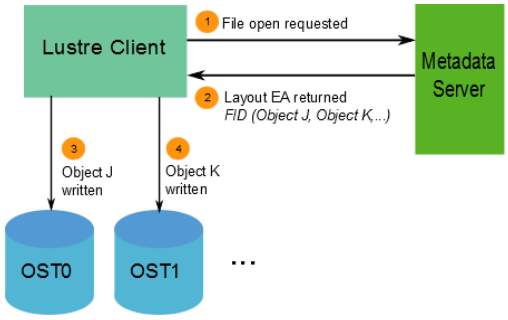


Figure 2.3. Diagram showing steps for Lustre client writing data (figure source: [98]).

filesystem, instead it provides general configuration data for end-users and Lustre components themselves [98].

Metadata Target (MDT) The MDT is responsible for maintaining Lustre metadata, e.g., file and directory names, permissions. The MDT is the storage portion of the Lustre metadata, and is accessed via the MDS [98]. The MDT is associated with a filesystem (one filesystem, one MDT) and the shared target (MDT) can be accessed by many MDSs, but for consistency only one should use it. In the event of an MDS failure, a secondary MDS can serve out the MDT to clients [98].

Object Storage Servers (OSS) The OSS provides I/O services for network file requests. The OSS obtains the file data from OST. An OSS is generally paired with 2-8 OSTs [98], which can each be as large as 128 TB [70]. The MDT, OST and Lustre clients can run on a single node but generally are separated to different machines, e.g., MDT on node, OSTs on OSS node, Lustre client on compute nodes [98].

Object Storage Target (OST) The OST stores file data as objects, i.e., "chunks of user files" [98]. Multiple OSTs are generally used for a single Lustre filesystem with file "chunks" (objects) spread across the different OSTs. The mapping between file and OST is not necessarily one-to-one, and in many cases to improve performance the files are spread across several OSTs [98]. The management of these "file striping" over OSTs is maintained by a Logical Object Volume (LOV) [98].

Lustre Lock Manager (LDLM) The LDLM coordinates filesystem access, which is run by the MDS [98]. The lock manager used by Lustre is based on the design employed by the VAX distributed lock manager [59].

Portal RPC (PTLRPC) The Portal Remote Procedure Call (RPC), `PTLRPC`, is the underlying mechanism used within LNET for the client/server exchanges. The mechanism is responsible for [118]:

- sending requests through imports[2] and receiving replies,
- receiving and processing requests through exports and sending replies,
- performing bulk data transfer, and
- error recovery.

[2]Lustre import/export are communication pairings used for receiving/sending over LNET.

Lustre Clients The clients are Linux kernel modules that interfaces with the Linux Virtual File System (VFS) and the backend Lustre data servers [98]. The clients mount the Lustre filesystem to provide a seamless view of the underlying parallel infrastructure. The client ensures POSIX compatibility for the user of the filesystem, which include coherent, synchronized filesystem access at all times [98]. There are different clients for the various Lustre components [98]:

- Metadata Client (MDC) to interface with MDT,
- Object Storage Client (OSC) to interface with OST, and
- Management Client (MGC) to interface with MGS.

Lustre Network Driver (LND) A LND provides the interface between the underlying physical network and the abstracted LNET. There are several drivers available for Lustre, e.g., Ethernet (TCP/IP), Infiniband, Quadrics Elan, Myrinet, and Cray (SeaStar Or Gemini) [98].

Lustre Networking (LNET) Lustre uses an internal network abstraction layer that offers an API for managing metadata and file I/O via server and clients [98]. The LNET layer abstracts the underlying network fabric to allow various network transport layers to inter-operate; the Lustre clients and servers use LNET to have a common communication substrate independent of the underlying network technology [98]. LNET uses Lustre Network Drivers (LNDs) to communicate with the underlying networks, e.g., Ethernet, Infiniband, SeaStar.

2.4.2 The GPFS Storage Architecture

General Parallel File System (GPFS) is a propriety storage architecture from IBM that supplies a feature-rich filesystem. This is a cluster filesystem, providing concurrent access to files from multiple clients. While it possesses many enterprise class storage attributes, GPFS is also a popular choice for HPC. Its features include replication, information life-cycle management (ILM), cloning, snapshots, global namespace, encryption and authentication. Also, GPFS is available for several operating systems: Linux, AIX and Windows.

There are two main methods by which GPFS shares data across clusters: GPFS multicluster and Active File Management (AFM). Multicluster allows GPFS filesystems to share files amongst themselves after the appropriate authentications and authorizations. Multicluster is more suited for sharing filesystems in a Personal Computer (PC) environment as it assumes reliable links and offers higher performance.

AFM is not an HPC appropriate feature as it creates caches of filesets that are asynchronously maintained with the home fileset location. AFM is engineered to prioritize creating global namespaces of filesets rather than getting the most performance. Some of the filesets in the namespace can be read-only.

By default all nodes in a GPFS storage architecture perform the same functions. Thus we describe the GPFS architecture by enumerating the different functions in GPFS and their relationships to each other. A general overview of the system structure is shown in Figure 2.4.

GPFS Components

GPFS Cluster Manager There is one cluster manager per cluster. This node is chosen internally through an election amongst the quorum nodes. The cluster manager has many important duties including:

- Monitoring disk leases,

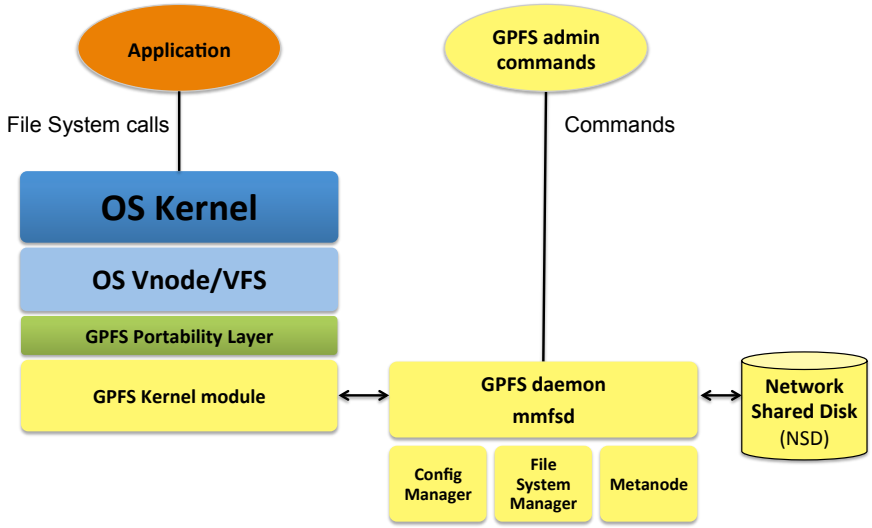


Figure 2.4. System structure for GPFS (adapted from figure in source: [58]).

- Detecting failures and managing recovery from node failures,
- Determining whether a quorum exists so that the GPFS daemon can start,
- Distribute configuration changes to other nodes in the cluster, and
- Manage UID mapping from remote clusters.

Filesystem Manager There is one filesystem manager per filesystem. This node is chosen by the cluster manager and manages all the nodes mounting the filesystem. The primary functions of the Filesystem Manager are:

- Managing adding disks to the filesystem,
- Changing disk availability,
- Managing all mount and umount requests for the filesystem.

Token Manager Server The Token Manager may be the filesystem manager. The Token Manager coordinates access to files on shared disks by granting tokens that convey the right to read or write data or metadata of a file.

Metanode The metanode handles file metadata. Unlike other storage architectures the metanode for a file is determined on a per-file basis and then on which node in the GPFS cluster has recently opened the file for the longest period. Thus over the lifetime of a file the metanode is dynamic. The distributed nature of metadata handling lends itself to good scale-out and metadata performance capabilities.

Network Shared Disk (NSD) Disks containing user data are called Network Shared Disk, NSD. Upon joining a filesystem each disk is tagged with a filesystem descriptor to uniquely identify it belongs to the filesystem. NSD is also the name of a protocol for network access to the disks. The NSD protocol must also be running on the nodes to which the NSD logical units (LUNs) are attached.

Quorum Server The quorum procedure is used for preserving data consistency in GPFS. Quorum means one plus one-half. When a majority, quorum, of nodes are communicating in a cluster then core cluster functions like filesystems mounts can proceed. This scheme prevents filesystem corruption by preventing nodes that have become cut-off from the rest of the cluster from performing filesystem mounts and data access.

2.5 Security Classifications

The "Orange Book" [29] 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.[3]

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

[3]Note, Appendix A includes details related to security "Protection Levels" from the DCID 6/3 manual [27].

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.

2.6 Supporting Security Technologies

2.6.1 NIST

National Institute of Standards and Technology (NIST) is a government body that "develops and issues standards, guidelines, and other publications to assist federal agencies in implementing the Federal Information Security Management Act (FISMA) and in managing cost-effective programs to protect their

information and information systems [34]".

2.6.2 FIPS

Federal Information Processing Standards (FIPS) are issued by the National Institute of Standards and Technology. FIPS are "publicly announced standardizations developed by the United States federal government for use in computer systems by all non-military government agencies and by government contractors, when properly invoked and tailored on a contract [121]".

2.6.3 GSSAPI

The General Security Services Application Programming Interface (GSSAPI) is a general application programming interface (API) for security schemes [119]. The GSSAPI allows security vendors to write GSSAPI compliant security modules which is sure to work with clients respecting the standard. A major benefit of the GSSAPI is that it removes the need for vendors to know details specific to the client. For example, the popular Kerberos [86] security mechanism is often used via GSSAPI to ensure a consistent API due to variations in Kerberos implementations.

2.6.4 Kerberos

Kerberos is a security system that provides authentication in a distributed system [86]. The Kerberos authentication system is widely used by many other systems, to include commercial products like Windows, GPFS, Lustre, etc. Kerberos itself does not provide authorization, but can be used to build authorization in other services [86]. Briefly, a client contacts a Kerberos authentication server and once authenticated obtains a *ticket*. This ticket contains cryptographic keys that ensure the identity of the authentication server and client, so that when the client speaks to a service it can transmit the $client_{ticket}$, which can be verified based on the established keys known within the infrastructure. A key element to this process is that the $client_{ticket}$ contains all the information needed for the service to verify the identity of the client and the authorizing server, which enables the security protocol to be more efficient by reducing the amount of communication required to authenticate a client [86]. A more thorough description of the Kerberos authentication service is provided in [86] and RFC-4120 [87].

2.7 System Management Tools

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 abstraction 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.

In some instances lower-level configuration management tools, e.g., Puppet, can be used to fill gaps in higher-level tools or aid in site customizing procedures. Therefore, we provide a short description for general background. We include a short description of OpenStack, which is detailed more fulling in a later chapter.

2.7.1 Puppet

As a very flexible and full-featured configuration management system, Puppet [60] 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 background 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 features 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 [25] 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.

2.7.2 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 report.

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 APIs 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.

Chapter 3

Virtualization

Virtualization dates back to the 1960's with research performed at IBM in conjunction with their large time-shared systems [24, 42]. In these environments the resources were prohibitively expensive, such that the resources needed to be shared among users. As noted by Goldberg [41], 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 [51, 61, 117, 125]. The early IBM VM/370 systems included additional hardware support for virtual machines [24]. The recent resurgence of interest in virtualization [38] has led to hardware enhancements to support virtualization on commodity architectures (e.g., Intel [116], AMD [2]).

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 [24]). The approach has received renewed interest to enhance security (trusted computing base), improve utilization (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
≥2.4.19	mnt	mount points & file systems to be isolated, (i.e., file system mounts in one namespace are hidden from another namespace)
≥2.6.19	ipc	Inter-Process Communication mechanisms within namespace
≥2.6.19	uts	hostname and domain name separate from values at host
≥2.6.24	pid	process isolation between namespaces
≥2.6.29	net	isolates the network devices and network stack
≥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 9.2) 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 process'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 9.2. 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 [17]. 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 [17].

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 [17]. All query and modify operations are done via this `cgroup` file system [17], 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 [17] 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 [104].

```
1  mount -t tmpfs cgroup_root /sys/fs/cgroup
2  mkdir /sys/fs/cgroup/cpuset
3  mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
4  cd /sys/fs/cgroup/cpuset
5  mkdir Charlie
6  cd Charlie
7  /bin/echo 2-3 > cpuset.cpus
8  /bin/echo 1 > cpuset.mems
9  /bin/echo $$ > tasks
10 sh
11 # The subshell 'sh' is now running in cgroup Charlie
12 # The next line should display '/Charlie'
13 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 [17].)

3.1.3 Linux-VServer

Linux-VServer [64] 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 [97] 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 [25], 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) [71] 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 B 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 [8] 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) [47, 63] 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 [116] and AMD-v [2].

KVM operates in conjunction with supporting user-level tools found within QEMU [9]. 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 [106]. 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* [10] to include a pluggable security framework [82]. sVirt can be used to put a "security boundary around each virtual machine" [96, Ch.15]. VM or VE processes and disk images are labeled 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 [3].

3.3.2 SELinux

SELinux [108] is an implementation of the Flask [110] 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 [5] is derived from the SubDomain project that dates back to 1998/1999 [4], and was rebranded as AppArmor after Novell acquired the work in 2005 [4]. 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 [4]. AppArmor uses the Linux Security Module (LSM) interface [23].

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) [5]. The program profiles are path-based. The system is intended 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 [5]. 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 latest Red Hat release (RHEL7) but the RPMS from openSUSE may be usable. AppArmor has also been integrated with libvirt [3] 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 [16]. As of Linux 3.17 the kernel has 37 such units. A thread possesses 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 [16]. 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 [114] 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.

Chapter 4

Reconfigurable Networks

There are a variety of methods for dynamically reconfiguring the network, each with different challenges and limitations. This section contrasts typical networking techniques with virtual networking methods.

4.1 Typical Networking Environment

In the typical environment, the system is analyzed and requirements documents are generated to realize the needs of the application. Initially the Layer 3 appliance is built out in the appropriate location and then specific security policies are added. The Layer 2 structures are built out depending on customer needs. In a rapidly changing environment, meeting the Layer 2 change requests presents a challenge. If a customer requires additional Layer 3 instances on top of the requested Layer 2 functions, it can force a complete redesign of existing security policies. Additionally, the physical connections must be considered for the deployment, taking into consideration:

1. *Are the tenant nodes connected?*
2. *Is there a need for external WAN and Internet connectivity?*
3. *As the system grows, how will load balancing be handled?*

The primary concern with traditional networking deployment models is that the physical connections have to change as rapidly as the user's needs change. This means a considerable amount of time is spent reallocating existing physical connections and facility (space) resources as well as the down time associated with reconfiguring the existing equipment or adding new appliances.

4.2 Static Networks Involving VRF and Preconfigured VLANS

The availability of preconfigured static Virtual Routing and Forwarding (VRF)s and corresponding Virtual Local Area Network (VLAN)s allows for tenants to be placed into separate service areas, where all traffic is carried between physical virtualization servers using different VLANs. This topology works well if the network is rigidly defined with fixed connections and port definitions. Access to external servers and infrastructure is handled through connection policies maintained directly on the server. This method does not allow the tenants to run services in an environment where inbound and outbound filtering is applied. This is especially true if each customer has unique and frequently changing network requirements. As the

requirements change, the Access Control List (ACL)s, firewall rules, and both physical & virtual switch policies and configurations must change as well. Altering connectivity requires updating all of the traffic defining policies and connection information.

4.3 Software Interfaces for Reconfigurable Networks

A common method used in reconfigurable networks is deployment automation. This methodology implies that all network appliances are managed by a centrally control system. The network administrators push out bulk changes and policies across the entire network without having to configure each device manually. In a secure environment, after verifying the correct permissions, tenants can request additional resources and these requests are pushed through the central network management system. The central management system configures the available resources with appropriate security policies and connection rules and then pushes out these changes where they are needed. All SDN methods have a need for a common interface to abstract the physical connections from the vendor specific (physical) device [7]. As each vendor uses a combination of proprietary hardware and appliance OS the associated application software commonality is accomplished through a vendor specific application program interface (API).

The vendor API interacts with the OpenStack service software and acts as a common interface to translate SDN functionality into direct corresponding functions on the vendor hardware. The resulting control interface approaches the desired universal control layer envisioned by the virtual system designers without sacrificing the capabilities of the individual hardware. Additionally this method allows, within some parameters, the multiple vendor deployment capability desired by server farm administrators. This capability further eliminates the need to have one specific switch appliance vendor throughout the facility. Systems are streamlined in this virtual common platform approach. As an example, with a large group of tenants, this allows the central management system to limit the networks carried over a Layer 2 trunk to the physical virtual servers, which saves bandwidth by removing unneeded broadcast traffic. The existence of common connectivity templates in place allows each tenant to securely operate without interacting with other tenants. Further, access to external networks is provided to the tenant systems, while maintaining existing connection and security policies. Network access APIs are vendor specific, with only common functions called out specifically in the OpenFlow standard. In many cases vendors add functionality to increase performance, statistics or reliability outside of the standard in an effort to entice the use of their hardware. Additionally since switch fabric bandwidth and connection agility change the vendors port configuration, vendor syntax will often be different. The syntax for shutting down and reconfiguring ports will vary from vendor to vendor on the command line interface. The use of a vendor API allows the central management system to communicate with the physical appliance over a controller attached port, using a common control language. This allows the central control authority to authenticate and then make any necessary network configuration changes. The OpenFlow based API can make all changes at all layers, once the established security policies have been met. In addition to configuration APIs, some network appliances support third party configuration management software such as Chef, or Puppet. In most cases APIs will use a JSON like interface for human readable code and configuration data, but will have commands that are unique for their specific capabilities and platforms.

4.4 Traditional SDN

In a traditional SDN, there is no implied intelligence on the network appliance as all decisions with respect to the control plane and data plane originate from a logically central control authority. While this

methodology offers flexibility, it does not however scale well, representing a recognized single point of failure. The use of traditional SDN through a single central controller works well for small deployments with high flexibility and low availability requirements [124]. Some of these limitations can be addressed by employing a clustered central controller architecture with an active/passive or active/active failover strategy. There are SDN architectures that adopt an active/active centralized controller architecture to address both scalability and resiliency requirements.

4.5 Hybrid SDN

Hybrid SDN uses a separate control plane like traditional SDN, but network devices also maintain control plane functions independently. In this model each network device still functions independently from the central controller, but also receives configurations from the central controller. The central controller handles traffic by reconfiguring the individual nodes as needed by each device. This allows a simple method for dynamically handling traffic hot-spots.

Applications can talk to the central controllers via API to get network health, or to make provisioning changes leveraging the abstraction concept. In a hybrid SDN control responsibility is both shared and dispersed, losing the central controller does not result in the loss of the entire network, only the management and configuration is crippled until the system is repaired. Individual network appliances can still be configured if central control is lost, the network is still capable of running the current applications with a temporarily frozen configuration and policies rule set. The hybrid SDN model scales better, and maintains high availability.

4.6 Overlay Network

Network overlays are accomplished by using tunneling or encapsulation techniques. This allows the extension of the network at Layer 2 from one location to another, increasing flexibility in terms of scaling the network as large as needed. The use of overlays also overcomes some of the intrinsic limitations of network appliances such as the 4096 VLAN limit. Overlay networks are beneficial in a data center environment due to low latency, higher bandwidth, and increased control over bandwidth utilization. Additionally, overlay methods extend Layer 2 networks across Layer 3 boundaries, either within the data center, or across WAN links. Keeping this local to the data center allows additional control flexibility unless dedicated paths are required. For example, if Service Level Agreements (SLAs) exist to a certain path across a provider's network. Overlay networks can extend across the WAN interface to other data centers as long as the connection is compatible in performance such as bandwidth, latency, and jitter. Essentially the network overlay is a network built on top of an existing network structure. Connectivity is accomplished through the creation of network tunnels, requiring endpoints within both connected domains which are configured to allow traffic transferred across the tunnel appearing as a contiguous Layer 2 domain.

The necessary overlay endpoints can be created manually, or via APIs. Similar overlay methods found within the data center are implemented by using encapsulation methods such as Virtual eXtensible Local Area Network (VXLAN). Protocols like VXLAN allow you to create virtualized Layer 2 networks across different Layer 3 networks and can scale up to 16 million logical networks.

4.7 SDN with OpenStack

OpenStack uses a hybrid SDN approach where the network appliances are considered stand-alone devices and function as separate entities from OpenStack. It is possible to have all Layer 2 and Layer 3 traffic preconfigured statically on the individual network device. In this configuration, OpenStack handles traffic between tenants. Some network vendors support OpenStack plugins that allow OpenStack to make port and VLAN configuration changes as part of their OpenStack interface API. OpenStack includes a network control node application called *Neutron* that facilitates SDN networking accomplished using the internal OpenStack routing engine for both inter- and intra-VLAN traffic. Neutron has the capability to communicate with the network via dynamic routing.

4.8 Implementing Neutron Routers

The use of multiple flat networks require bridge interfaces for each network connection, the addition of VLANS further complicates the setup by requiring switch and gateway configuration per instance. Neutron contains a plugin agent specifically to handle L3 connectivity. This agent allows both administrators and tenants to create routers that handle traffic between directly connected tenant network interfaces, either Generic Routing Encapsulation (GRE) or VLAN, and a single management or controller network node. Access to external provider networks, including WAN services are handled through this Neutron router structure. The external network is typically implemented as either a FLAT or VLAN provider network.

Nova compute nodes use both fixed and floating IP addresses. The fixed IP addresses are assigned to the compute instance on creation, and remain until the instance is terminated. Floating IP addresses are dynamically associated with the instance as needed. Floating IP addresses can be associated or disassociated with a instance at any time. A public or provider network involves a connection that is potentially outside of Neutron control. In a Nova network the use of 1:1 NAT translation allows for a customizable "floating" IP address implementation, it is common for the same IP that is used as the L2 address to also be used in the bridge to the hypervisor. This is accomplished by using the `iptables` configuration on the host by modifying the Source Network Address Translation (SNAT)/Dynamic Network Address Translation (DNAT) rules. Re-association of a floating IP address is accomplished by removing the rule from the `iptables` SNAT/DNAT rules list and re-associating on another instance, in this way the instance IPs remain static, only the NAT rules change.

Neutron routers act as gateways for each tenant instance using the Neutron L3 agent, instead of manipulating the `iptables` on the hypervisor. The `iptable` in the router handles the NAT translations, by instantiation of connections to Virtualized Network Interface Control (VNIC) devices connected to its ports. The floating IP addresses are procured from the provider network through pre-determined tables or using the Neutron DHCP agent. Containers[1] or VE can be instantiated without worrying about using redundant IP addresses on the same networks nor requiring the user to reset or manually load tables as part of a start up script. Access to the node within a container is only granted by using the network ID (namespace) and setting the connection in the routing tables. Attempts to access without proper credentials can be tagged and monitored easily in this configuration. This method limits the floating IP addresses to that of the WAN address space. The MAC addresses of the tenant NICs can have fixed IP addresses in the NAT tables as well as be associated with defined security group IDs. The Neutron L3 agent should be present on both the network and controller node. Once a container is established only the compute node

[1] Also referred to as Virtual Environments (VEs) in the other project report titled, "Review of Enabling Technologies to Facilitate Secure Compute Customization."

within the container have unfettered access to each other and controlled access to the external network. Nova nodes are simply added to the table in the router as created, no other management action is necessary. This also applies to VMs that perform other functions including additional routers for separate internal networks.

The use of routers in Neutron is possible using existing technology, however it is a fairly new development. Bottlenecks have been observed in the layering necessary to perform the function as it is currently being built up with existing software blocks rather than implemented as a stand-alone function. The redundancy factor is higher than normal to achieve otherwise simple NAT pairings. Preliminary reports suggest it is possible to have near zero latency network within a LXD structured environment. [2] The list of current considerations for implementing virtual routers is given below.

A. When configuring the L3 agent using the agent config file, specifying an external network bridge, causes Neutron to associate the external NIC directly with the bridge. The attributes for "vlan" "segmentation ID", and "provider network" are ignored, Neutron assigns an IP address to its translation table from the provider network.
B. The gateway can be manually specified using the `gatewayexternalnetwork_id` attribute, otherwise Neutron looks for the gateway from the provider network if the attribute `external=true` is set otherwise, Neutron will stall if gateway not found.
C. If an external bridge is not set, Neutron uses the external interface into the Open Virtual Switch (OVS) bridge specified by the provider network from the Neutron Controller. Any subsequent network traffic is handled through the Open VSwitch flow rules present in the controller. This is the typical interface for controlling VLANs through OVS.
D. Traffic within a GRE based tenant network is limited to that network only, bridging is now through the router.
E. The Neutron router will allow directly connected tenant networks to communicate with each other freely, and the external provider network only if the router rules allow the connection. All tenant nodes are behind the Neutron router, and no longer have floating IP addresses, therefore there is no direct connection to them outside the Neutron router or within the DHCP namespace instance.

A test network (Figure 4.1) is proposed as a sandbox test using a Network Node, and Network Controller Node, and a Nova Compute Node. In the test there is only one Nova Node but more are possible, the container must be limited to one network Node and one Network Control Node however. All nodes have both Neutron control Agents and the OVS agent running, the controller node does not require the OVS agent. The Open vSwitch plugin can be replaced with a proprietary Neutron Switch interface provided by the physical switch manufacturer. The provider network is modeled using a network node instance. Connections to the networks are through bridging (br-ext) and using GRE tunnels (br-tun) set in the OVS router configuration file.[3] Internally the VLAN interfaces are configured using bridging adapters configured using Neutron as shown in Figure 4.2.

[2] The beta release of LXD in Ubuntu 14.04 OpenStack and the re-writing of most of the access agents in Neutron is well underway.

[3] Note, this describes a general OpenStack/Neutron test scenario that would use GRE tunnels. However, in the primary SE testbed running on the CADES hardware at ORNL, we do not use GRE tunnels, rather we use VLANs programmed via the SDN interface to the Arista switch that is exposed via a Neutron plugin. Thus the "Not in CADES" annotations in Figure 4.2.

Neutron OVS Configuration

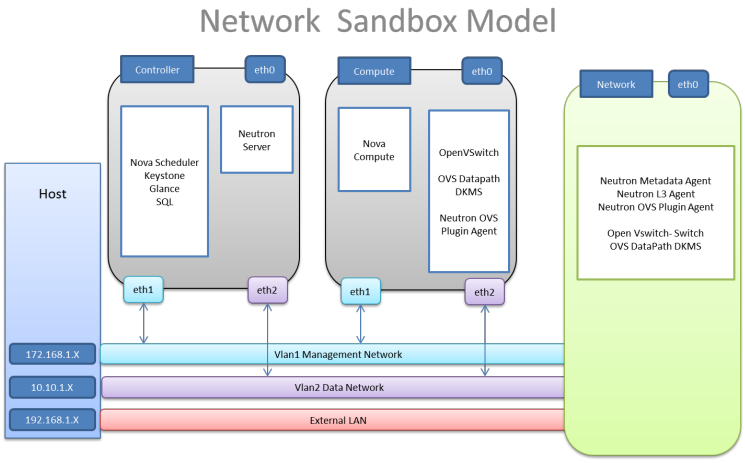


Figure 4.1. Neutron OVS SDN Router Configuration.

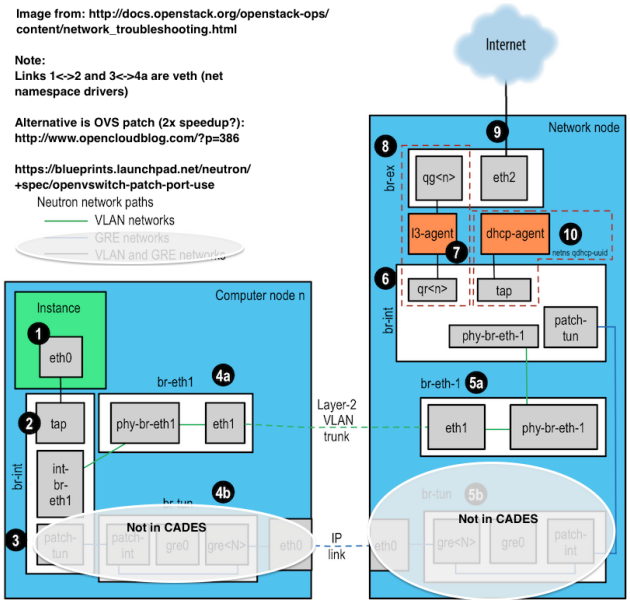


Figure 4.2. VNIC interface configuration.

4.9 Networking with LXD

LXD [72] is an extension of the successful LXC [71], which includes the use of Docker [35] support and similar services in establishing containers. It was envisioned by OpenStack Canonical to have the virtualization environment where LXC is the client support and LXD is the server. LXD will allow secure containers set ups for Linux based compute nodes, there is no support for other operating systems. Further the LXD/LXC pairing eliminates the redundant bridge structures that cause intrinsic delays in packet delivery by replacing the need for additional structures to perform the routing function directly within the host. The routing between containers can be enhanced using the LXD manager, which is aware of the end-points (containers) and hosts, to avoid adding redundant bridges.

Chapter 5

Security in HPC Storage

5.1 Lustre

To employ Lustre in a secure enclave model we outline its security features. Lustre release 2.7 adds an important capability, dynamic LNET configuration, which we leverage to automate the isolation of environments sharing access to the filesystem. Another upcoming feature which enhances Lustre's authentication capabilities, is GSSAPI support. Since Lustre is POSIX compliant, file locking mechanisms promote serialized file access and POSIX ACLs provide user-managed authorizations. Administrators of Lustre can employ root-squash[1] to restrict root clients on the mounted filesystem. There are other features under development for the Lustre 2.8 release scheduled for Q2 2015 [65].

5.1.1 Isolation

A Lustre server can communicate with clients and servers via the LNET API over one or more network interfaces. The common use case is a single LNET on a single physical medium, on which the server can reach all other servers and all clients. However, the identifier Lustre uses for each network node could specify a different network interface (NI) (e.g. `@tcp2` instead of `@tcp1`, where each are in different subnets). If the server has LNET configured on both `@tcp1` and `@tcp2` NIs, then it could reach a client `172.20.0.1@tcp1` as well as `192.168.0.100@tcp2`. These NIs can be separate physical interfaces or different VLANs, but in each case traffic between the two NIs is isolated. Only servers that are dual-homed on both NIs have visibility into both network segments.

This has been a feature of Lustre to have so-called "multi-rail" LNET configurations, but until Lustre 2.7 with the *Dynamic LNET Configuration* feature, it has not been possible to make changes to add or remove an NI from a server without taking the entire filesystem down. Now a command-line tool `lcmd` and a YAML specification can be used to add and remove these interfaces on-demand.

5.1.2 Authentication

GSSAPI New authentication mechanisms can be added to Lustre through its support of the GSSAPI [119]. Currently support for Kerberos is included this way and soon a shared host-key mechanism is due for release.

[1] The "root-squash" option restricts a local `root` user's permissions from being applied to the remote system providing the share, i.e., squashing local `root` permissions from transferring to the remote system.

Kerberos According to OpenSFS, the organization currently responsible for ensuring that Lustre "remains vendor-neutral, open, and free [92]". Lustre's support for Kerberos is in some disrepair [120]. An older work describes Kerberos intended uses in Lustre. Kerberos allows mutual authentication amongst clients, OSSes and MDSes. It also provides both privacy and integrity for `PTLRPC` messages. All entities, users and services alike, are represented as principals to the Kerberos server. Each principal shares a secret key with the Kerberos server that allows them to verify messages from the server. Kerberos implementation is simplest if the Key Distribution Center (KDC) and all the other services share the same user database [98].

Shared Key Authentication Shared Key Authentication and Encryption in Lustre is currently in development and is expected to be completed in Lustre version 2.8. This mechanism will provide host-based authentication and encryption and will use Lustre's existing support for GSSAPI. In this scheme a single key is generated for each client and is installed on client and server [30]. This key is created using a notion of cluster ID, "a string used to uniquely identify a cluster." Data integrity will be provided by creating a message digest, HMAC, of each message or block of data. The keys used to create this HMAC will be obtained from the Linux keyring. The scheme proposes to use userspace tools to create the keys and LNET control utilities, `lctl`, to make the keys available to Lustre. For encryption, a Diffie-Hellman key exchange will be performed to generate a per-session encryption key. Lustre's `PTLRPC` will be tasked with performing the Diffie-Hellman key exchange.

5.1.3 Authorization

POSIX & ACLs Lustre employs a POSIX compliant UNIX filesystem interface [45, 93]. The full suite of POSIX tests completes on a Lustre filesystem just as they do on a regular `ext4` filesystem [93]. This means that regular UNIX users encounter familiar filesystem interfaces and behaviors and can almost immediately begin using a Lustre filesystem.

Root-Squash Since version 2.6 Lustre has supported "root-squash." This is the ability to specify to which local UID/GID should `root` on an accessing client should be mapped [98]. Thus filesystem administrators can apply arbitrary restrictions on clients accessing the filesystem as `root`.

5.1.4 Integrity

Lustre can provide data integrity checks by computing checksums on data [93]. A 32-bit checksum for data read or written on the client and server is computed to guard against corruption in transit. Alder32 and CRC32 are amongst the common algorithms utilized. It should be noted that the backend filesystem does not do any persistent checksumming and so cannot determine whether data residing on disk is corrupted.

5.1.5 Features in Development

Two features currently under development that could add to the overall security of a shared Lustre filesystem are UID/GID mapping scheme and support for clients to mount sub-trees of the filesystem.

Developers from Indiana University have put forth a plan for implementing a nodemap scheme for UID and GID mapping within Lustre funded by an OpenSFS grant [109]. It's functionality was demonstrated with Lustre 1.6 and 1.8 releases, but work is under way to bring the code up to date for a Lustre 2.8 re-implementation. There have been patches submitted for review [68], but the development is not far

enough along that we could perform an evaluation of the feature. Briefly, the nodemap defines a relationship between NID ranges (clients) and UID/GID maps. The nodemap is distributed via LNET to each Lustre OSS and MDS for enforcing what system IDs map to IDs on the filesystem based on the NID of the client. This is not unlike our proposed use of user namespaces to map the UIDs/GIDs within a container to IDs on the host. The combination of these two mapping techniques would provided a layered approach to securely isolating UID/GID ranges. For example, user namespaces can map the `root` user within a container to a normal user on the host, where the nodemap defines what IDs on the host are allowed to map to filesystem IDs. This could potentially limit the filesystem access rights of an adversary if a VE host were to be compromised.

The ability for a client to mount a subdirectory of a Lustre filesystem is proposed in a patch currently under review [69]. This patch proposes to add a capability similar to NFS, where instead of mounting the Lustre root directory, the client could choose to limit the filesystem namespace that is exposed to a subdirectory of the Lustre filesystem. It is true that this doesn't add any security in enforcing isolation at the client level because if the client is capable of mounting a subdirectory, then there's nothing preventing the client form mounting the root directory instead.

5.1.6 Gaps

Having discussed the security features of Lustre, we note significant gaps that appear in the current Lustre release. The first is the absence of server-enforced subtree mounts, instead granting full namespace access to client. There is not a method to limit the subtrees of the filesystem to export to specific clients. This leaves little protection for data on the filesystem if an adversary has escalated to root on any one client. We view this as significant component of shared filesystem security, and we explore alternative ways to mitigate this risk with Lustre in Section 6.

A second gap is the lack of encryption support at rest. Encryption of data in flight is made possible by the GSSAPI support in Lustre, however, it is not likely tested enough to be used in production. Furthermore, objects and metadata are stored in unencrypted `ldiskfs` (a variant of `ext4`) format on the the OST devices. Other non-native techniques would be needed to achieve encryption with Lustre.

As evidenced by the frequent appearance of Lustre on Top500 lists, the focus has been more on the scalability and performance aspects rather than security features seen in other filesystems targeted at the "enterprise-class market."

5.2 GPFS

GPFS is a storage architecture rich in security features. It is POSIX compliant; GPFS implements file locking algorithms that ensures serialization of file updates and uses POSIX ACLs authorizations to manage file access. GPFS also provides both authentication and encryption between clusters owning a filesystem and cluster wishing to mount that filesystem. Apart from encryption in authentication, GPFS also provides encryption for "at rest" files on the filesystem. Below we explore these features in greater detail.

5.2.1 Authentication

GPFS supports mutual cluster authentication and authorization in Multicluster [52]. This allows distinct GPFS clusters to authorize and authenticate each other and then share filesystems. The cluster

owning the filesystem must explicitly grant access to other clusters wishing to mount that filesystem and also explicitly grant access for the specific filesystem to be mounted. On the other hand, clusters wanting to mount a remote filesystem must define the cluster owning the remote filesystem as well as the filesystem it wishes to mount. For this GPFS uses RSA authentication; each cluster generate key pairs then exchanging their public keys. GPFS also supports client clusters at multiple security levels[52]. In this model key pairs of the appropriate strength are exchanged with the different clusters[52]. It is important to note this authentication is between two clusters so nodes within each cluster use the cluster keys for authentication. In addition, this mechanism does not include user authentication[52]; it is assumed the users authenticate to the operating system of the client node by means external to GPFS.

5.2.2 Authorization

POSIX & ACLs GPFS is fully POSIX compliant [1]. Locking, POSIX ACLs and other shell utilities makes the GPFS filesystem experience very similar to a standard Linux filesystem installation.

Kerberos in GPFS Kerberos can be used in combination with SSH as a means of authenticating administrative commands in GPFS [113]. However, it appears to have limited applicability in authenticating clusters to each other. In addition, Kerberos can be used to authenticate users at login into nodes belonging to a GPFS cluster.

GSSAPI in GPFS In GPFS the GSSAPI seem mostly confined to usages within SSH authentication of clients to nodes belonging to a GPFS cluster at login and not as a means of authentication between GPFS services[52].

Root-Squash in GPFS Root squash in GPFS is the ability to map a root user on a client mounting the filesystem to another user with little authorization on the filesystem. This is a restriction imposed by the cluster owning the filesystem and not a device of any UID mapping application nor is it a function of the client cluster mounting the filesystem.

Multicluster Authorization GPFS multicluster requires both filesystem cluster and client cluster various authorization episodes. First, each cluster must authorize participating in multicluster sharing. Then each cluster must authorize connecting to each other; the cluster owning the filesystem must authorize clusters mounting the filesystem and then authorize each connecting client to mount a specific filesystem. Similarly, the client cluster must authorize the cluster with the filesystem and the particular filesystem [52].

5.2.3 Encryption

GPFS supports on-disk encryption in GPFS Advanced Edition and then only in the latest version of the 4.1 filesystem. Encryption is managed through keys and encryption policies and supports several different ciphers. This encryption only applies to data and not metadata. GPFS also advertises secure deletion where data is effectively inaccessible because the encryption keys are deleted from the filesystem.

Encryption from authentication coupled with the on-disk encryption effectively provide end-to-end data encryption in GPFS. Encrypted files at rest on disk are transported through the wire in its encrypted format, to be decrypted in the memory of the client mounting the filesystem. GPFS also supports encryption in a multicluster environment.

Table 5.1. Lustre vs. GPFS

Feature	Lustre	GPFS
Authentication	yes	yes
Encryption in Authentication	yes	yes
On-disk Encryption	no	yes
Subtree mounts	no	no
POSIX Compliant	yes	yes
User authentication to storage	no	no
Kerberos Support	yes	other*
GSSAPI Compliant	yes	other*
Performance	good	good
Scalability	good	good

* denotes that feature is not applied directly to GPFS processes.

In GPFS, Master Encryption Keys (MEKs) are used to encrypt File Encryption Keys (FEKs). FEKs encrypt portions of a file when the file is first created. The FEK is stored, in encrypted format, in an attribute of the file. MEKs are stored on Remote Key Management Servers (RKMs). The RKMs also contain other encryption and policy information.

All GPFS security mechanisms are NIST compliant. To ensure other compliances like FIPS 140-2 or NIST SP800-131A GPFS uses variables, like `FIPS1402mode=yes`. These variables must be set before generating the key-store.

5.2.4 Features & Gaps

GPFS possesses attractive protections. Its authentication and encryption capabilities as well as the "at rest" file encryption capabilities make GPFS a highly secure storage architecture. GPFS also lacks subtree export control capabilities in its NSD protocol thus client cluster mounts cannot be restricted to a subtree of the filesystem. This is a serious shortcoming.

5.3 Discussion

5.3.1 Comparisons of Security with Lustre and GPFS

While both the Lustre and GPFS storage architectures produce highly performant scalable filesytems, there are significant differences in their security capabilities (Table 5.1). Both suffer from the inability to only export subtrees of their global filesystem. This is a significant shortcoming as it means that any client that mounts one of these filesystem mounts all the data on the filesystem. This an aspect that we discuss in Section 6, where modern OS technologies can be used to provide an additional layer of isolation between end-users and access to the root filesystem namespace. Both GPFS and Lustre support GSSAPI and Kerberos but in different manners. In GPFS, Kerberos functions to authenticate users for login to systems at the OS level and possibly to authenticate administrative commands. However, in Lustre, Kerberos is used for authentication amongst Lustre specific services. A significant difference between the security stance of the two filesystem is that GPFS natively supports data encryption "at rest," while Lustre does not. Another area where there are significant differences is authentication amongst storage specific

technologies. GPFS authenticates pairs of clusters to allow filesystem mounts whereas Lustre authentication is host-based. So client node wanting to mount the Lustre filesystem would have to be authenticated against each Lustre service. Pointedly lacking in both storage architectures is end-user authentication to either Lustre or GPFS specific processes.

5.3.2 Performance in Lustre and GPFS

On at 18PB system with 5000+ servers using GPFS v3.4 has been documented to achieve 240GB/sec [44]. No further details were given on the setup. The Lustre Spider filesystem at ORNL has been documented to produce about the same throughput [99]. This paper predicts that the next generation of Lustre filesystem will reach 1TB/s [102].

Chapter 6

Bridging Technologies for Secure Storage

There are several technologies that may be useful for bridging current gaps in HPC secure storage. The methods we highlight are generally focused on introducing isolation or protection mechanisms that can be leveraged to overcome voids in current HPC storage technologies.

6.1 Virtualization

The virtualization capabilities in enclaves can be leveraged to overcome certain issues by carefully applying the available isolation mechanisms. We briefly highlight some of the more interesting capabilities that can be of use for bridging current gaps in secure storage. Note, more details on isolation with virtualization can be found in Chapter 3.

Virtual Machines In hypervisor-based systems, the VM can have virtual devices that do not necessarily match the exact hardware. This can be useful for interposing on a VM's device layer to provide capabilities that are transparent to the guest running inside the VM. For example, if encryption was a priority the data passing from the VM to virtual devices could be encrypted on the fly.

Another example where virtualized devices can be advantageous is when there are changes made to make the interface more efficient by adapting the VM's interface for performance reasons. This is commonly referred to as para-virtualization and is commonly used for customizing the VM's interface to better suite a given use case. For example, virtualized IO can avoid translation in some layers of the software stack because similar work is happening at the system level. Note, this is in contrast to techniques like VMM-bypass, which have the hypervisor and VM setup an interface that allows the VM to have (controlled) direct access to resources without passing through the hypervisor. The `virtio` [106] interface is a standardized API within Linux for creating efficient IO devices for VMs.

Containers & Namespaces The container-based approach to virtualization uses a single OS kernel and adds additional isolation mechanisms for running processes. In this VE-based environment, there are several mechanisms available for limiting access and visibility of the system that can be advantageous for securing storage. For example, the `mnt` namespace provides a kernel level restriction for limiting access to portions of the filesystem. This can be combined with `user` namespaces to allow for controlled mapping of UID/GID privileges within and outside the VE and host contexts. For example, a user may have `root` permissions in a VE but not outside the VE. Additionally, since there is a single kernel in the VE context,

very efficient isolation mechanisms like bind-mounting can be employed to restrict access to filesystems, e.g., restrict a user to a specific region of the filesystem.

6.2 VLAN/Network Segmentation

The ability to restrict access to different portions of the communication network is another mechanism for limiting access and protecting storage. The creation of dynamic network segments, i.e., overlay networks and VLANS, can be used to segment the network to specific hosts and users. These network restrictions may be within the network connecting the hosts, which run the VEs and VMs, such that only the hosts have the ability to make configuration changes to these segments. This would restrict users (VEs/VMs) from seeing each other. Additionally, the restrictions can be within the storage network itself. For example, limiting which hosts may access the Lustre network, i.e., LNET, can limit potential risks to the backing storage network (Chapter 5.1.1). Note, more details on techniques for network segmentation can be found in Chapter 4.

6.3 I/O Forwarding

The forwarding of I/O requests through some intermediate layer or service is a very general method for controlling access. A common method for performing this I/O forwarding is to use a network protocol for marshalling the interactions between a client and server. This is often useful for restricting access to filesystem subtree to limit the namespace accessible by a given user, i.e., restrict mountable filesystem shares. When employing virtualization, the forwarding layer may simply be between a host/guest. When working under a single kernel, the filesystem subtree can be restricted via a combination mechanisms, e.g., bind-mounts, `pivot_root`, and `user` namespaces, as used by containers. However, when working with multi-kernel configurations, e.g., VMs, the sharing must be modified to suit the two kernel environment. The remaining paragraphs in this chapter describe I/O forwarding mechanisms that could be used to retain control of the filesystem tree exposed to the guest.

6.3.1 NFS

Lustre does not have the capability to restrict client mount to only subtrees of the filesystem. This is a very common feature of the NFS protocol and is immensely useful. It also provides a measure of security as it restricts client visibility of the filesystem to anything outside the mounted area. Even if a client on the storage network was compromised, such that the client machine could issue a mount of the entire filesystem, with NFS export restrictions, the NFS server would not allow mounts to succeed outside of the configured subtree for that client's IP address.

6.3.2 VirtFS

VirtFS is a para-virtualized filesystem designed to optimize passing filesystems up from the host operating system through to the guest environment. Popular methods of passing filesystems into a guest are NFS and CIFS[1] Both these options suffer from performance deficiencies and both are unable to capitalize on the virtual nature of the environment. The ingredients for VirtFS are QEMU [9], KVM [63], VirtIO [106] technologies and the 9P2000.L protocol [31, 55]. VirtFS moves away from the traditional

[1]The CIFS protocol is often referred to by the name *Samba*, which is an open-source implementation of the CIFS/SMB server.

notion of creating virtual block devices in the guest environment for the filesystem passed to it and instead passes I/O to memory objects it shares with the host. These I/Os are then relayed to the local filesystem of the host. This minimizes the number places where the same information is cached between client and server and reduces the number of layers through which data must flow between them. A QEMU/KVM server would export part of the hypervisor's filesystem hierarchy into the guest environment where it is mounted using the 9P2000.L protocol. At this point the guest uses the filesystem as if it were local. In reality however, the guest I/O is actually happening on the hypervisor filesystem [55].

6.3.3 DIOD

DIOD is an I/O forwarding server that uses the 9P protocol to share a filesystem [31]. This work is being carried out by Jim Garlick at LLNL and is currently only being tested with NFS filesystems [31]. There is potential for using this with parallel filesystems like Lustre and GPFS but the documentation indicates this has not yet been fully tested [32]. Even with the experimental Lustre support, performance is limited without a patch to the v9fs driver in Linux to increase the packet payload size beyond 64k [33]. Attempts to push this patch upstream were unsuccessful. However, there has been related work that used 9P for I/O forwarding of Lustre, which appears to be using the NFS-Ganesha server with support for 9P [103].

Chapter 7

OpenStack Implementation Details

7.1 Core OpenStack Components

There are several components in the OpenStack framework. They each offer a distinct interface to the functionality for a distributed computing environment. Figure 7.1 illustrates the different elements in the OpenStack architecture and shows the interactions between the various component interfaces.

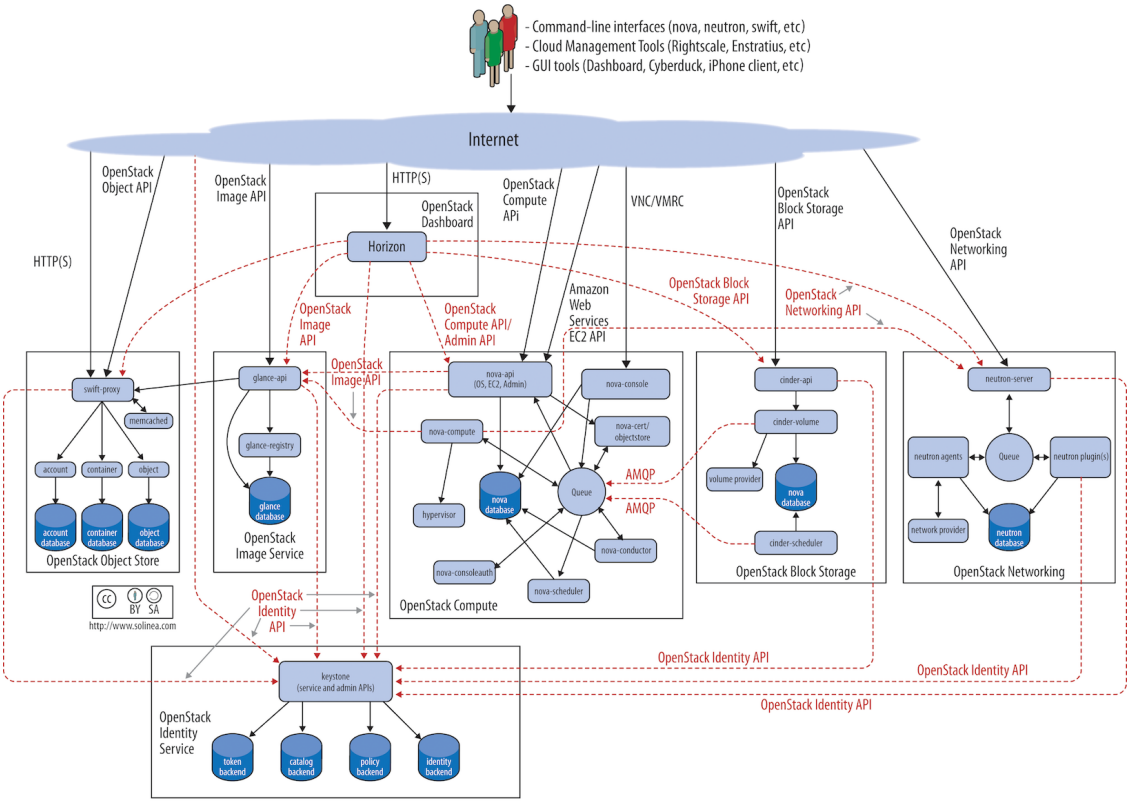


Figure 7.1. OpenStack Logical Architecture (source: [95])

7.1.1 Horizon – Dashboard

A user of OpenStack has a choice of interfaces with the various components. Commonly service interactions are run through a command-line interface that generates the appropriate API calls, to Nova for example. Alternatively the graphical interface known as Horizon can be used as a dashboard for a user (tenant) or administrator. Per-tenant instances can be managed and currently used resources can be tracked using Horizon. Like the CLI, Horizon interacts with individual OpenStack services via the REST-based API.

7.1.2 Nova – Compute

Nova is the compute service of OpenStack and is the primary interface for users to create/start/stop/migrate instances. Given a set of nova-compute nodes in an OpenStack cluster, the nova-scheduler service will make placement decisions of VM's on particular nova-compute hypervisors. The nova-compute nodes are typically running KVM and hosting virtual machines, but alternative nova-compute drivers can host containers (nova-compute-lxd) and even coordinate the booting of baremetal nodes (nova-compute-ironic)

7.1.3 Neutron – Networking

Neutron is a complex service in OpenStack handling the networking between tenant VM instances and network services such as DHCP, routing, load balancing, and the metadata service (for assisting in booting nodes). Note, Neutron is also discussed in the reconfigurable networking section (Chapter 4.7).

7.1.4 Keystone – Identity Services

Authentication and authorization of users in an OpenStack environment is handled by the Keystone service. Keystone maintains roles and project memberships of users and provide a mechanism for authenticating each API call.

7.1.5 Glance – Image Service

Glance is the name of the service that manages VM images. Each image is stored in the glance-registry for retrieval from nova-compute when a VM boots.

7.1.6 Swift – Object Storage

OpenStack [94] contains an object storage service known as "Swift" [111]. The purpose of which is to provide massively scalable, redundant storage across commodity hardware platforms. The basic model for this service is along the lines of Amazon S3 storage severs and is managed in similar fashion. The redundant nature of this storage server allows for VM instance templates used by OpenStack compute nodes.

Object Storage Support Structure OpenStack uses many services to complete a storage environment setup. The following is a list of services necessary to run the object storage service in an OpenStack environment:

- Swift: This service handles all of the common files shared between other OpenStack object storage servers and packages, including the Swift client itself;
- Swift-Proxy: The proxy service is the outward facing component that clients connect to.
- Swift-account: The account management service clients use to gain access to OpenStack Storage;
- Swift-Object: Service package that manages object storage and *rsync* file transfer synchronization methods;
- Swift-Container: The package that handles the OpenStack Object Storage Container Class and server;
- Swift-recon: Middleware used on a Swift server node to collect access and usage statistics;
- Memcache: Memory object caching system;
- Network time protocol (NTP): Protocol used in multi-node environments to synchronize the nodes such that the transfers, snapshots and cache scheduling can all be synchronized;
- Xfprogs: This module controls the XFS filesystem used in many OpenStack object storage systems.

OpenStack relies on an NTP server to keep all of its nodes and transfers synchronized. In Swift storage, NTP is used to allow multiusers access to shared data by using a scheduler to make sure one device does not lock out the other requesting client nodes. A maximum resolution is 5 seconds; if the synchronization exceeds this limit the results are unpredictable. Careful load balancing is necessary to provide adequate networking between storage and compute nodes. If the number of compute instances wishing access to data conflicts with the scheduling or desired latencies, Swift allows for replication and redundant repositories that self-synchronize. The effect is the data and storage scales with the demand.

In keeping with the Amazon S3 storage model, Swift is designed to be both highly scalable and highly redundant. It can be installed on any commodity hardware that has sharable storage to be used across the entire installation. The visible structure of the storage service is the folder or directory tree rather than the disk itself. The physical disk is invisible to the virtual plane and may contain one single tree or multiple top directory trees that will appear as individual storage sites depending on group and user permissions.

Object Replication A key element in a highly scalable and redundant filesystem is the ability to handle replication with minimum overhead. In the case of multiple VMs that are set up to share similar resources, such as configuration files and operation parameters, Swift must be able to not only have replicated structure, but synchronization so that all nodes sharing common resources can be maintained simultaneously. The Swift object server relies on the `rsync` service to maintain this replication requirement. The attributes set to achieve this are found in the configuration file `rsyncd.conf`. Each active synchronized instance has values indicating individual performance, such as access rules, the maximum connections allowed. The file attributes, such as read-only and locks, are set in the configuration for the instance. The max connections attribute is used to set a value based on the limitations of the physical host where the storage server is located. If this machine has high throughput networking, large amounts of RAM and multiple high speed disks, the max connections value can be set higher. If the storage node is lacking in physical services, such as available cores, networking throughput and cache memory, then setting this value lower helps keep the system from being overwhelmed by the virtualization of the cloud environment.

Client Connections – Proxy Server The use of a proxy server to handle client connections allows the system administrator to scale out the object storage environment without affecting the connection to the front end. User authentication, access logs and connectivity rules are set in the node's `/etc/swift/proxy-server.conf` file. This allows the system to direct connection requests

specifically to a pre-determined source for authentication verification, without the necessity of having to change the basic authentication structure locally. This configuration file also contains the IP address and port to use for both physical or virtual LAN support in this authentication process.

Account Server The account server provides a list of containers available on the node. The accounts are actually in this case the `rsync` account numbers for the connected containers within the storage system. The account server also is used to assign directory folders from the physical disk to a specific proxy server account. Access to specific data could be limited to a single server and account number or shared on multiple servers or accounts.

Container Server The configuration of the container server is very similar to that of the account server, the purpose is to establish server nodes that support the previously generated storage accounts.

Object Server The object server associates the data object with a particular account and container server such that the data is available to an authorized user from a virtualized source on the cloud cluster. This is accomplished by tying the three servers together Account, Container and Object servers in what is called a service ring. The purpose of this structure is for data to be available even if it is non-contiguous and is located across multiple physical disks and multiple hardware platforms. The system will see one or more server storage devices that may be made up of scattered physical drives. The ability to replicate the data on companion storage allows systems to be brought out of service without affecting the perceived location and availability of the data source.

Isolation The storage can be isolated using zones and rings structure. A zone is a group of storage nodes isolated from other nodes in the system. This terminology can be confusing as it is also used in reference to the use of separate physical appliances, network connections, power source, or geographical location. A zone will refer to storage nodes that share common networking and access rules. The Swift Ring structure allows multiple Swift servers and services to locate objects. Swift uses zones to store backup copies or replicas of data on separate systems. Zones are also used to add client access capacity, for example if the storage objects are high demand, and access is requested for multiple users causing access slow down, a zone can be stood up that copies the data and brings up an additional server to share the requested data. In this example we see how the account and synchronization servers come into play. Updates to the data object will be synchronized across all zones as the data source is authenticated. New data appears in each replicated zone within the collective ring such that all sources are up to date.

In a similar fashion, storage nodes can be removed or taken offline from the cluster. This is done by setting a node's priority weight to zero and issuing a ring rebalance command. The node will be removed from service and replications will not include copies and verifications to that node. Once this rebalancing is complete a remove node command can be issued.

System Health and Physical Audits Swift also contains the ability to perform health checks and physical audits on the physical drives attached to a server node. Automated audits help detect physical drives with defects or damaged drives that allow the redundancy systems to replicate the data and prepare for the replacement of the physical disk. Once a physical drive is replaced, the configuration files that describe the Account, Container and Object servers can replicate the missing data and re-attach the drive to the storage ring. This is done by re-enabling the drive back into the ring descriptor, the replicator will reload and check the data on the device and notify the system that the storage object is back in service.

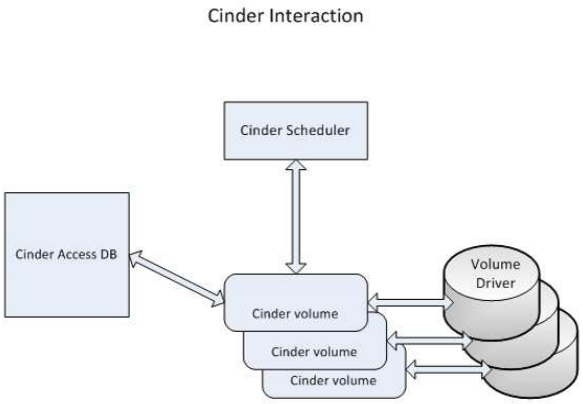


Figure 7.2. Diagram of Cinder component interactions.

This is accomplished by issuing a ring rebalancing command, once the physical drive is replaced. Statistical analysis is accomplished by using the middleware application `swift-recon`. This command provides usage statistics including bytes transferred, read, written and errors. The command line allows for both full reporting on the storage disks and zone based reports. The average loading on the system can also be reported as can the time averaged usage statistics. `swift-recon` also reports on quarantined data objects and replication metrics such as success, failure and any discrepancies between replicated objects.

7.1.7 Cinder – Block Storage

OpenStack allows the user to maintain persistent storage methods. When data is written to virtualized node instances the data is not persistent, meaning when the instance is lost the written data can be lost as well. Block storage volumes are a form of persistent storage that can be attached to existing running compute nodes. The block storage methods used in OpenStack are similar to those in Amazon EC2 elastic storage. The running compute node uses an iSCSI based LVM grouping listed as `cinder-volumes`. This is in keeping with the OpenStack Block Storage service called *Cinder* [19]. The service that supports this is called Cinder and the LVM (Logical Volume Manager) volume group is referred to as `cinder-volumes`. Before any such connections are made the `open-iscsi` service must be mounted. Figure 7.2 shows a high-level view of the interactions between the Cinder block storage components. A logical diagram of a Cinder Storage Node is shown in Figure 7.3.

Preparing the Physical Disk Before launching the `cinder-volumes` service the physical disk or part of it must be properly prepared. This is accomplished by creating a partition configured as an LVM volume.

Authentication Cinder uses the existing authentication methods established during the installation of OpenStack and requires keys and ACL checking just as any of the other services in OpenStack, by defining an endpoint, authentication service and user credentials. This includes container class user ID and namespace identification as well as network ID. In a secure environment a Cinder volume can be assigned access only through a VLAN port on a localized Neutron node within the container.

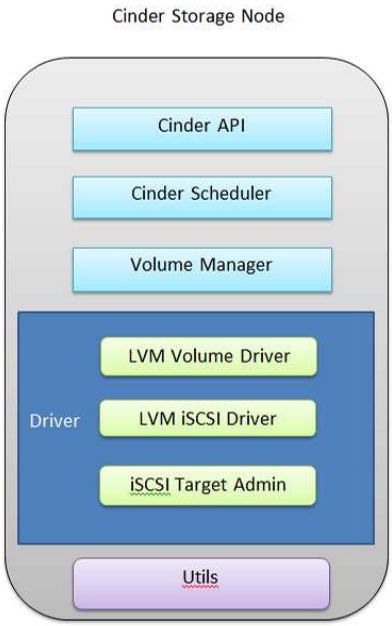


Figure 7.3. Diagram showing components in a Cinder Storage Node.

Replicating Instance Once a Cinder volume service is established and its group credentials created, the instances can be copied like any other instance template and only the group ID and VLAN access credentials changed. This is used to allow multiple users access to a data block without specific use information regarding what data and who is accessing it available to the other users.

Compute Node Attachment Once a Cinder volume is established it can be attached to a compute node with the `nova volume-attach` command from the `nova` client. This will occur only after the authentication service has verified the permissions and network node has authenticated the connection credentials. The perspective from the compute node has been compared to the plugging in of a USB storage device, once it appears on the compute node request layer (after authenticating) it is automatically mounted and the connection takes on characteristics native to that of the attached node. Example, it may be the third attached LVM Cinder volume and therefore appear on this device with a different drive name. During authentication the configuration file allows the administrator to make the drive volume name unique to that user. This helps to obfuscate the use or importance of the drive from other nodes and systems sharing a similar attachment to the same data volume. Once a Cinder volume is attached, only that compute node has access to the block. This follows the USB analogy completely, if another node requests the volume from the system, it must first be relinquished by the current user, then after release it is available for attachment to the new compute node.

7.2 Emerging OpenStack Components

7.2.1 Manila – Filesystem-As-A-Service

A file sharing service called Manila [75] is currently under development. It is based on the Cinder architecture and provides an OpenStack interface for distributed/parallel filesystems, e.g., NFS, GPFS. The Manila project was started in 2013 and was added to the Juno release with an "incubation" status. There are reference implementations for developers as well as initial support for a few vendors/filesystems, e.g., IBM's GPFS, NetApp, etc.

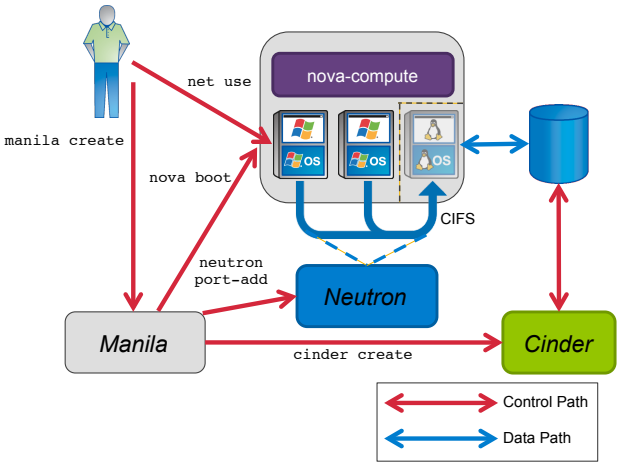


Figure 7.4. Example of the 'generic' Manila file share component for OpenStack (figure source [76]).

The currently supported protocols for file sharing are NFS and CIFS (i.e., Samba). The access restrictions are host based (IP addresses). The project is exploring additional access controls beyond the basic host-based (IP) approach, e.g., LDAP user/group [76]. The Manila file share service interfaces with the Neutron networking layer, which creates a new neutron subnet that exposes (exports) the share. Currently, this share is limited to a single network and is available to the set of hosts on that network [76]. A snapshot of the share can be created, which as of early 2014 was limited to read-only [76].

An overview presentation from the 2014 OpenStack *Juno* Summit in Atlanta is available off of the Manila project's wiki [76]. This includes a demonstration of the code and details from several vendors supporting or developing Manila modules for the OpenStack component. An example of the reference 'generic' Manila component, useful or development/testing, is shown in Figure 7.4. This 'generic' component dynamically instantiates a VM that is used as the file server, which is attached to the respective compute node subnets. This file server exports the share via the NFS or CIFS protocol. At the time of the presentation, the filesystem `mount` from within the tenant VM of the network share (e.g., NFS export) is done manually, e.g., via a remote SSH to the compute node. They mention that they are currently investigating ways to try and automate this mounting process [76].

7.2.2 Magnum – Containers-As-A-Service

The Magnum [100] project was made part of the OpenStack core recently in March 2016 for providing container orchestration within OpenStack. A design principle of Magnum is using the existing OpenStack

projects for their respective services and adding the abstractions necessary to support containerized applications. For parity to VM deployment models in OpenStack, it uses Glance for the image repository, Nova for managing container instances, Heat for service orchestration, and Neutron from networking.

The only supported container format (to our knowledge, future plans may differ) is Docker containers. This would imply Docker's libcontainer as the lower level container execution driver. In contrast to LXD, Magnum is targeting the higher level goal of integrating other OpenStack services for the management and orchestration of container deployments. It provides a "bay" abstraction to place containers in, where one tenant has a bay, and a bay may consist of one or more physical compute nodes (actually nova instances, so virtual compute nodes are supported). There are different "bay types" for representing groups of containers, e.g., Docker Swarm model, Google Kubernetes [100]. The lowest-level container format is important to the SE project because user namespace support by libcontainer (for Docker/Magnum) is not as complete as it is in LXC (for LXD) today.

A current drawback of Magnum's implementation is that containers are provisioned on virtual machines. The support for bare metal provisioning of containers with Magnum is currently under development for inclusion in a future release. Possible inclusion could be Q2 of 2016.

For a complete introduction to the Magnum project, see the OpenStack Summit presentation [101] and visit the project website [74].

7.2.3 LXD – System-Containers-As-A-Service

LXD has a stated goal of managing "system containers" much as a hypervisor manages virtual machines. This is in contrast to the "application container" that is the focus of the Docker ecosystem. LXD is filling a niche role for secure and lightweight system containers that Canonical has identified as a priority, while the broader container community is more interested in the orchestration capabilities and pure OpenStack integration of the Magnum project.

The LXD project was announced in 2014, but was missing several features until recently including its image management functionality and OpenStack integration. This was because LXD was held back by Canonical in anticipation of announcements at the OpenStack Summit, where it was presented as a functional alternative to system-level virtualization with KVM. This effort is very similar to the design that we have proposed in prior reports, but until recently, it didn't support the dynamic provisioning model used by OpenStack for VMs. Now with the nova-compute-lxd plugin, it is possible to start and stop per-tenant containers, as would be done with VM instances. There are more technical details for LXD [101] available in recordings of the recent OpenStack Summit presentations.

Status of user namespaces in Docker: This is not complete as of June 2015. An experimental feature of remapping the root user in a Docker container to an unprivileged user on the host is close to being merged. This is sufficient to support the use case of single-user isolated containers that enforce unprivileged access to a shared filesystem.[1] This feature is now targeted for the Docker 1.8 release, which has a target release date of 4-August.

7.2.4 LXD vs. Magnum

Two main advantages of Magnum over an LXD environment would be a higher degree of integration with Neutron networking options and an API specifically designed for container management, rather than

[1] Docker user namespace progress can be tracked at: https://github.com/docker/docker/pull/12648

the nova API designed for virtual machine management. In a presentation on LXD at the Summit [101], it was argued that since LXD targets the use case of OS containers, the same API's targeted for virtual machines are adequate. Additionally they believed that providing networking access to containers through a Linux bridge on the host was adequate. As part of this project, we have been using the ML2 Arista plugin, which is not yet supported in the LXD model. Magnum claims support for all ML2 drivers. A brief comparison of Magnum and LXD is given in Table 7.1.

Feature	**Magnum**	**LXD**
Neutron networking	All ML2	Linux bridge only
Image storage	Glance	Local per node
Scheduling (via nova)	Supported per bay	No support mentioned
Live migration	n/a	Supported
Exposing block device	n/a	Future feature
Hardware assisted isolation (VTX)	n/a	Future feature
Persistence model	Ephemeral	Persistent by default

Table 7.1. Comparison of Magnum and LXD feature based on latest technical review of available information.

Remarks: Therefore, if a container environment supporting user namespaces were required today (June 2015), the best path would be using LXD as the container management system and the nova-compute-lxd driver for integration with OpenStack. However, the Magnum Container-As-A-Service and Docker functionality is schedule to be released soon.

Chapter 8

Secure Enclaves System Architecture

An important facet of secure-enclaves is the ability to create the perception of single-user environments out of shared resources. This is achieved through a layering of different isolation mechanisms. In this chapter we will present our view of an isolation-centric system architecture. The end user's compute tasks are managed within virtualization based instances (e.g., Chapter 3), which are either hypervisor based virtual machines (VMs) or single-kernel based virtual environments (VEs). The networking between instances can be created through dynamic (reconfigurable) means (e.g., Chapter 4). The architecture employs different underlying storage technologies (e.g., Chapter 5) in concert with bridging technologies (e.g., Chapter 6) to provide the requisite controls for persistent storage.

8.1 Isolation-Centric Architecture

Given the security strengths and frailties of Lustre and GPFS from previous chapters, we advocate a entirely different approach that might not be immediately intuitive. We believe isolation is key to providing security as well as preserving the performance expected in a HPC environment. To this end, we redefine the storage layer up from the filesystem (Lustre or GPFS) to extend into an enclave where the user's view of storage is strongly restricted to authorized areas. These restrictions are achieved in a layered fashion using different isolation mechanisms like OS containers, virtual machines and network segmentation capabilities. The network isolation mechanisms can extend into the storage architecture by implementing a dedicated VLAN per user into the storage filesystem and another for a user's compute resources (nodes). For example, a user requiring 128 nodes for a job will have one VLAN dedicated to the 128 compute nodes and another dedicated to the storage traffic. This holistic approach to a secure-enclaves design is a consequence of the challenge to balance performance and protection. In many cases, as highlighted in earlier chapters, there are gaps in available security capabilities for shared resources in HPC systems. For example, there are limitations in HPC filesystems that inhibit securing the storage layers through native mechanisms due to practical implementation gaps. Thus, the requisite protection and isolation mechanisms are not directly available by the HPC filesystems and must be complemented with additional layers. An illustration of this isolation-centric architecture is shown in Figure 8.1.

The gaps in current HPC filesystems can be overcome via the bridging technologies we have mentioned before, e.g., VirtFS, DIOD, NFS. These technologies can be used to restrict the end-user to approved areas of the parallel HPC filesystem. For example, in Figure 8.1 end-user jobs run in the VEs and the VMs. In either case, the end-user's view of the storage has been restricted by host level mounts from Node1, Node2 and Node3 into their virtual environment. It is important to note that all the hosts

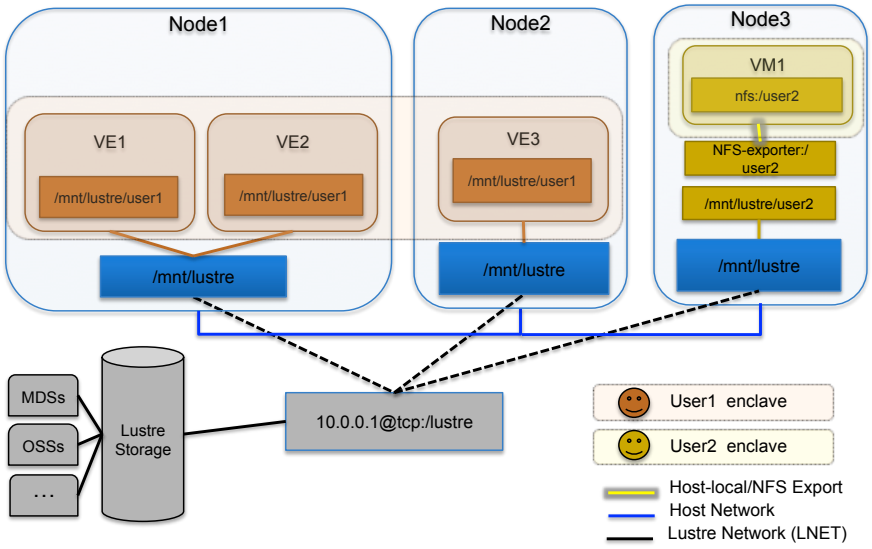


Figure 8.1. Diagram showing different layers of isolation-based architecture using Lustre, three VEs and one VM.

(Node1..3) mount the full filesystem and pass the "user appropriate portion" into the virtual environments. Also note that there is no commingling of end-user processes due to the compute layer virtualization based isolation mechanisms. In those instances where we restrict each user's traffic to their own network VLAN, both into the storage and the VEs or VMs belonging to that user, we have essentially installed a single-user environment (i.e., enclave) – though contention for shared resources remains (e.g., parallel filesystem).

As identified in previous chapters the ability to control the namespace accessible by a tenant is a common gap in many HPC parallel filesystems. This issue of restricting subtree access for a global filesystem can be achieved in an isolation-centric model by employing kernel based namespace restrictions or via I/O forwarding methods in multi-kernel scenarios. These map to the VE and VM use cases within the secure-enclaves architecture, where VEs operate within a single kernel and VMs have distinct kernels. In both cases, the controls are implemented at the host-level, i.e., outside of the VE/VM context, and restrict access to the underlying storage services. The VE based approach may be implemented using "bind mounts" and Linux namespaces to restrict the tenant to a subset of the shared filesystem. The VM case may use NFS or 9pfs to "export" the shared filesystem at a specific depth to restrict access. As noted in earlier chapters, para-virtualized filesystem interfaces may provide more efficient "re-exporting" of host-level filesystems to the guest (VM) context by passing virtual IO devices, e.g., `virtio`.

There are also instances where it can be beneficial to limit portions of the *storage network*, which is the portion of the network dedicated to the storage LAN, e.g., Lustre's LNET. A normal configuration is to have a single storage network with all hosts directly connected. Note, these host machines are running the tenant compute VEs/VMs, which are connected via compute VLANs. The scenario of a single storage network is illustrated in Figure 8.2 where a all hosts in the cluster are directly connected to the single storage network.

An additional degree of network traffic segregation can be achieved by creating "storage VLANs." In this scenario, the hosts are grouped and each group has a distinct interface for the different storage VLANs, i.e., VLANs for the storage-facing region of the network. These interfaces can be used to segregate traffic for the storage network. For example, in a Lustre environment a separate network interface (NI) could be

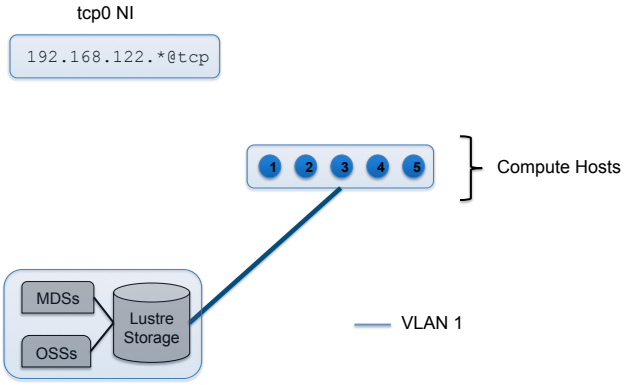


Figure 8.2. Diagram showing all hosts on a single storage network.

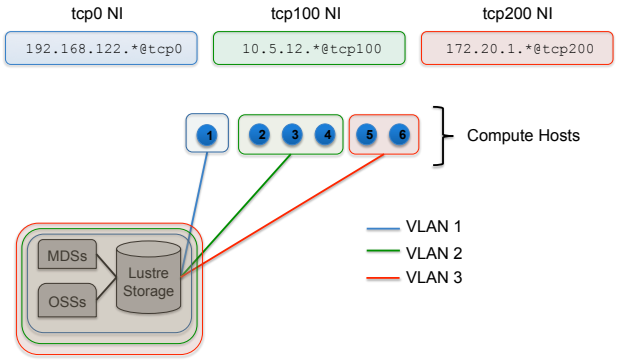


Figure 8.3. Diagram showing hosts on separate VLANs to restrict overall access to storage network.

created on the host for the different types of groups. This multiple storage VLAN example is shown in Figure 8.3. Note, the addition of features like *Dynamic LNET Configuration* (Section 5.1.1) provide a way to create these grouping in a much more agile fashion that can be influenced by the tenant assignments for a given host.

8.2 Instances of the Isolation-Centric Architecture

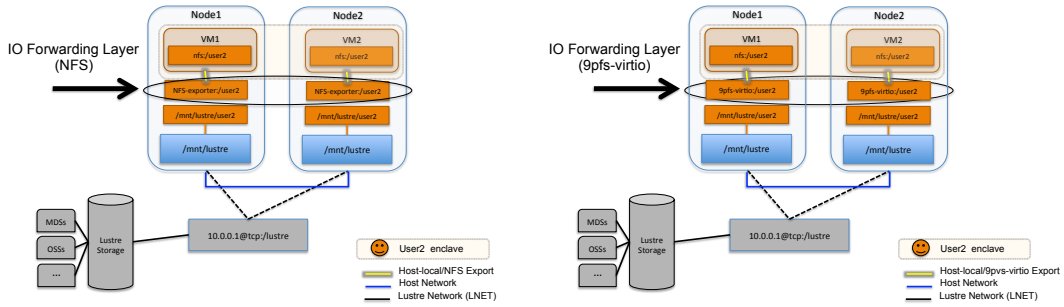
The following examples show concrete instances of configurations that employ the proposed isolation-centric architecture. These scenarios illustrate approaches for managing one of the major challenges when creating secure enclaves: access to shared storage. The configurations use different storage technologies (Chapter 5) and as necessary bridging technologies (Chapter 6) to implement the controls for the isolation-centric secure storage. In Sections 9.7 & 9.8, these examples are evaluated using a Lustre filesystem with controlled access from the VM and VE scenarios.

8.2.1 Parallel filesystem with host-based subtree limitations for VM

Lustre, NFS re-exporter with KVM This configuration seeks to implement subtree export capability using Lustre and NFS (Figure 8.4(a)). A node which is a Lustre client also assumes two other

responsibilities; that of a NFS server and host for a KVM instance. The KVM instance is in turn a NFS client mounting the filesystem served by the NFS server. In this approach, the NFS server only exports that subtree of the Lustre filesystem it wishes to make available to the user. We expect some performance penalty for using NFS.

Lustre, 9pfs re-exporter with KVM This configuration seeks to implement subtree export capability using Lustre and p9fs with KVM (Figure 8.4(b)). A node which is a Lustre client also assumes one other responsibility; that of a KVM instance. The KVM instance in turn uses the p9fs protocol to mount the part of the host filesystem. In this approach, the KVM instance is allocated the part of the filesystem it is allowed to mount via p9fs at the time of its creation. We expect some performance penalty for using P9fs.



(a) NFS re-exporter (b) 9pfs-virtio re-exporter

Figure 8.4. Example instance of Lustre, IO re-exporter with VM. This shows two different approaches for the IO re-exporter, one that uses NFS and another that uses 9pfs with virtio.

8.2.2 Parallel filesystem with host-based subtree limitations for VE

Lustre, bind-mount with LXC namespaces This model seeks to restrict the LXC instance to a portion of the filesystem tree. A Lustre client hosts a LXC instance into which a subtree of the filesystem is bind mounted (Figure 8.5). This maintains the high performance of Lustre in the LXC instance, while protecting other areas of the global filesystem. Another benefit here is that the Lustre client authentication mechanisms are all preserved.

GPFS, bind-mount with LXC namespaces This model seeks to restrict the LXC instance to a portion of the filesystem tree. A GPFS client hosts a LXC instance into which a subtree of the filesystem is bind mounted (Figure 8.6). This maintains the high performance of the GPFS in the LXC instance, while protecting other areas of the global filesystem. Another benefit here is that the GPFS client authentication mechanisms and the filesystem encryption protection are all preserved.

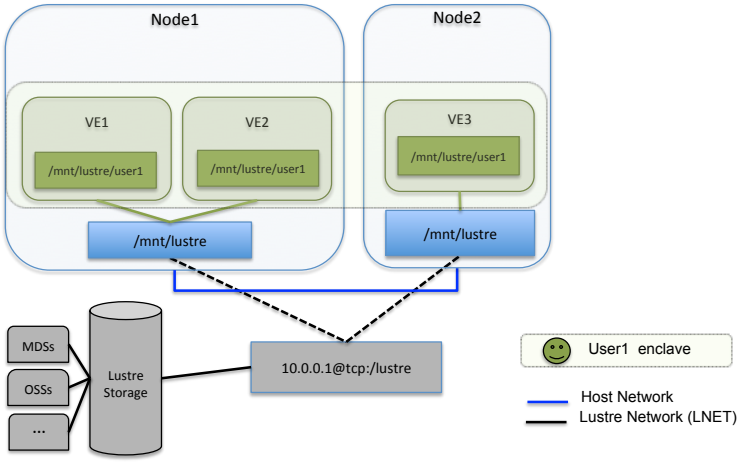


Figure 8.5. Example instance of Lustre, bind-mount with VE.

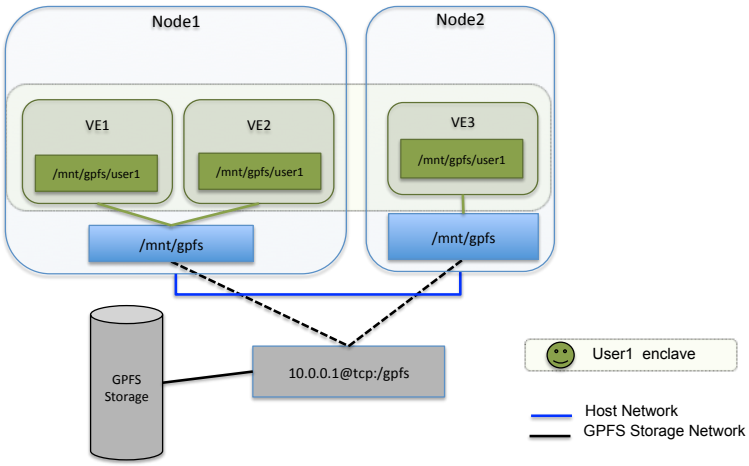


Figure 8.6. Example instance of GPFS, bind-mount with VE.

Chapter 9

Evaluation

A variety of tests have been carried out to evaluate technologies used in a prototype based on the architecture discussed in Chapter 8. The evaluations presented in this chapter provide details about performance and demonstrate the viability of an OpenStack based secure enclaves prototype.

9.1 Secure Enclave Testbed Description

ORNL has constructed a testbed environment to develop and evaluate the use of HPC and cloud computing technologies. This testbed, illustrated in Figures 9.1 & 9.2, has been used to experiment with several technologies, e.g., Software Defined Networking (SDN) for on-demand tenant networks, benchmark controlled access to parallel storage.

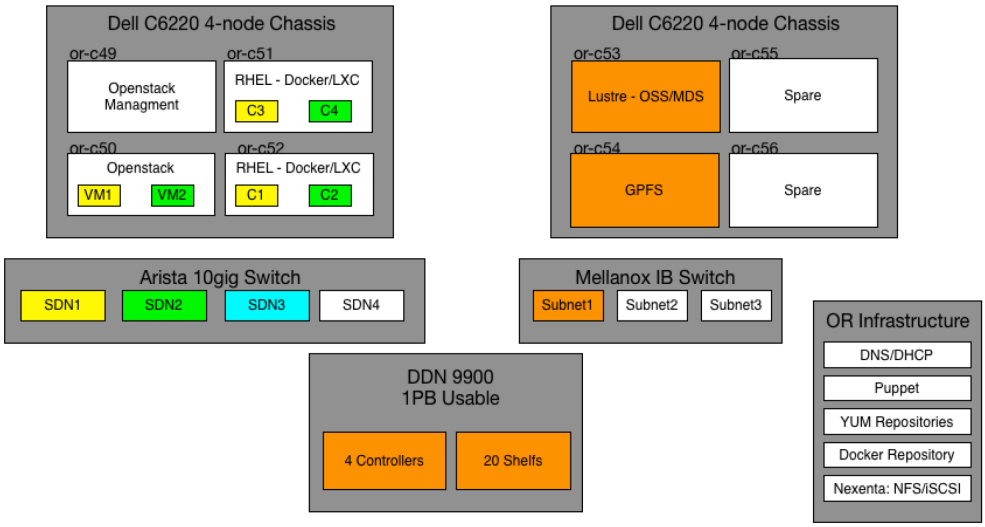


Figure 9.1. Secure Enclaves Testbed Logical Diagram.

The secure enclaves (SE) prototype system is comprised of eight dedicated nodes within the CADES resources at ORNL. The testbed has been configured to aid evaluations (presented in this chapter) that operate with and without the OpenStack environment. As such, four of the nodes were used for manual

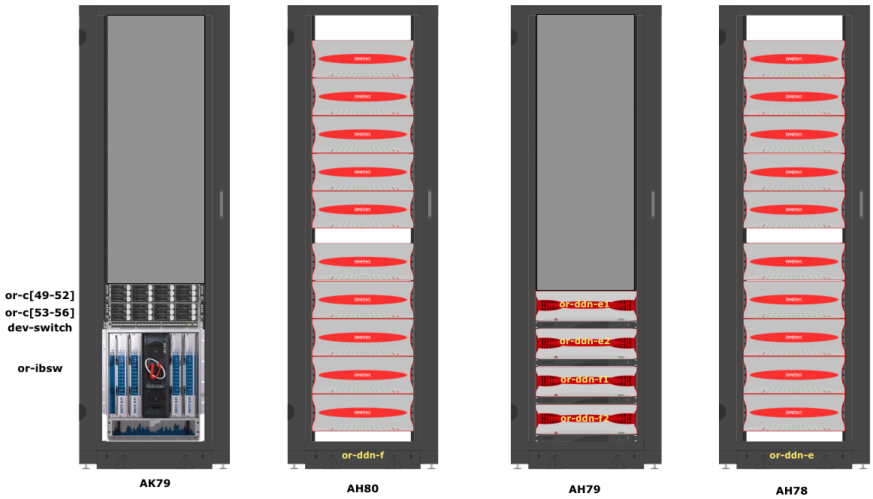


Figure 9.2. Secure Enclaves Testbed Rack Diagram.

configurations/testing and four were used for an OpenStack environment.[1] Briefly, the secure enclave (SE) testbed consists of eight Dell C6220 nodes configured as follows:

- 1 OpenStack management system
- 3 compute systems capable of running bare metal OS images, Virtual Machines, and Linux containers
- 4 compute system are used for general VM/VE tests
- Note: A Lustre storage system (4 OSSs & 1 MDS) is setup on identical hardware and directly connected via 10G to the eight nodes in SE testbed [2]

There are two DDN 10K storage systems accessible from the testbed, each with dual storage controllers and over $1/2$ petabyte usable capacity. Arista 7150S network switches connecting compute and storage resources. InfiniBand is used for connectivity between storage servers and storage controllers.

The SE testbed is based on the "Juno" release of OpenStack and uses Red Hat Enterprise Linux (RHEL) 7 for the host operating system (OS). The OpenStack Neutron component is used to configure networking for the OpenStack compute instances. The Arista ML2 plugin for Neutron provides on-demand enclaving via SDN, which enables creation of dynamic per-tenant network enclaves (i.e., VLANs).

9.1.1 SDN in Testbed

To expose SDN capabilities to OpenStack, Arista provides plugins and drivers for OpenStack integration of Layer 2 and Layer 3 functionality. The Layer 2 plugin enables the OpenStack networking service (Neutron) to communicate with Arista's CloudVision eXtension (CVX) through an Arista mechanism driver over the Arista Command API (eAPI) to provision tenant networks. A typical Layer 2 OpenStack integration is shown in Figure 9.3. CVX is a series of open source extensions to Arista switches

[1]Note: The plan is to add the four manually configured machines into the SE testbed once we have fully integrated all software elements with OpenStack.

[2]Some tests were performed outside of the OpenStack environment using temporary allocations of additional compute nodes, or with customized node builds. In Figure 9.21, the Lustre system is shown for such a test using 10 additional Dell C6220 compute nodes for a single VE+Lustre experiment.

that enable them to use the open-standard XMPP protocol to establish a single view of the network via an industry-standard CLI. eAPI allows applications and scripts to have complete programmatic control over the switch. Once the API is enabled, commands using Arista's CLI syntax are accepted. Responses are machine-readable output and errors serialized in JSON, served over HTTP.

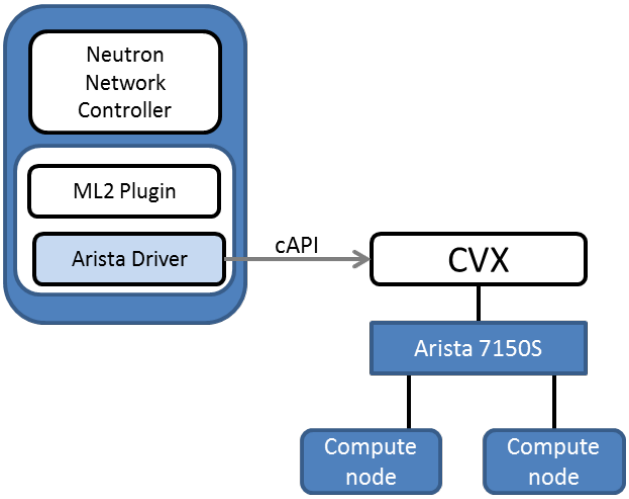


Figure 9.3. OpenStack L2 Deployment.

CVX has visibility of the entire network environment and provisions VLANs on switch interfaces so that the compute instances on the compute nodes have connectivity to the appropriate tenant VLANs. CVX can run in a VM or on an Arista switch itself. The Arista Layer 3 Service Plugin communicates directly with the Arista switches, either TOR or Spine, to provision routing functionality. In response to router create/delete and interface add/remove requests in the OpenStack environment, appropriate SVIs (Switched Virtual Interfaces) are created on respective switches. In future releases the Layer 3 service plugin will communicate through CVX. A typical Layer2/3 OpenStack integrated environment is depicted in Figure 9.4.

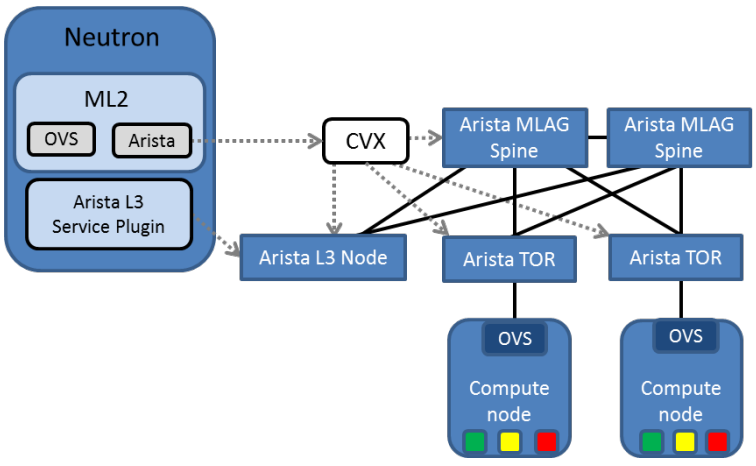


Figure 9.4. ML2 and Layer 3 Service Plugin interactions

9.2 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).

9.2.1 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 filesystems, 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 can 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 contiguous 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.

9.3 HPCCG

9.3.1 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 [77]. 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 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 [14], 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 [48].

9.3.2 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[3]. 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[4]. The parallel tests were run using the same dimensions, which results in a larger overall problem size based on the number processors used, i.e., overall problem size is $nx * ny * (NumProcs * nz)$ [77]. 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[5], using the loops shown in Figure 9.6 for serial and parallel (MPI) tests, respectively. The output from a serial run of the benchmark is shown in Figure 9.5, with the *Total execution time* and *Total MFLOPS* highlighted in green.

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

[4]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.

[5]The selection of $N = 430$ was empirically 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 9.5. 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 9.6. 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 9.7. The startup for the VE based tests with Docker are shown in Figure 9.8. 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.

```
  # 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 9.7. 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 9.8. 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).

9.3.3 Discussion & Observations

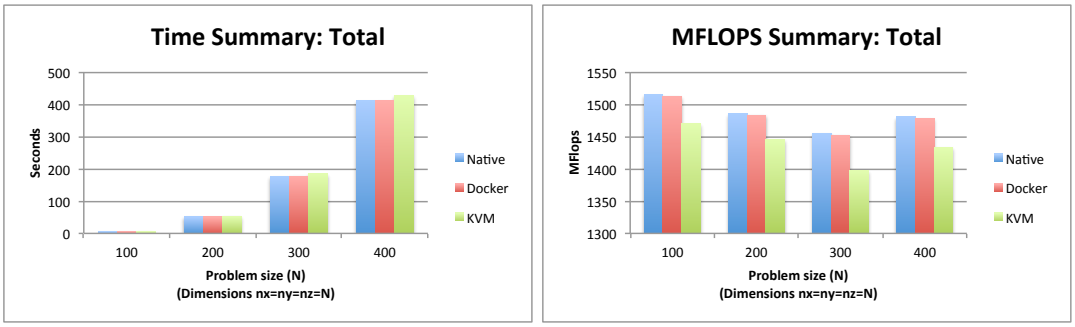
9.3.3.1 HPCCG (serial & parallel)

All results in Figure 9.9 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 9.9, with details in Tables 9.2 & 9.3, reflect this moderate overhead for the VM case and show near-native performance for the VE case.

As shown in Tables 9.4 & 9.5, 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 9.5 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).

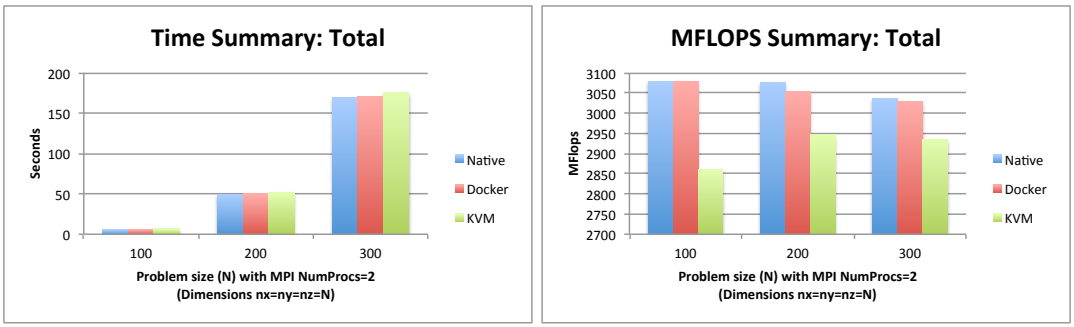
While these baselines were for single node (HPCCG serial mode), they provide a basis for future



(a) HPCCG (serial) Time in seconds

(b) HPCCG (serial) MFLOPS

Figure 9.9. HPCCG (serial) under Native, Docker and KVM at different problem sizes N.



(a) HPCCG (parallel) Time in seconds

(b) HPCCG (parallel) MFLOPS

Figure 9.10. HPCCG (parallel) under Native, Docker and KVM at different problem sizes N.

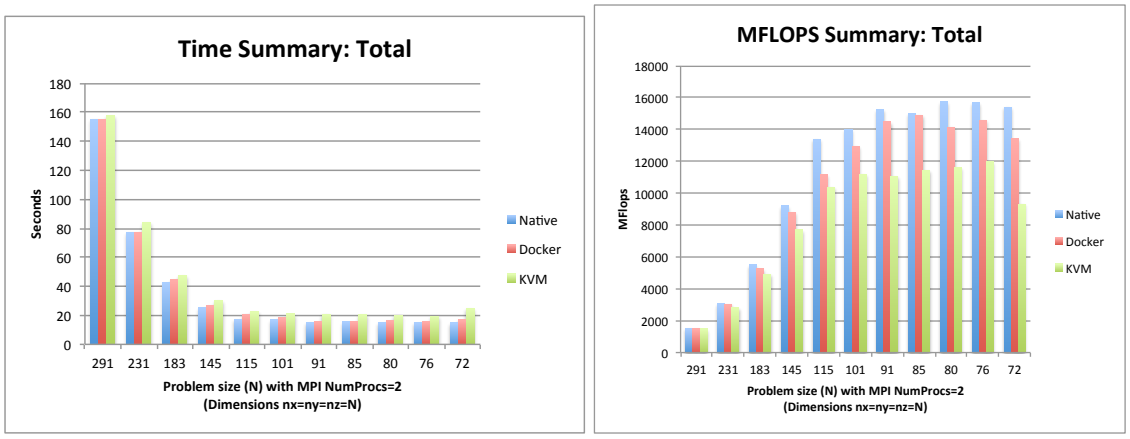
comparisons later in the project. The near native performance of Docker and fast launch times make it a very interesting candidate for further tests with workloads that do not require multiple kernels. Additionally, the tools for Docker launch and execution environment customization show great promise. 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 9.10.

9.3.3.2 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 9.11, 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 9.11. 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 9.10)

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 9.1. The N value was calculated so that $nx * ny * (NumProcs * nz)$ is roughly equivalent 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 9.1. 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 9.2. 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 9.3. 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 9.4. 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 9.5. 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.

9.4 iperf: TCP Bandwidth

9.4.1 Description

We ran basic network performance measurements using the `iperf` benchmarking / tuning utility [54]. The tests provide data about the *Native* (host) performance and the comparison when running under *Docker* and *KVM*.

9.4.2 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 9.12 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 9.3 on page 69.

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.

```
1  # Start server, binding to 10Gig interface
2 [mpiuser@160d5aae2f91 ~]$ ./iperf_2.0.5-2_amd64 -s -B 10.255.1.10
3
4  # Start client TCP test
5 [mpiuser@160d5aae2f91 .ssh]$ ./iperf_2.0.5-2_amd64 -c 10.255.1.10
```

Figure 9.12. Example showing the commands for staring the server and client of iperf test.

9.4.3 Discussion & Observations

The tests were each run 10 times and the averages are shown in Table 9.6. The Native tests achieve most of the 10GigE bandwidth, 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 C.

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

Table 9.6. Network Bandwidth for TCP tests with `iperf` between two nodes on the 10GigE interface.

9.5 On-demand Network Enclaving via SDN & OpenStack's Neutron

9.5.1 Description

An important element in creating customizable secure enclaves is the ability to support on-demand tenant resources, while maintaining isolation between tenants. We completed the setup and demonstrated on-demand tenant network provisioning using OpenStack and Software Defined Networking (SDN).

9.5.2 Setup

There are two Arista 7150S-64 10GigE switches in the testbed. The vendor provides plugins and drivers for OpenStack integration of Layer 2 and Layer 3 functionality. The Layer 2 plugin enables the OpenStack networking service, Neutron, to communicate with Arista's CloudVision eXtension (CVX) through an Arista mechanism driver (plugin) using the Arista EOS command API (eAPI) to provision tenant networks (Figure 9.13). The eAPI allows applications and scripts to have complete programmatic control over the switch. CVX has visibility of the entire network environment and provisions VLANs on switch interfaces so that the compute instances on the compute nodes have connectivity to the appropriate tenant VLANs. In response to router create/delete and interface add/remove requests in the OpenStack environment, appropriate SVIs (Switched Virtual Interfaces) are created on respective switches.

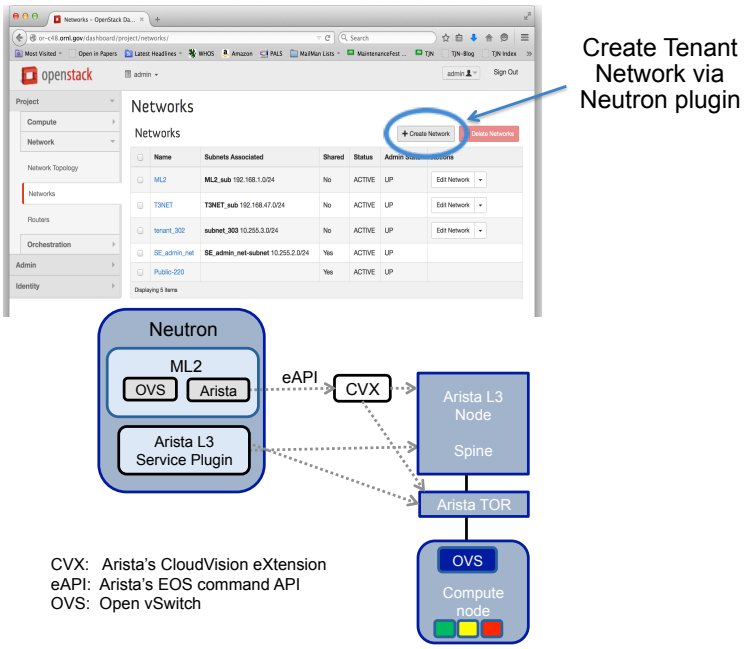


Figure 9.13. Illustration of OpenStack interface creating dynamic tenant networks via Arista's Neutron L2 plugin.

The demonstration was performed using the Arista Level-2 plugin for OpenStack's Neutron component. This configuration allows an authenticated OpenStack user to create and delete their own private networks for use by their compute instances. Figure 9.13, shows a screen capture of the OpenStack interface for creating a new Neutron network, which is the interface to the underlying Neutron ML2 Arista plugin (depicted in block diagram) that communicates with the switch to create the new VLAN. The

network isolation is maintained at the Arista switch via VLAN's that are dynamically created from the OpenStack dashboard interface, i.e., users do not interface with the switch/network controller directly.

Remarks There were some initial complications related to configuration of the software/hardware that slowed the setup of the OpenStack/Arista environment. Most of these were resolved after an upgrade to the embedded operating system on the Arista switch, i.e., upgraded to EOS-4.14.6M.

9.5.3 Discussion & Observations

In the SE testbed environment we use SDN to control VLAN trunking at the switch level. The per-tenant networks (VLANs) are created by either standard users and/or system administrators using OpenStack. To demonstrate the functionality, we performed a basic workflow test and verified the configurations at the switch level using an out-of-band network administrator utility.[6] Figure 9.14, illustrates the creation of a dynamically allocated tenant network ("T4NET"); also shown is the before and after view at the switch level for the defined VLANs. In Figure 9.14, the new tenant network "T4NET" corresponds to switch VLAN 304 named "VLAN0304".

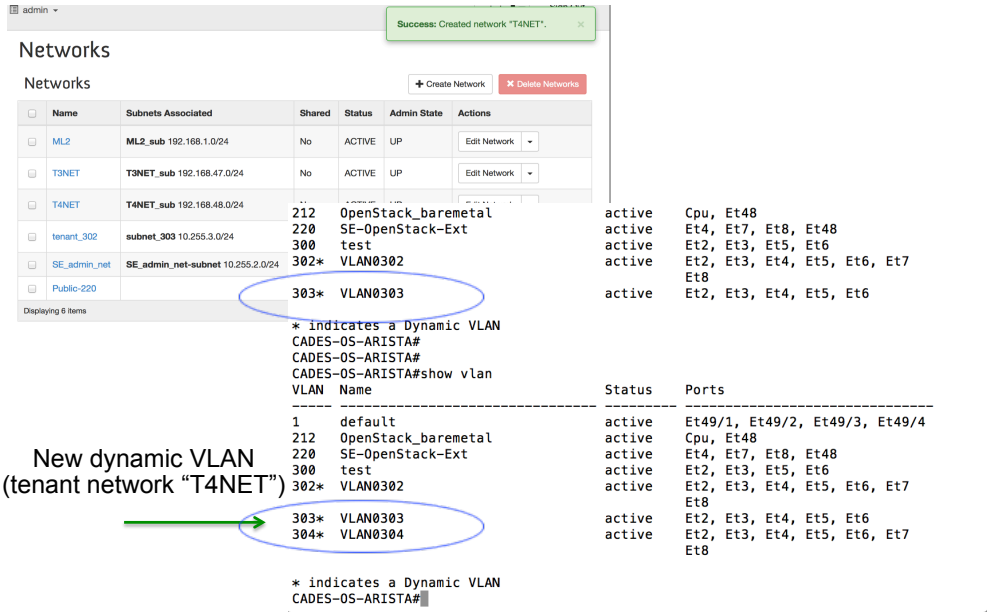


Figure 9.14. View of dynamically created tenant network "T4NET" with OpenStack and display of underlying switch details (before/after add).

Later, when the user removes the network via the Dashboard interface, the switch is updated to delete the corresponding VLAN (e.g., "VLAN0304"). The network definitions can persist beyond VM instantiation, i.e., deleting an OpenStack compute instance does not remove the associated networks. Thus, the tenant (or administrator) can define networks and reuse them across multiple experiments. In Figure 9.15), we show the life-cycle from the low-level switch view:

- the tenant network (T4NET at VLAN0304) exists,

[6] Appendix D contains a series of screen captures demonstrating the steps involved in on-demand network enclaving with SDN & Neutron.

- the tenant's VM instances are terminated, and
- then the tenant network is removed (T4NET at VLAN0304).

Again, these captures were taken from the low-level utility for the Arista switch. A standard user would not need to function at this level and would only see the OpenStack Dashboard or CLI interface to the Neutron definitions.

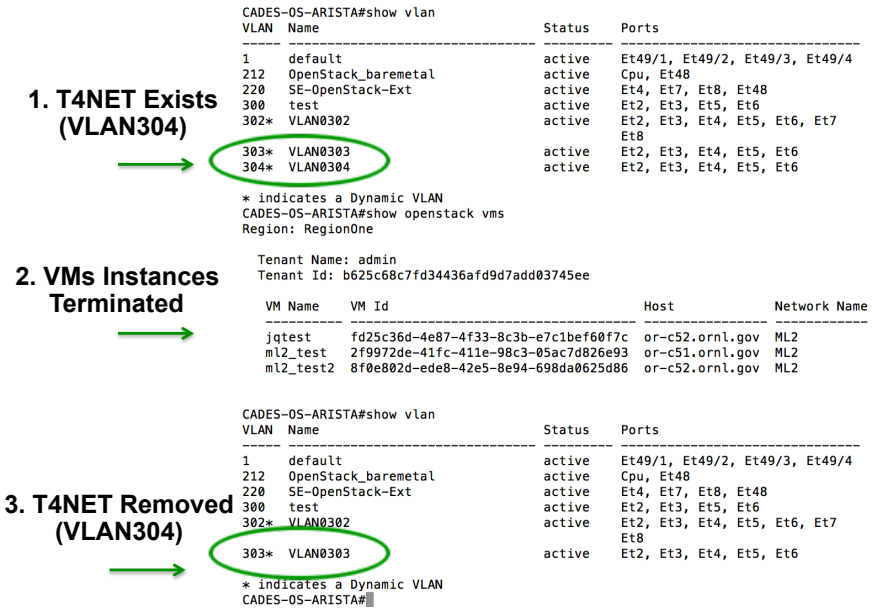


Figure 9.15. View of Arista switch details for life-cycle of dynamic tenant network ("T4NET"), which was created and later deleted using OpenStack Dashboard.

9.6 Network Isolation Testing

9.6.1 Description

The tenants in a secure enclave may have private regions on the network. This network isolation can be used to restrict node access to a particular user. A VLAN will be the primary network isolation mechanism employed in our efforts. Therefore, we reviewed some of the most common attacks against VLANS, including: (i) MAC flood attack, (ii) DHCP Starvation Attack, (iii) Multicast Brute Force Attack, and (iv) VLAN Hopping attack. We then created a set of tests to evaluate these attacks within an OpenStack Neutron testbed to determine the effects of these "attacks".

Note, in conjunction with this effort, we performed a literature review to identify metrics that could be used to quantify isolation. There was very little work found with a focus on the measurement of isolation. Therefore, due to this lack of existing prior work, the most practical approach will be to define relevant areas in which isolation will be necessary, e.g., network isolation, memory isolation, CPU isolation, etc. Then create tests that probe each of these individual isolation areas. The probing may range from brute force to more elegant approach. A suitable metric would then be a pass/fail, where a configuration or implementation is said to "pass" if all the probes measuring the isolation properties are unsuccessful in breaking these properties, and "fail" if the probes do break the isolation properties. Hence, in this section we discuss a set of network isolation tests that were developed for the evaluation of mostly brute force VLAN tests using this boolean pass/fail metric.

9.6.2 Setup

The tenant networks used VLAN tagging mechanism to isolate the network traffic. We performed a review of standard "VLAN attacks" to develop a set of preliminary network isolation tests. The tests are intended to probe the integrity of the isolation mechanisms used in OpenStack's Neutron component. The setup for the tests included two nodes, each with 1 KVM VM, which are assumed to both be apart of 1 enclave. The enclave was arbitrarily assigned VLAN 6. The two VMs "ping" back and forth in order to produce traffic. The attacker is connected to the same switch with 1 VM using VLAN 10 (Figure 9.16).

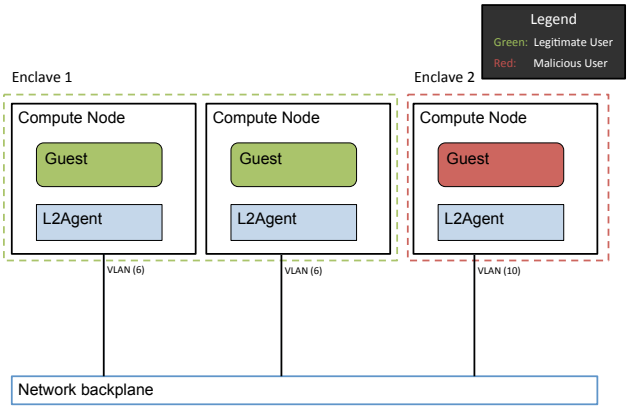


Figure 9.16. Network isolation testbed configuration.

The assumptions here are that the host is not used, only a VM with a bridged network and the VLAN differs than that of the target. The data for the experiment was gathered using `tcpdump`, which was run on

each host and each guest. The purpose is to monitor traffic to determine if a leakage takes place. [7]

9.6.3 Discussion & Observations

A description of the current isolation tests is given below along with a summary of results in Table 9.7. Note, a failed test means the isolation remained intact – so we want the "attack tests" to fail. In general, the isolation mechanisms were maintained (tests failed) with the exception of the DHCP-starvation test, which was able to exhaust available addresses.

1. MAC Flood Test
   - Tests switch's configuration of MAC table per port
   - Goal: Overflow MAC table on port, making switch operate like a hub
2. Multicast Brute Force Attack
   - Tests processing speed of switch
   - Goal: Multicast packets will leak into VLANs of other users
3. VLAN Hopping (Double Tagging)
   - Tests configuration of VLAN filtering
   - Goal: Malicious user able to join other VLANs
4. DHCP Starvation Attack
   - Tests ability to interfere with dynamic host addressing services
   - Goal: Malicious user able to consume all possible IPs for new enclave

Test Name	Status	Remarks
MAC Flood	Failed	Attach from VLAN10 should overflow switch memory, but does not. Did produce DoS attack due to flooding of switch. Can inject packets into other VLANs if using the management VLAN (VLAN 1)
Multicast Brute Force	Failed	No leakage, increased ping overhead ~2ms
VLAN Hopping	Failed	Due to having only one switch for test
DHCP Starvation	Success	Worked in separate scenario. Three VMs on one host with one handing out IPs. DHCP server failed to hand out IPs after roughly a minute.

Table 9.7. Summary of isolation testing results. (Note: Failed test → Isolation remained intact.)

9.6.3.1 Description of Isolation Tests

Here we provide a bit more detail about what each tests is doing with illustrative graphics to clarify the objective of the isolation test.

[7] Note, these isolation tests were run on a separate testbed than the primary SE testbed, which did not include the Arista switch. The evaluation system for the isolation tests did contain a single physical network switch with SDN support.

MAC Flood Test This tests a switch's configuration for the per port MAC table. If the test is successful, the attack will overflow the MAC table on a specific port causing the switch to operate like a hub. The general intention of the MAC Flooding test is outlined in Figure 9.17.

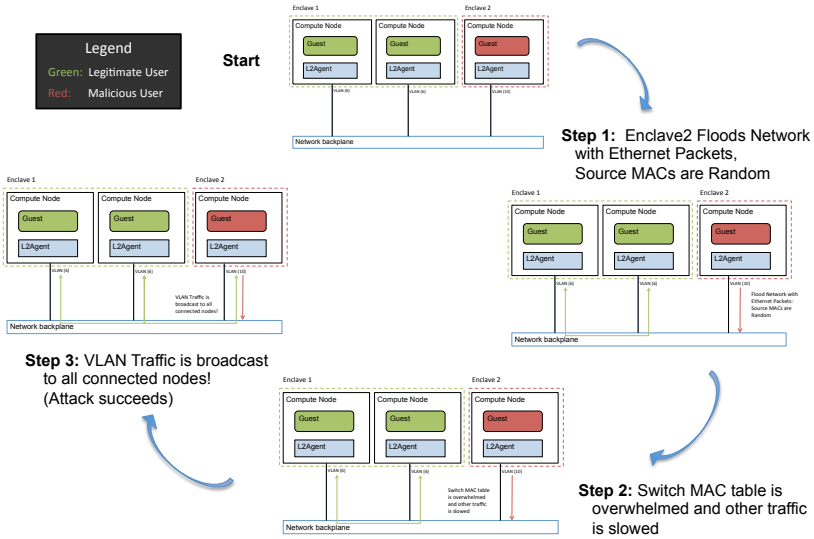


Figure 9.17. Illustration of the steps for "MAC Flooding".

Multicast Brute Force Attack This tests the processing speed of a switch. If the test is successful, the multicast packets will leak into the VLAN(s) of other users. Thus allowing the attacker to inject packets into VLAN(s) outside the the enclave where they were created. This test is outlined in Figure 9.18.

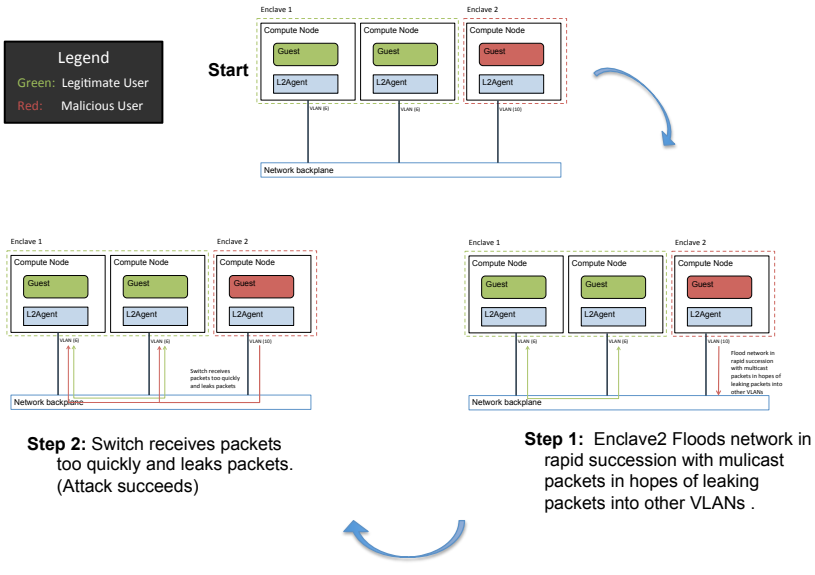


Figure 9.18. Illustration of the steps for the "Multicast Brute Force Attack".

VLAN Hopping (Double Tagging) This tests if an ingress filtering on the connected switch is configured properly. The connected switch is the second switch on the packet's path in the route to destination. If the test is successful, a malicious user could potentially join other VLANs. The test is outlined in Figure 9.19.

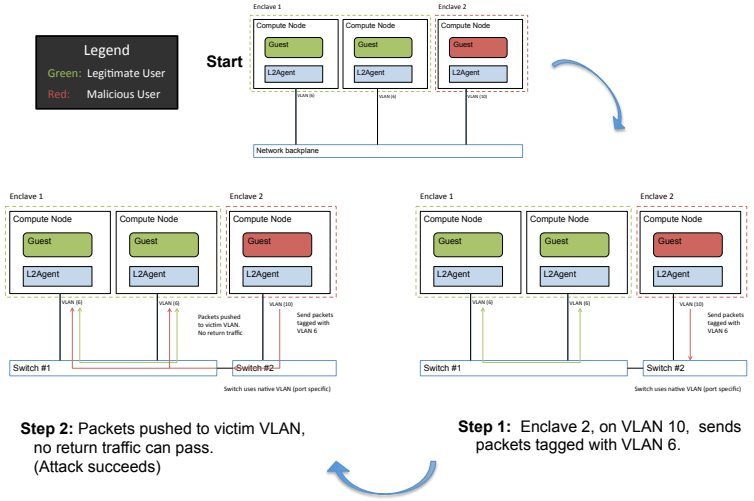


Figure 9.19. Illustration of the steps for the "VLAN Hopping (Double Tagging) Attack".

DHCP Starvation Attack This test is slightly different than the others as it is not testing the VLAN directly, but is generally applicable to network addressing. This attack tests the configuration of the DHCP server and the allocation of addresses on the network. If the test is successful, a malicious user could have all possible IP addresses allocated, leaving none for new enclaves (Figure 9.20). Also, in theory if a rogue DHCP server could be launched, this could provide a basis for interposing on IP services in the network.

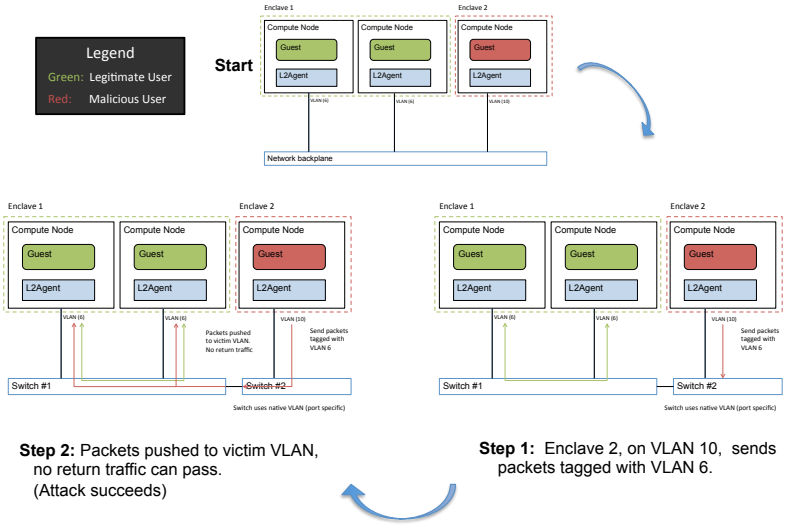


Figure 9.20. Illustration of the steps for the "DHCP Starvation Attack".

9.7 Controlling VM access to Lustre with IO-Forwarding

9.7.1 Description

In the Virtual Machine (VM) case, there are multiple kernel instances, one belonging to the host and another to the guest. The VMs use QEMU/KVM for the OS isolation mechanism and filesystem re-exports or forwarders from the host to the guest VM as filesystem partitioning mechanisms. For Lustre re-exporting we compared the performance of two remote procedure call (RPC) protocols: Network File System (NFS) and Virtualized FileSystem (VirtFS). The VirtFS implementation is based on an RPC protocol from Plan9 that is supported in the standard Linux kernel but disabled by default.

9.7.2 Setup

The KVM host connected to the Lustre storage servers by a 10 GigE interface. These KVM hosts ran on Red Hat Enterprise Linux 7 (Linux 3.12 + kernel options beginning with CONFIG_NET_9P and CONFIG_9P_FS). A Lustre 2.7.0 client allowed the KVM hosts to mount the entire Lustre file system. KVM based VMs ran inside these KVM hosts using the same server kernel version and 9p kernel options as the KVM hosts. Then either NFS or VirtFS were employed as IO forwards to expose subtrees of the host's Lustre file system in the guest VMs. The various IO tests were performed against the file system in these KVM guest systems. When using VirtFS as the IO forwarder, user-level networking (i.e, QEMU's '-net user' option) was employed. The setup is summarized as follows:

1. Host with 9p extensions enabled and Lustre 2.7 client mounts Lustre parallel filesystem
2. 1 VM instance with 9p extensions enabled instantiated
3. Host exports only certain Lustre areas to the VM
4. VM has indirect access to Lustre via exported fs
    - VM access to Lustre marshaled by NFS / VirtFS
5. Run benchmarks from within the VM

Single process FIO with NFS and VirtFS passthrough: The FIO test suite was employed to obtain I/O performance for a single I/O process from a single VM using both NFS and VirtFS as the filesytem passthrough. Table 9.8 show the results. The command used to produce the results displayed in Table 9.8:

```
fio --stonewall --ioengine=sync --iodepth=1 --rw=write --bs=1024k \
 --direct=0 --size=128g --numjobs=1 --fallocate=none --loops=3 \
  --ramp_time=10s --end_fsync=1
```

Single VM, single I/O process with VirtFS and NFS passthrough results		
FIO	**Lustre Read (MB/s)**	**Lustre Write (MB/s)**
Native	1123.4	1154.9
NFS	75.18	69.62
VirtFS	599.2	697.0

Table 9.8. FIO single client I/O performance (IO-Forwarding from host to VM).

Multiple process FIO with VirtFS passthrough: The FIO test suite was employed to examine how I/O performance would scale using the VirtFS passthrough. The command used to produce the results displayed in Table 9.9:

```
fio --stonewall --ioengine=sync --iodepth=1 --rw=write/read --bs=1024k \
 --direct=0 --size=128g --numjobs=N --fallocate=none --loops=3 \
  --ramp_time=10s --end_fsync=1
Where N was 1,2,4,6,8,16 and --rw was either read or write
```

Single VM, multiple I/O processes with VirtFS passthrough results

NumProcs	VirtFS Read (MB/s)	VirtFS Write (MB/s)
1	599.2	697.0
2	1057.3	1177.3
4	1095.2	1174.6
6	1102.2	1177.8
8	1120.7	1176.9
16	1090.4	1176.6

Table 9.9. FIO multiple processes writing I/O performance. Note, all on single host (1 node). IO-Forwarding from host to VM via VirtFS.

Single process IOR with NFS and VirtFS passthroughs: The IOR test suite was employed to obtain I/O performance for a single I/O process from a single VM using both NFS and VirtFS as the filesytem passthrough to the VM. Below is the command used to produce the results displayed in Table 9.10:

```
IOR -a POSIX -b 128g -o 128G_file -F -E -g -v -e -w -r -k -t 4m -i 5 -d 10
```

Single VM, single I/O process with VirtFS and NFS passthrough results

IOR	Lustre Read (MB/s)	Lustre Write (MB/s)
Native	951.0	907.3
NFS	76.40	76.64
VirtFS	747.74	761.5

Table 9.10. IOR single client file-per-process I/O performance (IO-Forwarding from host to VM).

Multiple process IOR with VirtFS passthrough: Parallel performance of the Lustre filesystem was evaluated through the VirtFS passthrough using the IOR test suite. Processes numbers ranged 1 to 16 within the VM. Thus the largest aggregate file size from the single VM was 2 TB. Below is the command used to generate the results of Table 9.11.

```
mpirun -np <NumProcs> IOR -a POSIX -b 128g -o 128G_file \
   -F -E -g -v -e -w -r -k -t 4m -i 5 -d 10
```

Here the MPI library was employed to parallelize the IOR runs even though they were all on one VM.

Single VM, multiple I/O processes with VirtFS passthrough results		
NumProcs	**VirtFS Read (MB/s)**	**VirtFS Write (MB/s)**
1	747.74	761.50
2	1169.36	1234.71
4	1197.24	1232.71
8	1212.29	1235.13
16	1178.39	1233.02

Table 9.11. I/O Performance for multiple IOR processes per VM client.

9.7.3 Discussion & Observations

All I/O tests were performed from within a single KVM based VM residing on a single host. Caching effects were mitigated in our I/O tests by using file sizes per process that were 2 times the amount of RAM and aggregate file sizes ranged from 128 GB to 2 TB. Both Table 9.9 and Table 9.11 display quick, clear convergence to almost constant performance levels that held despite growing aggregate file sizes. This is a clear indication that caching effect were minimized.

NFS as a filesystem passthrough performed at least 10 times slower than native and 8 to 10 times slower than VirtFS in all tests. Single client, single process performance from VirtFS was about 30% less performant than single client, multiple process performance. The latter displayed close to native performance from 2 through 16 processes and showed signs that perhaps its performance was capped by the 10 GigE, 1250 MB/sec, link to storage (See Table 9.11). An immediate unanswered question here is the upper limits to the performance of the VirtFS passthrough method. VirtFS established itself as a viable candidate for filesystem partitioning in future work.

Future investigations will explore how the VirtFS I/O passthrough performs as I/O tasks span multiple VMs. Additional work will also explore bandwidth limits of the VirtFS passthrough.

9.8 Controlling VE access to Lustre with kernel isolation

9.8.1 Description

In the VE case, there is a single kernel instances, e.g., Linux, that accesses a Lustre parallel file-system (PFS). The VE uses kernel isolation (containers) and bind-mounting to control the guest's access to the PFS. Our tests of a VE accessing a PFS used LXC for its user namespace capability. With this configuration, overheads are minimized both in the virtualization mechanism (containers as lightweight VM's) and by avoiding multiple filesystem layers to bridge the PFS to inside the user's environment.

9.8.2 Setup

The filesystem directory structure was set up such that through a combination of bind-mounting for isolation and user namespaces for access rights enforcement, a VE is restricted both in its visibility of the filesystem namespace and its capability to perform filesystem modifications other than the designated unprivileged user.

The tests were run with hosts using Red Hat Enterprise Linux 7 (Linux 3.12 + kernel options CONFIG_USER_NS). The host kernel was using a Lustre 2.7.0 client that was connected via 10 GigE to the Lustre storage servers. The tests used The guest LXC containers ran unprivileged on the host (via user namespaces), but the guests had root access within the container.[8]

The steps for preparing the VE tests were the following:

1. Host mounts Lustre parallel filesystem
2. Host "bind mounts" a section of Lustre filesystem for use by a user from within the VE
3. Two LXC based VE instances (on separate hosts)
4. VE has indirect access to Lustre via mounted fs
    - VE access to Lustre marshaled by Linux VFS
5. Run benchmarks from VE

FIO benchmark: We use the FIO benchmark to compare single tenant performance between the Native (host-level access with no virtualization) to VE performance.
Test command for results shown in Table 9.12:

```
fio --stonewall --ioengine=sync --iodepth=1 --rw=write --bs=1024k \
 --direct=0 --size=128g --numjobs=2 --fallocate=none --loops=3 \
  --ramp_time=10s --end_fsync=1
```

Metric	Native	VE
Write Throughput (MB/s)	1107.4	1109.5
CPU Load (%)	70.25	65.13

Table 9.12. FIO performance comparison between Native and VE (bind-mount from host to VE).

[8]Note, the VE benchmark tests did not require root permissions in the guest context but ran with this option to demonstrate that there was no additional overhead.

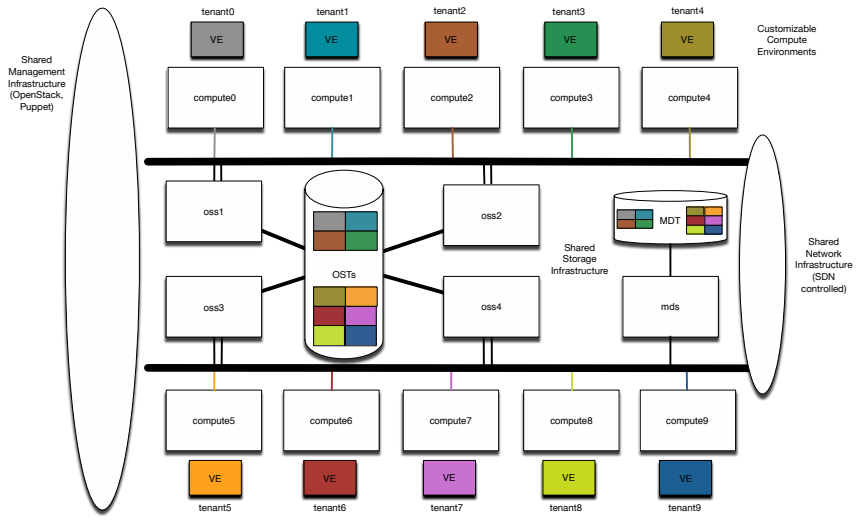


Figure 9.21. Diagram showing the multiple tenant setup using several VEs over multiple hosts all connected to Lustre shared storage.

IOR Setup: Parallel performance of the Lustre filesystem was evaluated using IOR comparing the performance between 1 and 2 nodes (1 process per VE per node).
Test command for Table 9.13:

```
ior -b 128g -t 4m -a POSIX -F -i 5 -E -C -g -v -e -w -r -d 10
```

Processes	Native Write (MB/s)	Native Read (MB/s)	VE Write (MB/s)	VE Read (MB/s)
1	907.3	951.0	939.7	951.8
2 (1 per node)	1629.2	1794.7	1818.2	1865.2

Table 9.13. IOR performance comparison between Native and VE, for 1 and 2 nodes (processes).

Multi-tenant IOR Setup: We also ran the IOR benchmark using a more realistic multi-tenant configuration across multiple hosts. The software setup was the same as previously described but the tests spanned multiple hosts, each running a VE representing a separate tenant (Figure 9.21). The results of this 10 node IOR benchmark test are shown in Figure 9.22.

9.8.3 Discussion & Observations

The LXC based VE achieved near native I/O performance in both filesystem benchmarks. The FIO tests verified that performance of the VE setup is the same as native performance on a single client. Running tests with IOR took advantage of the parallel capabilities of the Lustre filesystem and showed that performance scales equally well between the native and VE cases as the number of VE's (or tenants) are accessing the filesystem concurrently.

Although parallel filesystem benchmarks are typically used to measure I/O performance of a single tenant, running a single parallel job, this setup is demonstrative of concurrent streaming I/O from multiple

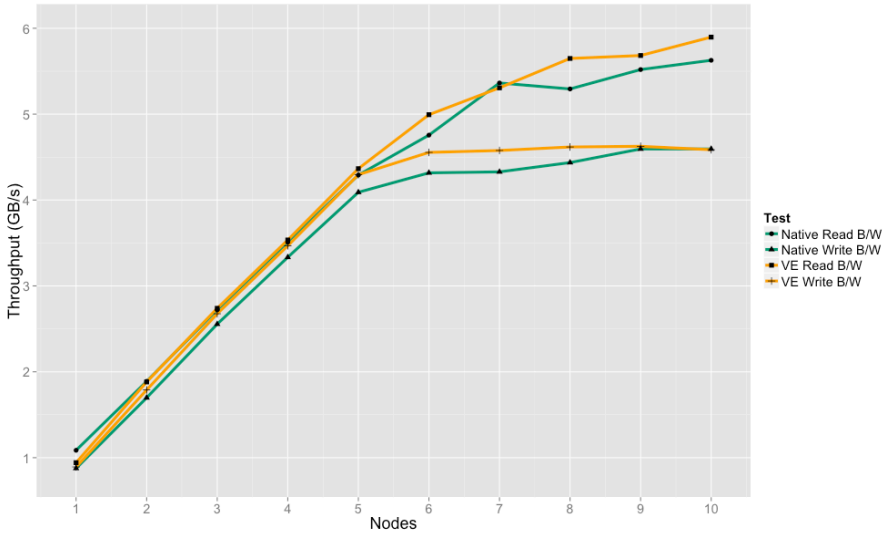


Figure 9.22. IOR performance over 10 nodes comparing Native and VE (LXC). Illustrates VE isolation without I/O performance penalty.

VE's. Our tests were coordinated for the purpose of obtaining an aggregate value, but a scenario where VE's are engaging in uncoordinated I/O would also be able to take advantage of Lustre's parallel performance. These tests are representative of streaming I/O however.

We observed a few instances where the VE performed slightly better than native, which is presumably due to variations in the scheduling policy for processes and containers in the Linux kernel. The Lustre security is maintained by restricting all access to standard VFS operations within the VE, i.e., only the host kernel has a network client to the storage servers.

Chapter 10

Secure Compute Vulnerability Assessment

10.1 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.

10.2 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.

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

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 10.1. 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 10.2. Virtualization solutions and their vulnerabilities' targeted region of the system.

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 10.1 and Table 10.2.

10.2.1 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.

10.2.1.1 The Xen Hypervisor

The Xen hypervisor [8] 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.

10.2.1.2 KVM

The KVM hypervisor [57] 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.

10.2.2 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 10.2.1 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.

10.2.2.1 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.

10.2.2.2 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.

10.2.3 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.

10.3 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 10.2.1, 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 11

Network & Storage Vendor Analysis

11.1 Key Vendors and their role in SDN

All network traffic between physical servers eventually must be connected through real network switching appliances. Each of these hardware devices are built to industry connection standards while providing proprietary hardware, software and capability to make them more desirable on the open market. In addition to the industry developers working on OpenFlow [115], OpenDaylight [91], and OpenStack [94], there are many key network vendors embracing these software platforms and tools as an industry standard. This section is a brief overview of the key players in this field and a discussion of current capabilities and contributions of each.

11.1.1 Arista

Arista Networks, employs the EOS (Extensible Operating System), a Linux based platform, that provides resiliency and programmability across their network products. The purpose of this extension is to provide uniformity in management, the end user does not define and manage individual network appliances as much as the entire enterprise system as a whole. Modern networks require agility and scalable provisioning to handle changes in deployment and recovery from changes or outages. They support OpenFlow, and DirectFlow (Section 2.1.2). Arista supports the OpenFlow v1.3 API providing the ability to control flows through a centralized OpenFlow controller. In addition, Arista has developed DirectFlow, an Arista proprietary technology, which allows controller-less direction of flows using the capabilities within their EOS platform. Support for OpenFlow is provided through an interface on top of their DirectFlow API. Arista's support for OpenFlow on top of DirectFlow can be categorized as a hybrid SDN technology relying upon functionally independent switching infrastructure that can take configurations from a centralized controller. In addition to OpenFlow, Arista supports application interface plugins for OpenStack and was one of the first vendors to support VXLAN. Arista offers centralized management via zero touch provisioning, and uses Extensible Messaging and Presence Protocol (XMPP) [122] to configure groups of network devices. Arista supports multi chassis link aggregation (MLAG) making active use of all links in the network. Network redundancy and making all paths available is important to the modern data center and relevant to reconfigurable networks [6].

11.1.2 Brocade

Brocade's entry in SDN is the Brocade VCS (Virtual Control System) Enhancing the existing Linux based OS with embedded OpenFlow API capability such that the inclusion and addition of specific plugins to handle OpenFlow SDN are not necessary. Brocade favors a higher level of support for virtualization as a key feature of its entry. This is accomplished by combining the OpenFlow command control features with additional support for data plane overlay protocols such as VXLAN. Brocade supports Open Stack. In addition to selling Ethernet, and Fiber Channel equipment, they also have SDN and NFV products. They recently purchased the company Vyatta, and now offer a virtualized firewall product, and OpenDaylight based Vyatta SDN controller [11, 12, 13].

11.1.3 Cisco

Cisco is undoubtedly the largest player in the networking appliance market. They have embraced SDN as the path forward with Cisco ONE (Open Networks Environment). Ironically they have the most to lose from the trend toward SDN as they hold the majority of legacy systems in the marketplace. The SDN environment commoditizes Cisco's enterprise strategy, allowing competitor devices to seamlessly share the network control space. Cisco has SDN support for OpenFlow [21] on their Nexus series, in addition to their Open Network Environment initiative. Cisco also supports VXLAN. Cisco supports multi chassis link aggregation (MLAG) which they call Virtual Port Channel (VPC). Cisco is also championing another form of application driven network configuration called Application Centric Infrastructure (ACI) [20]. ACI appears to be primarily a Cisco backed initiative. For automation Cisco supports standard command-line interface (CLI), API mechanisms, and can operate with configuration management tools such as Puppet [60], or Chef [18].

11.1.4 Dell

Dell has partnered with BigSwitchNetworks to provide its SDN support. This means Dell will use BigSwitch's Switch Light OS. The offering works on newer switches with all features and partial features on legacy Dell switches. Dell's network lineup consists of their Force10 acquisition, and the Power Connect series. Both product lines can work well as bare metal switches using Cumulous Linux [26], or can interoperate with Vendor specific APIs through a central controller such as the NEC ProgrammableFlow Networking Suite [84]. Both Power Connect and legacy Force10 lines now support Cumulus Networks [26]. They also support a Dell backed managed SDN solution [28].

The SDN offering from HP is the most generic on the market. HP's switch fabric ASICs are the most flexible in their intrinsic control capabilities and therefore can handle virtually any possible combination of routing paths possible on their ports. Utilizing a custom version of Linux, HP has most of the common open source SDM solutions available, and even allows the user to use their own flavor of Linux, such as RedHat, SuSE, or Ubuntu on the switches if they so desire [49].

11.1.5 Juniper

Juniper Networks has incorporated the Junos OS as their solution to SDN and virtualization support. Their approach is to provide a platform that allows the OS to work with all of their products and to add functionality through extensions and applications running on the OS. This means that the OS has been designed to provide support for these functions rather than the functionality itself. Junos has a modular approach that allows their range of network appliances to be configured to meet specific needs. Different

models therefore may have specific versions of modules that share similar features with other models but are written for that device. The underlying OS is common for all [56].

11.1.6 Mellanox

Mellanox is known for their high speed, low latency InfiniBand appliance. They expanded into Ethernet networking and have one rack unit on top of rack switches. These are non-blocking and have full Layer 2, and Layer 3 functions, as well as SDN support [81]. They have introduced aggregating switches into a fabric using a technology called Virtual Modular switching. This is similar to MLAG and VPC, except, if you lose one of these switches, you don't lose half of your bandwidth [80].

11.1.7 Network Vendor Conclusion

All of the major vendors are supporting SDN to various degrees. Some are embracing their own solutions, while others are embracing the open source community and standards, often including industry standards combined with proprietary capability. Cisco, and Arista are looking at making bare metal switches and using Cumulus Linux as their operating systems. The advantage to OS and SDN standards is the ability for network appliance vendors to focus on the ASIC and hardware development. The adoption of open-source software technologies by switch vendors poses challenges for differentiation in a competitive marketplace. It would be reasonable to expect that switch vendors will continue to adopt open source software and open standards while continuing to differentiate by offering more advanced features through proprietary software and interfaces. For the secure enclaves project, we will focus on the use of broadly supported SDN capabilities to alleviate reliance on vendor proprietary technologies [36].

A summary of the vendor compliance with the OpenFlow standard is given by Nunes et al. [89], which is repeated here for easy access in Table 11.1.

Maker	Switch Model	Version
Hewlett-Packard	8200zl, 6600, 6200zl, 5400zl, and 3500/3500yl	v1.0
Brocade	NetIron CES 2000 Series	v1.0
IBM	RackSwitch G8264	v1.0
NEC	PF5240 PF5820	v1.0
Pronto	3290 and 3780	v1.0
Juniper	Junos MX-Series	v1.0
Pica8	P-3290, P-3295, P-3780 and P-3920	v1.2

Table 11.1. Main current available commodity switches by makers (vendors), compliant with the OpenFlow standard (from Table-II of [89]).

11.2 Storage Vendor Overview

There are several vendors that support network storage or more specifically NFS (Network File Systems). Most of these are a form of classic large enterprise centralized archive storage such as SAN (Storage area network) servers or just simply additional servers that have large RAID (Redundant Array of Inexpensive Disks) arrays. In an HPC deployment model they would be externally connected peripheral services, not aligned with the compute nodes directly. Some of these hardware based storage configurations

can be adapted using the OpenStack services to perform a more abstracted storage service within the cloud, but few exist that have been configured to handle the rigors of the HPC compute environment. The following is a sparse list due largely because few vendors have started servicing the HPC market directly, and fewer still provide the necessary bandwidth and configuration for high demand access adopting rather the slower and more accessible archive transfer model. The following is a synopsis of the vendors that have either directly adopted the design path necessary to support HPC or systems that are part of emergent technology that can easily meet the rigors of HPC.

11.2.1 Seagate/Xyratex

Seagate corporation is one of the longest standing disk drive manufacturers in the world, and has supported SAN and RAID server technology for years. Recently they acquired *Xyratex* [123] and started to aggressively pursue the storage appliance market. This division of Seagate offers several high speed redundant storage offerings but further examination reveals that they all have a very common platform topology and are equally scalable. This discussion will cover one of their smaller build outs, the *ClusterStor*™ 1500 as an example, the other models in this family are mainly scaled-up size and capabilities of either racks of this model or larger versions of the same. The ClusterStor devices are built upon the successful Seagate OneStor enterprise/data center platform.

11.2.1.1 Security

The ClusterStor series is ICD 503 (Intelligence Community Directive) (DCID 6/3 PL4) Compliant, including MAC (Mandatory Access Control) and uses explicit auditing of both usage tracking, access logging. The policy “least privilege” is enforced using audit logging and encryption. The system supports multiple separate file zone classifications on a single server or filesystem.

11.2.1.2 ClusterStor Platform

This appliance is marketed directly for high performance computing and uses the open source Lustre filesystem [66] taking advantage of its parallel file structure. The intrinsic customized OS a modified SELinux (Security Enhanced Linux) loaded on the ClusterStor 1500 organizes the storage blocks base units with network interface and Expansion ports referred to as SSU (Scalable Storage Units), System administrators can alternatively attach additional physical storage devices called ESU (expandable Storage Units), these physical appliances contain access controllers and additional disks but lack the network interfaces for external connection. In most common configuration each data file is striped across the entire assigned filesystem organization on SSU and ESU so that access is handled in parallel. The physical disk drives are abstracted in a sub layer directly handling I/O read and write requests in a synchronized simultaneous manner. Custom SATA (serial AT attachment the AT from the original IBM PC AT model circa 1980’s) [107], ASIC devices provide the synchronized access scalable between the nominal 1GB/s up to 100 GB/s. A single ClusterStor SSU unit can have up to three additional ESU units attached, for full physical expansion build out. The SSU contains the management unit that handles all of Lustre’s metadata services.

11.2.1.3 Physical Drive Capability

Each of the drive bays can handle any of the available 2.5” SATA disk drives commonly available on the market, however all of the published specifications assume that the dual port 6Gb/s SAS drives are used

for maximum performance. The number of drives installed is limited by the size and model of the SSU and ESU attached, the embedded Lustre filesystem abstracts the system to a single storage space that can be linearly scaled out with additional units and across other Lustre servers.

11.2.1.4 Data Throughput

A single physical system can provide between 1 GB/second up to 100 GB/s however a parallel cluster of systems can theoretically support throughput rates of a sustained 1 TB/s [22].

11.2.1.5 Specification Table ClusterStor™ 1500

The following is a table of capabilities and specification for the range of the model 1500, any of the other larger models will be similar with a higher capacity, the basic throughput will be identical to a configuration with multiple smaller devices. The difference in models appears to be mainly to consolidate physical space requirements by sharing a larger enclosure. Table 11.2 was copied from the ClusterStor 1500 product bulletin courtesy Seagate Inc [22].

Parameter	Description
Filesystem Performance	1.25GB/s up to 110GB/s sustained read and write
Raw OST Capacity	80TB (4TB SAS HDDs) to 10.PB (6TB SAS HDDs)
Usable File System	60.4TB (4TB SAS HDDs) to 7.9.PB (6TB SAS HDDs)
Lustre Network Protocol	Infiniband QDR / FDR or Ethernet 10Gb/s 40Gb/s
File System Lustre	2.1 (with Seagate add ons)
Maximum Files	280 Million
SSU Storage units	(2) 2.5” HDDs 300GB RAID 1 1+1 (21) 3.5” RAID 6, 8+2
ESU Storage Units	(21) 3.5” RAID 6, 8+2
Hot Swappable	HDDs, Redundant Power/Cooling supplies
Operating System	SELinux
Management Network	Dual 1 Gigabit Ethernet

Table 11.2. Seagate ClusterStor product specifications (table source: [22]).

11.2.2 Oracle ZFS Storage Appliance

The ZFS is originally designed to support VM and cloud storage requirements, it follows an architecture paradigm of mapping a HDD (hard disk drive) with large arrays of DRAM storage. This format is referred to as Hybrid Data Pool (HDP) [126].

11.2.2.1 Resiliency

The HDP topology uses the HDD physical drives for resiliency of the data. Access requests use the high speed scalable DRAM arrays and periodic backups are made to the slower HDD and checked against read and write/store requests. The user access is handled directly from the DRAM and mimics the smaller CACHE memory access used on most high efficiency compute nodes. Simultaneous access to the DRAM array provides the system with parallel failover and self healing filesystem capability.

11.2.2.2 Vitualization Support

Although designed largely for enterprise applications, the Oracle ZFS system does include plug-ins to support virtualization. Including OpenStack and other cloud implementations.

11.2.2.3 Filesystem Support

The Oracle ZFS specifically includes support for Oracle data base file structures, although there is no listed support for Lustre or GPFS, there is no indication that these appliances cannot be reconfigured to use either. This will require additional research and inquiries to ascertain. The system also supports the OpenStack Cinder (block storage server) [126].

Parameter	Description
Filesystem Performance	1.25GB/s up to 110GB/s sustained read and write
Usable File System	60.4TB (4TB SAS HDDs) to 7.9.PB (6TB SAS HDDs)
Lustre Network Protocol	Infiniband QDR or Ethernet 1Gb/s 10Gb/s
File System	Oracle Solaris ZFS
Hot Swappable	HDDs, Redundant Power/Cooling/ supplies
Operating System	Semi custom Linux
Management Network	1-10 1 Gigabit Ethernet ports

Table 11.3. Oracle ZFS Storage appliance specifications (table source: [126]).

11.2.3 Additional Systems

The other storage systems of note are not geared specifically toward HPC, but rather fit into standard Enterprise/Cloud service models. These include SAN (Storage Area Network) servers from HP and Dell, Fujitsu and others. In most cases these are currently used in OpenStack storage server applications, additional research is need to ascertain the performance specifications when used in an HPC environment with Lustre and GPFS. Certainly all of the models collected for this study can be upgraded with Mellanox Infiniband and 10 + GB Ethernet controllers, but the backplanes and server structure does not specify the advantage nor ability to increase beyond native network interface speeds. Most of these are iSCSI (Internet Small Computer System Interface) which is a relatively slow interface compared to the two identified vendor products.

Chapter 12

Conclusion

Supporting multi-tenant environments within HPC systems holds the promise of supporting a diverse set of workloads at significantly higher levels of performance and scalability than a traditional utility compute cloud environment. Traditional cloud computing environments address the security challenge of multi-tenancy through judicious use of full machine virtualization, network virtualization, and per-tenant storage. This approach sacrifices performance, scalability, and usability in favor of secure multi-tenancy. Our work is focused on providing multi-tenant environments, "secure enclaves", at very low overhead through the use of carefully selected isolation mechanisms at the compute, network and storage layers.

12.1 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 and dynamically configurable networks.

A review of virtualization classifications is provided in Section 2.2, which detailed the different types of OS-level and system-level virtualization. The main distinction between different virtualization technologies has to do with the degree of integration with a host kernel. Throughout this report we have used the terms *virtual environment (VE)* and *virtual machine (VM)* to distinguish between container-based (single kernel) and hypervisor-based (multiple kernel) virtualization. A brief background of SDN and NFV is discussed in Section 2.3. A review of key terminology and relevant SDN concepts is is discussed in Section 2.1.2. This includes a review of state-of-the-art technologies and emerging standards such as OpenFlow and OpenDayLight. We also provide background information on two HPC filesystems, Lustre and GPFS, in Section 2.4. A short review of relevant security classifications is discussed in Section 2.5. This was followed by information on supporting security technologies (Section 2.6) that are particularly relevant for secure storage, e.g., FIPS, Kerberos, GSSAPI.

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.

Chapter 4 provides an overview of methods for implementing reconfigurable networks with specific focus on utilizing reconfiguration as a mechanism to support secure enclaves. This overview of methods includes traditional SDN, Hybrid SDN, and the use of OpenStack Neutron. A more detailed treatment of Neutron based approaches is provided in Section 4.8.

In Chapter 5, we focused on the security mechanisms provided by Lustre and GPFS, highlighting protections and identifying vulnerabilities. Based on this, in Chapter 6 we described several "bridging technologies", which can be used to supplement protections at different layers in the software stack. This is particularly relevant for addressing specific weaknesses in storage isolation, when trying to also maintain high performance IO operations. For example, in Section 8.2 we outline the application of "bridging technologies" to protect the gaps in Lustre and GPFS to for a more secure HPC storage solutions.

Based on this information, in Chapter 8 we presented our vision for a Secure Enclaves System Architecture. The architecture is centered around the careful use of isolation mechanisms at different layers in the software stack to offer protection, while maintaining performance. The prototype for this approach is described in Section 9.1, and is based on OpenStack (Chapter 7). The technologies used in the prototype are evaluated through a series of experiments that demonstrate the performance of VM and VE use cases in conjunction with different compute, network and storage tests (Chapter 9).

The evaluations that have been carried out thus far in the project are described in Section 9. The first test (Section 9.2) focuses on experiments using the Linux `user` namespace to differentiate guest/host user contexts and provide different access rights accordingly. The next evaluation (Section 9.3) 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. Section 9.4 provides basic network bandwidth/latency numbers for VM, VE, and native tests using the 10G network interface. In Section 9.5 we detail the demonstration of on-demand network enclaving via SDN using OpenStack's Neutron component.[1] This was followed by a series of network isolation tests (Section 9.6) that probed the VLAN mechanisms to determine if simple brute force attacks could succeed against OpenStack's Neutron.[2] The final two experiments are focused on controlling access to storage, specifically controlling VM and VE access to a parallel filesystem (Lustre). In Section 9.7, two different IO-Forwarding methods were compared to evaluate the performance of re-exporting a host mounted Lustre filesystem to a guest VM. The VirtFs/9pfs showed very good performance for IOR and FIO benchmark tests. The other tests (Section 9.8) looked at the VE instance and showed that a combination of bind mounts and user namespaces could provide secure near native performance for accessing Lustre from the VE.

In Chapter 10, 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 CVE data indicated that many of the errors tend to occur in the same types of code regions, e.g., device & instruction emulation, across the different virtualization implementations.

SDN and NFV vendor technologies are reviewed in Section 11.1, including Arista, Brocade, Cisco, Dell, Juniper, and Mellanox. Each of these vendors are supporting SDN and NFV to various degrees either

[1] Appendix D contains a series of screen captures demonstrating the steps involved in on-demand network enclaving with SDN & Neutron.

[2] The isolation tests were performed out in a separate testbed than the on-demand tests, simply because of setup times and the scheduling of tests in the project.

by adoption of open source community technologies or through a combination of proprietary technologies and open APIs. A summary of vendor compliance to the OpenFlow standard is presented in Section 11.1.7. In Section 12.2.5 an overview of notable issues and limitations for SDN/NFV is discussed. Although SDN and NVF are becoming the operational standards of large compute resource deployments, the implementation of these standards is still being refined [89]. Additionally, several HPC storage appliances were examined in Chapter 11.2 to decide if particular vendors provide more complete security protections than other products.

12.2 Observations

We will now briefly discuss observations from this end of year report. The intent is to briefly summarize, or highlight, important points that are useful for creating secure high performance computing environments.

12.2.1 Benchmarks

Compute 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.

I/O The evaluations performed with single kernel (VE) and multiple kernel (VM) environments showed very promising results. We were able to maintain parallel filesystem isolation for multi-kernel instances using an IO-Forwarding approach. The FIO benchmarks showed near native write performance at one process per node for a VM accessing a Lustre share that was exported from the host via VirtFS/9pfs. The IOR benchmark results also showed very good results, compared to purely native access to Lustre, with roughly 15-20% less than native performance for the VirtFS/9pfs re-export to VM. We found that the performance of the VirtFS approach is greatly dependent on the `msize` (payload size) parameter for the 9pfs protocol. Our initial tests using the default VirtFS `msize` showed roughly equivalent performance between VirtFS and NFS. However, changing this to an `msize=1M` resulted in good performance that improves as the the number of processors (writers/readers) increases. As for usability, the VirtFS requires a kernel recompilation to include the 9pfs and 9pnet capabilities, but the code is part of the standard Linux source.

The VE based tests for controlled access to a Lustre filesystem showed excellent performance. The LXC based tests achieved near native I/O performance in both filesystem benchmarks (IOR and FIO). The CPU load in the FIO tests were roughly equal as well, with the VE case actually having a slightly lower

percentage of CPU load than native. The IOR tests showed that both read and write performance was effectively the same for native and the VE instances. Note, there was a slightly better than native performance in the VE tests, which was assumed to be related to differences in the scheduling of containers within the Linux kernel. This trend was maintained in a slightly larger demonstration that employed 10 nodes using VE's running the IOR benchmark – the VE read/write was equal or above native performance. Regarding security, the Lustre storage network was separated from the guest VE context by marshalling all accesses through standard VFS operations within the VE. Only the host had a network interface attached to the Lustre storage network (LNET) and all VE access passed through the bind-mounted Lustre share. In the future, UID/GID mapping functionality that is emerging in Lustre v2.7 will offer complementary security capabilities.

12.2.2 User namespaces & Container Isolation

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 9.2 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 9.2 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.

OS Containers There are two very interesting projects emerging that are focused on container based isolation, namely LXD and Magnum. These were described and contrasted in Section 7.2. The LXD (Linux Container Daemon) project from Canonical provides an API for managing LXC containers with `user` namespaces enabled by default [72]. However, like Docker, it currently takes a simplified approach to the mapping of user IDs from host-to-VE. Currently, LXD 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).

The Magnum Containers-As-A-Service project is focused on providing container orchestration within OpenStack. Magnum provides a new OpenStack abstraction called a "bay", which is where containers are placed. The Magnum "bay" provides a tenant the ability to place containers on one or more physical

compute nodes (actually Nova instances, so virtual compute nodes are also supported). This provides a higher level of abstraction and leverages much of the functionality available in OpenStack. The Magnum effort, currently, only supports the Docker container format via libcontainer. The status of user namespaces in Docker, as of June 2015, is incomplete. Docker currently has sufficient support for single-user isolated containers that enforce unprivileged access to a shared filesystem. This feature is now targeted for the Docker 1.8 release, which has a target release date of 4-August-2015.

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.

Based on the current features in LXD and Magnum, as summarized in Table 7.1, if a container environment supporting user namespaces were required today (June 2015), the best path would be using LXD as the container management system and the nova-compute-lxd driver for integration with OpenStack. However, the Magnum Container-As-A-Service and Docker functionality is schedule to be released soon and we plan to continue our evaluation in future work.

12.2.3 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.

12.2.4 Security Classifications

Regarding the security classifications discussed in Section 2.5, 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.

Protection Levels: We also reviewed the details of DCID 6/3 Protection Levels [27] to clarify where we stand with respect to compliance in the OpenStack SE testbed. The concept of Protection Levels applies

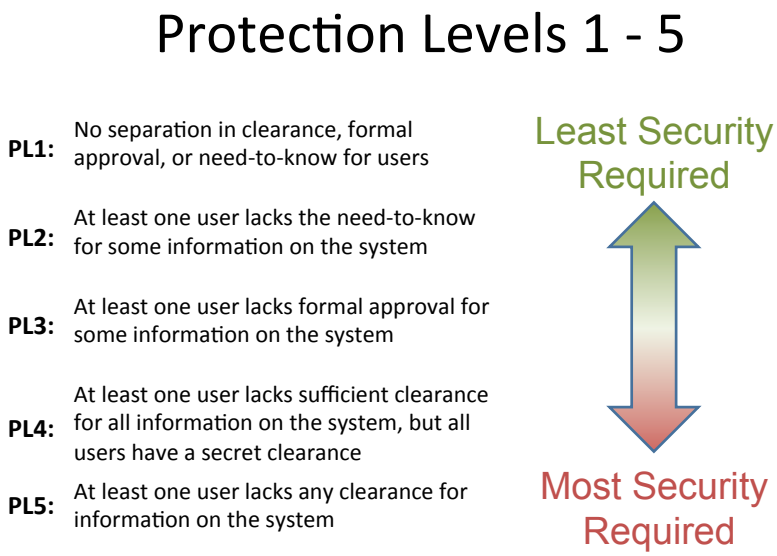


Figure 12.1. Summary of DCID 6/3 Protection Levels (PL) for inter-tenant security requirements.

only to "confidentiality". There are five Protection Levels, PL1-PL5, which are summarized in Figure 12.1. Based on this information, and input provided by the customer, PL3 seems like the primary target for security constraints for the SE testbed. A review of the PL3 requirements and an initial evaluation for the SE testbed (i.e., PL3 and OpenStack) is given in Appendix A.

12.2.5 Networking

There are many very promising options for networking due to the increased demand for cloud based services. This will help drive the technology for use in smaller markets like HPC. Additionally, several limitations or challenges were recognized with current networking technologies ranging from vendor dependence, technology complexity & management overheads and scalability.

Reconfigurable Network The growth of SDN is pushing enabling interesting experiments in the network infrastructure, without having to have direct access to the internal network software in the switch. The ability to use OpenFlow and other standards based interfaces to dynamically change the network enables more agile reconfiguration of the network. Thus, enabling per-tenant configurations that would otherwise require manual interactions by network operations staff.

OpenStack The maturing of OpenStack is yielding benefits for networking functionality. The Neutron interfaces provide a standard method for configuring the network in a compute environment. This also allows other projects to focus on new functionality and leverage the Neutron interfaces to integrate with the network. This is also a way for vendors to expose their functionality in a relatively portable manner, e.g., Arista's ML2 plugin for Neutron. This facilitates the creation of on-demand networking that can be tailored to particular multi-tenant use cases.

Monolithic Vendor Dependence The first notable limitation is the monolithic vendor dependence. Large-scale data centers require routers and switches to meet the needs of the core network. The number of vendors providing these large-scale solutions is small and each have developed proprietary SDN technologies alongside open standards such as OpenFlow. The adoption of proprietary SDN interfaces to orchestrate these large-scale resources may result in vendor lock-in. Given this, insulating applications from these proprietary SDN interfaces should be a top priority either by only exposing open standards based APIs or through the development of middleware that insulates the application from the underlying proprietary API. As such, in the context of secure enclaves we will focus on the use open standards based APIs or the use of middleware such as OpenStack Neutron that will then interface with vendor proprietary APIs through Neutron plugins.

Complexity A dynamic reconfigurable network environment will let users, or tenants, request resources to include compute, storage, and networking. This functionality is made possible through standards based APIs that can control the configuration of individual networking components. While an API simplifies the mechanism by which network configuration takes place, the complexity of configuring many individual components to satisfy what might appear to be a simple tenant requirement remains. By using standard templates for common requirements that can be layered upon one another we can manage this complexity while simultaneously ensuring that network security policies can be verified and enforced. Using a template based approach, common low-risk configurations can be configured on-demand by the tenant, while other templates might require approval through a formal change request process. For example one tenant wishing to communicate with another tenant would require both tenants to agree and potentially be approved by a third-party. Once the request is reviewed, the tenant could be authorized to use the specified template.

Scaling Issues Vendors have their own method of managing a large number of devices and different ways of building large non-blocking fabrics. These design paradigms may include:

- Leaf and spine, to host.
- Leaf and spine to top of rack.
- Standard core, distribution, access models.

Each of these designs scale differently and optimal placement of a workload in these fabrics is dependent on a variety of factors [40]. Supporting multiple tenants within these environments while providing optimal data plan performance and scalability to meet tenant requirements will require a thorough understanding of the overall architecture and how compute and storage resources are interconnected within the networking architecture [105]. For the secure enclaves project we will make the simplifying assumption that tenants (enclaves) will be placed on compute resources that are interconnected in a fully non-blocking network. Orchestrating optimal placement of enclaves within alternate networking architectures based on performance and scale requirements, while an interesting challenge, is not a priority in our current work.

Another important aspect of scalability is the number of concurrent isolated enclaves that can be supported within a single network fabric. One mechanism of implementing isolation of enclaves is to map enclaves to one or more distinct VLANs. Under IEEE 802.1Q the maximum number of VLANs is limited to 4,094 (due to a 12-bit VID field minus reserved values 0x000 and 0xFFF). Using this technique would limit the number of supported enclaves within a single fabric to 4,094. Latest generation switching technologies that provide support for VxLAN scale to supporting up to 16 million logical networks. This is accomplished by encapsulating Layer 2 Ethernet frames within Layer 4 UDP packets. Many switch vendors and Open vSwitch are now offering VxLAN support [85, 73, 15].

12.2.6 Secure Storage

We acknowledge that there is a cost in terms of administrative complexity when combining OS isolation layers and filesystem technologies. Where practical, we suggest the use of automation tools such as OpenStack [94] and Puppet [60], both of which have significant momentum in HPC and cloud communities. This is evidenced by the rapid appearance of projects integrating new technologies, such as containers with `user` namespaces as in the `nova-compute-lxd` project [88]. The ability to automate the storage aspects such as with *Dynamic LENT Configuration* adds an additional isolation capability, but in order for it to be feasibly implemented, it must also be able to integrate with the encompassing automation workflow.

A compelling argument could be made for avoiding this complexity and adopting a complete solution stack from a vendor. Vendors such as IBM, Seagate and Oracle strive to deliver a complete secure storage solution. They will certainly have fewer layers and individual components to manage than a Lustre filesystem with virtualization-based isolation layered on. The disadvantage is in terms of flexibility. A Lustre deployment comes with enormous flexibility for customization, and features are continually being released by the community. The vendor based solutions to secure storage, even those that support Lustre, will always lag behind in the supported features due to the overheads and delays in certification of the features within their products. That also assumes that the features are aligned with their general product plans, which may differ from individual site specific objectives and roadmaps. This is effectively comes down to a balance on the "cost of ownership" for the secure storage portion of a system.

Throughout this report we have noted where particular security mechanisms are at odds with the performance of the storage system. This will be an ongoing competition between these goals, but we believe that the isolation-centric approach that we have discussed and evaluated forms a unique and effective solution for secure HPC storage.

12.3 Future Plans

We conclude with a few remarks about plans moving forward and possible avenues for further investigation.

The paravirtualized filesystem (VirtFS) [55] work that we investigated this year showed very good results. The enhanced security model offered by VirtFS called "mapped" would be something worth looking into in future investigations. The "mapped" model 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. This combined with the Filesystem-As-A-Service work that is currently being pursued in the context of the OpenStack Manila project seem like a nice pairing. It would be very interesting to investigate replacing the pNFS based RPC mechanism for exporting the backend filesystem with Manila using a 9pfs based remote procedure call (RPC) mechanism. This could initially be done with the centralized approach currently maintained in the Manila project, and later moved to something more similar to the approach employed in our evaluations where the re-export is done on a per-host basis instead of on a per-controller (central) manner. Note, the DIOD project [31] may be of interest for this Manila/9pfs investigation.

Additional benchmarking that scales up the resources would be interesting. In the context of VMs and IO-Forwarding, it would be useful to see how mulit-stream and increased node count influences the benchmark results for VirtFS/9pfs exporting Lustre to VMs. Another option for further evaluations with VMs and VirtFS would be to see if different virtio drivers effect performance. For example, the `virtio-blk-data-plane` backend could be tested for avoiding the use of the QEMU block layer, and instead using Linux Asynchronous I/O (AIO).

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 [37] to aid in forensics and debugging. These capabilities might also be advantageous for performing audits and even attestation that specific code was executed in [46] user-customizable environments.

Also, 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 access to dedicated physical memory. The Intel Software Guard Extensions (SGX) are another hardware capability that is of particular interest for future evaluations with secure enclaves [53]. 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 hypervisor (KVM or Xen VMM) [112, 62]. A new open-source project LXD is aiming to make use of hardware isolation capabilities as a "lightvisor" for LXC containers in addition to bringing management features such as live migration and OpenStack integration [72, 101].

The area of networking, quantifying the scalability of SDN controllers is an active research question [50]. A through review of the scalability of the Arista SDN functionality employed for our on-demand enclaving experiments would be useful. Also, an investigation of the robustness of the OpenStack Neutron API and SDN API would be useful for determining how well they can withstand attacks or bad/ill-formed requests from clients.

We have done initial isolation testing for network isolation, which should be continued and enhanced. Additionally, a thorough evaluation of storage isolation would be useful. In addition, we are particularly eager to determine whether we can provide enough protections to overcome the lack of file encryption in Lustre, or if alternate methods like disk encryption for at-rest data would be viable. For GPFS, it would be interesting to quantify the overhead of using native encryption in conjunction with the isolation-centric storage architecture outlined in this report. Also, further analysis of security vulnerabilities for selected storage related technologies and vendor products would be interesting.

In our investigation of the security features of Lustre, we found many features that are in-development, where some are nearing release-readiness such as shared key authentication/encryption [30] and dynamic LNET configuration [67], while others are under heavy development as in UID/GID mapping [109], and subtree support [69]. Along with Kerberos support [120], a thorough evaluation of these features, to include scale-up testing, would be very interesting.

Lastly, as noted previously the emerging work in Containers-As-A-Service is of great interest for future work on secure enclaves. As the Magnum and LXD projects stabilize, the results should be evaluated and compared for usability and performance. There are several announcements and roadmap milestones slated for those projects that will be of direct benefit for refinements to the secure enclaves prototype, e.g., OpenStack Magnum, user `namespace` support in Docker 1.7, OpenStack support for LXD and bare-metal containers.

12.4 Final Remarks

In closing, the container-based virtualization showed great promise for providing efficient and secure high-performance compute enclaving. This was demonstrated with filesystem benchmarks (IOR & FIO) that showed the VE achieved near native performance. Additionally, hypervisor-based virtualization can achieve very good performance with IO-Forwarding based on VirtFS (9pfs), which was also demonstrated via filesystem benchmarks (IOR & FIO). The ability to leverage OpenStack and SDN to achieve on-demand network enclaving enables many interesting options for creating user-customizable compute enclaves. Using these capabilities, a prototype for an isolation-centric approach to secure high-performance computing has been created at ORNL and will be further refined in the following year of the project.

Acknowledgments

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 (ORNL).

We would like to thank John Quigley from ORNL's Information Technology Services Division for his assistance on OpenStack and CADES related configurations.

Bibliography

[1] IBM InfoSphere BigInsights Version 2.1.2. *Comparison of HDFS features and GPFS features*. IBM. URL: http://www-01.ibm.com/support/knowledgecenter/SSPT3X_2.1.2/com.ibm.swg.im.infosphere.biginsights.product.doc/doc/over_filesystem_comparison.html.

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

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

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

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

[6] Software driven cloud networking, 2014. Arista Inc. URL: http://www.arista.com/en/products/software-driven-cloud-networking/articletabs/0 [cited 20-dec-2014].

[7] Siamak Azodolmolky. *Software Defined Networking with OpenFlow*, volume 1. Packt Publishing Ltd, first edition, October 2013. URL: https://www.packtpub.com/networking-and-servers/software-defined-networking-openflow [cited 20-dec-2014].

[8] 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.

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

[10] 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.

[11] Brocade Vyatta controller, 2014. Brocade Inc. URL: http://www.brocade.com/products/all/software-defined-networking/brocade-vyatta-controller/index.page [cited 20-dec-2014].

[12] Network Functions Virtualization (NFV), 2014. Brocade Inc. URL: http://www.brocade.com/products/all/network-functions-virtualization/index.page [cited 20-dec-2014].

[13] OpenStack overview, 2014. Brocade Inc. URL: http://www.brocade.com/solutions-technology/technology/openstack/index.page [cited 20-dec-2014].

[14] 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.

[15] Jefferey Butt. Cisco CTO warrior software-only SDN has 'limitations'. eWeek Online Magazine, June 2013. URL: http://www.eweek.com/networking/cisco-cto-warrior-software-only-sdn-has-limitations.html [cited 20-dec-2014].

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

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

[18] Chef: Automation for Web-Scale IT. URL: https://www.chef.io/ [cited 21-dec-2014].

[19] Cinder: Block Storage for OpenStack. URL: https://wiki.openstack.org/wiki/Cinder [cited 29-jan-2015].

[20] Cisco application centric infrastructure, 2014. Cisco Inc. URL: http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html [cited 20-dec-2014].

[21] Cisco plug-in for OpenFlow, 2014. Cisco Inc. URL: http://www.cisco.com/c/en/us/td/docs/switches/datacenter/sdn/configuration/openflow-agent-nxos/cg-nxos-openflow.pdf [cited 20-dec-2014].

[22] Seagate ClusterStor Product Bulletin. URL: http://www.seagate.com/products/enterprise-servers-storage/enterprise-storage-systems/clustered-file-systems/#specs [cited 29-jan-2015].

[23] 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].

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

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

[26] Cumulus Linux hardware compatibility list, 2014. Cumulus Networks. URL: http://cumulusnetworks.com/support/linux-hardware-compatibility-list/ [cited 20-dec-2014].

[27] Protecting Sensitive Compartmented Information Within Information Systems (DCID 6/3) – Manual, May 24, 2000. URL: http://fas.org/irp/offdocs/dcid-6-3-manual.pdf [cited 15-jun-2015].

[28] Dell and the software defined network, 2014. Dell Inc. URL: http://en.community.dell.com/techcenter/networking/w/wiki/4904.dell-and-the-software-defined-network [cited 20-dec-2014].

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

[30] Andreas Dilger. *Lustre File System: IU Shared Key Authentication and Encryption*. Open SFS, 2013. URL: https://jira.hpdd.intel.com/browse/LU-3289.

[31] diod: An I/O forwarding server based on the 9P protocol. URL: https://code.google.com/p/diod/ [cited 1-feb-2015].

[32] diod source code at github. URL: https://github.com/chaos/diod [cited 1-feb-2015].

[33] diod performance tests with Lustre. URL: https://code.google.com/p/diod/wiki/performance [cited 5-feb-2015].

[34] Computer Security Division. *Compliance With NIST Standards And Guidelines*. Nationa Institute for Standards and Technology, 2014. URL: http://csrc.nist.gov/groups/SMA/fisma/compliance.html.

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

[36] Jim Duffy. Cisco, Arista disaggregating? Network World Online Magazine, 2014. URL: http://www.networkworld.com/article/2844941/cisco-subnet/cisco-arista-disaggregating.html [cited 20-dec-2014].

[37] 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.

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

[39] Open Networking Foundation. Software-defined networking: The new norm for networks, April 2012. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf [cited 20-dec-2014].

[40] Dan Froelich. PCI express 4.0 electrical previews parts i & ii, 2014. PCI SIG. URL: https://www.pcisig.com/developers/main/training_materials/get_document?doc_id=b5e2d4196218ec017ae03a8a596be9809fcd00b5 [cited 20-dec-2014].

[41] 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.

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

[43] Douglas Gourlay. Arista: Making SDN a Reality, 2013. URL: http://www.bradreese.com/blog/4-1-2013.pdf [cited 21-jun-2015].

[44] *General Parallel File System (GPFS) 3.5 System Administration for Linux*, November 2013. Avanet Services IBM Training Student Notebook (Course code H005 ERC 1.0).

[45] Andreas Grünbacher. POSIX Access Control Lists on Linux. In *Proceedings of the USENIX Annual Technical Conference*, pages 259–272. USENIX, June 2003. URL: http://users.suse.com/~agruen/acl/linux-acls.

[46] 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.

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

[48] 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].

[49] SDN infrastructure technology, 2014. HP Inc. URL: http://goo.gl/XLErKS [cited 20-dec-2014].

[50] Jie Hu, Chuang Lin, Xiangyang Li, and Jiwei Huang. Scalability of control planes for software defined networks: Modeling and evaluation. In *Proceedings of the IEEE/ACM International Symposium on Quality and Service (IWQoS'14)*. IEEE, 2014. URL: http://www.cs.iit.edu/~xli/paper/Conf/scale-SDN-IWQOS14.pdf [cited 20-dec-2014].

[51] 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.

[52] IBM. *GPFS Advanced Administration Guide*. IBM Knowledge Center, 2014. URL: http://www-01.ibm.com/support/knowledgecenter/#!/SSFKCN/gpfs41/gpfs.v4r1_welcome.html.

[53] Intel® Corporation. *Intel® Software Guard Extensions Programming Reference*, October 2014. Order Number: 329298-002US. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf [cited 22-jun-2015].

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

[55] 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.

[56] Software Defined Networking, 2014. Juniper Inc. URL: http://www.juniper.net/us/en/products-services/sdn/index.page [cited 20-dec-2014].

[57] 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.

[58] Frank Kraemer. Software defined storage in action with GPFS v4.1, October 2014. Presentation at Linux Foundation's CloudOpen Europe 2014 Conference, Düsseldorf, Germany. URL: http://events.linuxfoundation.org/sites/events/files/slides/SDS-in-action-with-GPFSv41-kraemerf-102014.pdf [cited 27-jan-2015].

[59] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxcluster: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, May 1986. URL: http://doi.acm.org/10.1145/214419.214421, doi:10.1145/214419.214421.

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

[61] 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.

[62] 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.

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

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

[65] Community Lustre Roadmap. OpenSFS community Lustre Roadmap. URL: http://lustre.opensfs.org/community-lustre-roadmap [cited 31-jan-2015].

[66] Lustre community portal. URL: http://lustre.opensfs.org [cited 28-jan-2015].

[67] Lustre Ticket LU-2456: Dynamic LNet Config Main Development Work. Descr: This ticket has been created to track the main development work for the Dynamic LNet Config project. URL: https://jira.hpdd.intel.com/browse/LU-2456 [cited 06-feb-2015].

[68] Lustre Ticket LU-3291: IU UID/GID Mapping Feature. Descr: Tracking bug for Indiana University's UID/GID mapping and cluster project. URL: https://jira.hpdd.intel.com/browse/LU-3291 [cited 27-jan-2015].

[69] Lustre Ticket LU-5989: add subdirectory mounting support for Lustre. Descr: add subdirectory mounting support for Lustre. URL: https://jira.hpdd.intel.com/browse/LU-5989 [cited 31-jan-2015].

[70] Lustre hands-on at SC2011. Lustre Hands-On at SC2011 Presentation. No Links to proceedings. URL: http://cdn.opensfs.org/wp-content/uploads/2011/11/Lustre-hands-on-SC2011.pdf [cited 01-jan-2015].

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

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

[73] Bob Lynch. OpenFlow: Can it scale? SDN Central, June 2013. URL: https://www.sdncentral.com/technology/OpenFlow-sdn/2013/06/ [cited 20-dec-2014].

[74] The Magnum Containers-as-a-Service Project for OpenStack. URL: https://wiki.openstack.org/wiki/Magnum [cited 18-jun-2015].

[75] Manila: Shared file system service for OpenStack. URL: https://wiki.openstack.org/wiki/Manila [cited 29-jan-2015].

[76] Welcome to Manila: An OpenStack File Share Service, 2014. Presentation from Juno (Atlanta) Summit. URL: https://wiki.openstack.org/wiki/Manila/JunoSummitPresentation [cited 30-jan-2015].

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

[78] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. Technical report, March 2008. Stanford University, University of Washington, MIT, Princeton University, University of California Berkeley, Washington University in St. Louis. URL: http://archive.openflow.org/documents/openflow-wp-latest.pdf [cited 20-dec-2014].

[79] 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].

[80] Mellanox OpenStack and SDN/OpenFlow Solution Reference Architecture, 2014. Mellanox Inc. URL: http://www.mellanox.com/sdn/pdf/Mellanox-OpenStack-OpenFlow-Solution.pdf [cited 20-dec-2014].

[81] Mellanox's software defined networking (SDN), 2014. Mellanox Inc. URL: http://www.mellanox.com/sdn/ [cited 20-dec-2014].

[82] 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].

[83] Thomas D. Nadeau and Ken Gray. *SDN: Software Defined Networks*. O'Reilly Media, first edition, September 2013.

[84] ProgrammableFlow networking, 2014. NEC Inc. URL: http://www.necam.com/SDN/ [cited 20-dec-2014].

[85] The scaling implications of SDN, June 2011. NetworkHeresey.com. URL: http://networkheresy.com/2011/06/08/the-scaling-implications-of-sdn/ [cited 20-dec-2014].

[86] B.Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994. doi:10.1109/35.312841.

[87] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. RFC-4120: The Kerberos Network Authentication Service (V5), July 2005. URL: http://www.ietf.org/rfc/rfc4120.txt [cited 31-jan-2015].

[88] nova-compute-lxd source code at github. URL: https://github.com/zulcss/nova-compute-lxd [cited 5-feb-2015].

[89] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, Third 2014. doi:10.1109/SURV.2014.012214.00180.

[90] FAQ: What is OpenDaylight?, 2014. OpenDaylight.org. URL: http://www.opendaylight.org/project/faq#1 [cited 20-dec-2014].

[91] OpenDaylight technical overview, 2014. OpenDaylight.org. URL: http://www.opendaylight.org/project/technical-overview [cited 20-dec-2014].

[92] OpenSFS. *About OpenSFS*. OpenSFS, 2015. URL: http://opensfs.org/about/.

[93] OpenSFS. *Lustre® File System, Version 2.4 Released*. OpenSFS, 2015. URL: http://opensfs.org/press-releases/lustre-file-system-version-2-4-released.

[94] OpenStack: The Open Source Cloud Operating System. URL: https://www.openstack.org/software [cited 4-feb-2015].

[95] OpenStack Operations Guide, 2014. Abstract: This book provides information about designing and operating OpenStack clouds. URL: http://docs.openstack.org/openstack-ops/content/index.html [cited 21-jun-2015].

[96] 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].

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

[98] Oracle; Intel. *Lustre File System: Operations Manual Version 2.0*, 2011. URL: https://wiki.hpdd.intel.com/display/PUB/Documentation.

[99] Sarp Oral, David A. Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S. Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, and Ross Miller. *OLCF's 1 TB/s, Next-Generation Lustre File System*. OLCF at ORNL, 2013. URL: https://cug.org/proceedings/cug2013_proceedings/includes/files/pap151.pdf [cited 27-jan-2015].

[100] Adrian Otto and Steven Dake. Magnum: Containers-as-a-Service for OpenStack, May 18-22, 2015. Presentation at OpenStack Summit, Vancouver, Canada. URL: https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/magnum-containers-as-a-service-for-openstack [cited 18-jun-2015].

[101] James Page and Ryan Harper. LXD vs KVM: Getting closer to the metal with the LXD lightervisor, May 18-22, 2015. Presentation at OpenStack Summit, Vancouver, Canada. URL: https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/lxd-vs-kvm [cited 18-jun-2015].

[102] January 2015. Personal communication with administrative staff deploying and maintaining leadership class Lustre installation at ORNL.

[103] Grégoire Pichon. Experiments with io proxies over lustre, September 23, 2014. Presentation at the 2014 Lustre Administrators and Developers Workshop (LAD'14). URL: http://www.eofs.eu/fileadmin/lad2014/slides/18_Gregoire_Pichon_LAD2014_IOProxies_over_Lustre.pdf [cited 1-feb-2015].

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

[105] Arjun Roy, Kenneth Yocum, and Alex C. Snoeren. Challenges in the emulation of large scale software defined networks. University of California, San Diego, 2013. URL: http://cseweb.ucsd.edu/~snoeren/papers/forgery-apsys13.pdf [cited 20-dec-2014].

[106] 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.

[107] Wikipedia: Serial ATA. URL: http://en.wikipedia.org/wiki/Serial_ATA [cited 1-feb-2015].

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

[109] Stephen Simms and Josh Walgenbach. Scope Statement For UID/GID Mapping in Lustre 2.X, November 10, 2012. Revision v2. URL: http://wiki.opensfs.org/images/3/31/UID_GID_Scope_Statement_v2.pdf [cited 27-jan-2015].

[110] 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.

[111] Swift: Object Storage for OpenStack. URL: https://wiki.openstack.org/wiki/Swift [cited 29-jan-2015].

[112] 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.

[113] Visolve SSH Team. *OpenSSH Whitepaper*. Visolve. URL: http://www.visolve.com/ssh.php#Kerberos_Authentication.

[114] 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].

[115] Vivek Twari. SDN and OpenFlow for beginners with hands on labs. First edition. URL: http://www.amazon.com/SDN-OpenFlow-beginners-hands-labs-ebook/dp/B00EZE46D4/ref=sr_1_1?ie=UTF8&qid=1410613111&sr=8-1&keywords=SDN [cited 20-dec-2014].

[116] 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.

[117] 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.

[118] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, National Center for Computational Sciences, April 2009. URL: http://wiki.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf [cited 1-feb-2015].

[119] Wikipeda. *Generic Security Services Application Program Interface*. OpenSFS, 2015. URL: http://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface.

[120] Wikipeda. *Lustre GSSAPI/Kerberos Repair*. OpenSFS, 2015. URL: http://wiki.opensfs.org/Lustre_GSSAPI/Kerberos_Repair [cited 31-jan-2015].

[121] Wikipedia. *Federal Information Processing Standards*. Wikipedia. URL: http://en.wikipedia.org/wiki/Federal_Information_Processing_Standards.

[122] RFC-3920: Extensible Messaging and Presence Protocol (XMPP): Core, October 2004. IETF Network Working Gropu, P. Saint-Andre, Ed. URL: http://www.ietf.org/rfc/rfc3920.txt [cited 21-dec-2014].

[123] Xyratex. URL: http://www.xyratex.com [cited 28-jan-2015].

[124] S.H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, February 2013. doi:10.1109/MCOM.2013.6461198.

[125] 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.

[126] Oracle ZFS Product Bulliten. URL: https://go.oracle.com/LP=3687?elqCampaignId=6317&src1=ad:pas:go:dg:stor&src2=wwmk14054304mpp001&SC=sckw=WWMK14054304MPP001 [cited 29-jan-2015].

Appendix A

Protection Level

A.1 Protection Level

The DCID 6/3 manual [27] defines a Protection Level (PL) as:

> "*An indication of the implicit level of trust that is placed in a system's technical capabilities. A Protection Level is based on the classification and sensitivity of information processed on the system relative to the clearance(s), formal access approval(s), and need-to-know of all direct and indirect users that receive information from the IS without manual intervention and reliable human review.*" [27]

Table A.1 outlines the requirements for DCID 6/3 Protection Level 3 (PL3) [27]. The table includes annotations in column three to suggest how we meet, or could meet, the needs in the OpenStack based Secure Enclaves (SE) testbed. Note, this is an initial PL3+OpenStack evaluation and will likely be refined as we progress.

The 2^{nd} and 3^{rd} columns are symmetric, with column 3 providing the annotations describing how the current SE testbed meets, or could meet, requirements based on project reports/design and current installation options. Items that seem to be primarily under the customer's control/discretion are marked as "Customer defined". Finally, items that are entirely open or need further consideration are marked in red with "NEEDED / OPEN".

Table A.1. Review of DCID Protection Level 3 requirements and assessment of how these needs are met, or could be met, for OpenStack based SE testbed.

Category	Requirements	How we meet requirements
C1. Access Control	1. Denial of physical access without supervision 2. Procedures in place for controlling access 3. Process/mechanism for users to grant access to data for other users 4. Process/mechanism for user to determine data's sensitivity level	1. Customer defined 2. Customer defined 3. Presumably Swift can do this 4. NEEDED/OPEN
C2. Discretionary Access Control	1. System should use DAC	1. Linux uses this by default
C3. Account Management Procedures	1. Account type identification 2. Account establishment, activation, modification, and termination	1. Covered by Keystone and Linux 2. Linux tools covered local system and Keystone for the rest

Table A.1 – continued from previous page

Category	Requirements	How we meet requirements
C4. Audit Procedures	1. Records must include date/-time, system locale, entity and resources involved, and the action 2. Protection of audit logs 3. Audit data maintained for at least 5 years and reviewed weekly 4. Records must include logon/logoff, access to security-relevant objects and directories, console activities, and system-level accesses by privileged users 5. Include existence/use of audit reduction/analysis tools 6. Audit trail recording changes to its list of user formal access permissions (only needed if #5 is automated) 7. Individual accountability 8. Periodic testing of the security posture	1. Use Auditd 2. Use SELinux 3. Customer defined 4. Use Auditd 5. Use Auditd's suite of tools 6. Use Auditd 7. Use Auditd 8. Customer defined
C5. Identification and Authentication 1 (*must be specified*)	1. Initial authenticator content and administrative procedures for initial authenticator distribution 2. Individual and group authenticators 3. Length, composition, and generation of authenticators 4. Change process 5. Aging of static authenticators 6. History of static authenticators and non-replication of individual authenticators 7. Protection of authenticators with respect to confidentiality and integrity	1. Customer defined 2. Customer defined, but individual authenticators likely to be user passwords/keys 3. NEEDED/OPEN 4. Customer defined, but mechanism is contained in Keystone 5. Customer defined 6. Customer defined 7. SELinux can cover integrity, confidentiality should be covered

Table A.1 – continued from previous page

Category	Requirements	How we meet requirements
C6. Identification and Authentication 2	1. User generated passwords may be verified for strength in an automated way (optional) 2. Remote access users shall employ a strong authentication mechanism	1. NEEDED/OPEN 2. SSH
C7. Least Privilege	1. Each user/process is granted the most restrictive set of privileges or accesses needed for authorized tasks	1. LXC, Docker with capabilities & user namespaces. Users are restricted to unprivileged account
C8. Marking	1. Procedure/mechanism to ensure that either the user or the system marks all data stored/-transmitted to reflect the sensitivity level of that information	1. NEEDED/OPEN
C9. Parameter Transmission	1. Security parameters (e.g., labels or markings) are reliably associated with information exchanged between systems	1. NEEDED/OPEN
C10. Recovery	1. Recoveries should be done in a trusted and secure manner	1. NEEDED/OPEN
C11. Resource Control	1. Memory must be scrubbed prior to allocation/re-allocation	1. Linux ($>= 2.6.30$) can do this prior to re-allocation to another process

Table A.1 – continued from previous page

Category	Requirements	How we meet requirements
C12. Screen Lock	1. There should be screen locking functionality unless impossible to have. Should be able to: (a) Enable explicitly or due to idleness (b) Lock requiring a password to unlock (c) Not be a substitute for logging out	1. Seems out of scope, otherwise it will be NEEDED/OPEN
C13. Separation of roles	1. The ISSO and ISSM should be different people [1]	1. Customer defined
C14. Session controls	1. Users should be notified on login that they will be monitored, recorded, and subject to audit 2. Users advised use of system is giving consent to audit and unauthorized use could result in civil/criminal penalties	1. Not in design, but can be done via to `/etc/motd` or `/etc/issue`
C15. Enforcement of Session Controls	1. Procedures for controlling/auditing concurrent logons from different workstations 2. Station/session time-outs 3. Limited logon retries 4. Some action on unsuccessful logons	1. Use Auditd 2. Modify ClientAliveInterval in `/etc/ssh/sshd_config` 3. Modify MaxAuthTries in `/etc/ssh/sshd_config` 4. Use iptables to block IP

[1] Terminology: The Information System Security Officer (ISSO) is responsible to the ISSM for the maintained operational security of the Information System (IS). The Information System Security Manager (ISSM) is the manager responsible for an organization's IS security program.

Table A.1 – continued from previous page

Category	Requirements	How we meet requirements
C16. Storage (one of following)	1. Information stored in an area approved for open storage 2. Information stored in an area approved for continuous personnel access 3. Information secured as appropriate for closed storage 4. Information encrypted via NSA-approved encryption mechanism as appropriate for the data	1. Customer defined 2. Customer defined 3. Customer defined 4. GPFS supports encryption, Lustre does not support encryption
C17. Data transmission (one of following)	1. Distributed only within area approved for open storage 2. Distributed via a PDS 3. Distributed using NSA-approved encryption 4. Distributed using a trusted courier	1. Customer defined 2. Customer defined 3. Not currently in our docs, focus on isolation. NEEDED/OPEN 4. n/a
C18. Documentation	1. A system security plan 2. A security concept of operations 3. Guide/manual for the system's privileged users (a) Configuring, installing and operating the system (b) Making optimum use of security features (c) Identifying known security vulnerabilities related to the system 4. Certification test plans and procedures detailing the implementation of the features/assurances for the required PL 5. Report of test results 6. General user's guide	1. Customer defined 2. Partially covered in SE report, other pieces up to customer 3. Most covered in SE reports 4. Customer defined 5. Some tests in SE reports 6. Some of user's guide can be taken from SE reports

Table A.1 – continued from previous page

Category	Requirements	How we meet requirements
C19. System Assurance	1. Features/procedures to validate the integrity/operation of software, hardware, firmware 2. Features/procedures for protecting OS integrity 3. Control of access to software, hardware, and firmware 4. Assurance of the integrity of above 5. Isolation of (above) by means of partition/domains/etc. 6. Using up-to-date vulnerability assessment tools to test the system	1. Do not have, but could if necessary. NEEDED/OPEN 2. Use SELinux 3. Customer defined 4. NEEDED/OPEN 5. Use of VEs / VMs 6. Do not have this, but may be covered in Y2 robustness deliverables. NEEDED/OPEN
C20. Testing	1. n/a	1. Customer defined

Appendix B

Docker

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

```
# $Id: Dockerfile.centos7cxx 825 2015-06-11 19:31:04Z 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 B.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 C

libvirt

C.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 C.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>
```

Appendix D

Network Enclaving Demo

D.1 On-Demand Network Enclaving via SDN & Neutron

The following provides a brief demonstration of on-demand enclaving with OpenStack's Neutron. The screen captures were taken while performing a test to show the functionality in the ORNL SE testbed. The text windows show details from OpenStack Command Line Interface (CLI) utilities and details from the underlying Arista switch via its CLI interface. The output from the Arista switch is only for demonstration purposes, a user would not need to interface with the switch directly. For reference, the Arista CLI screen captures all have the prompt set to "`CADES-OS-ARISTA#`".

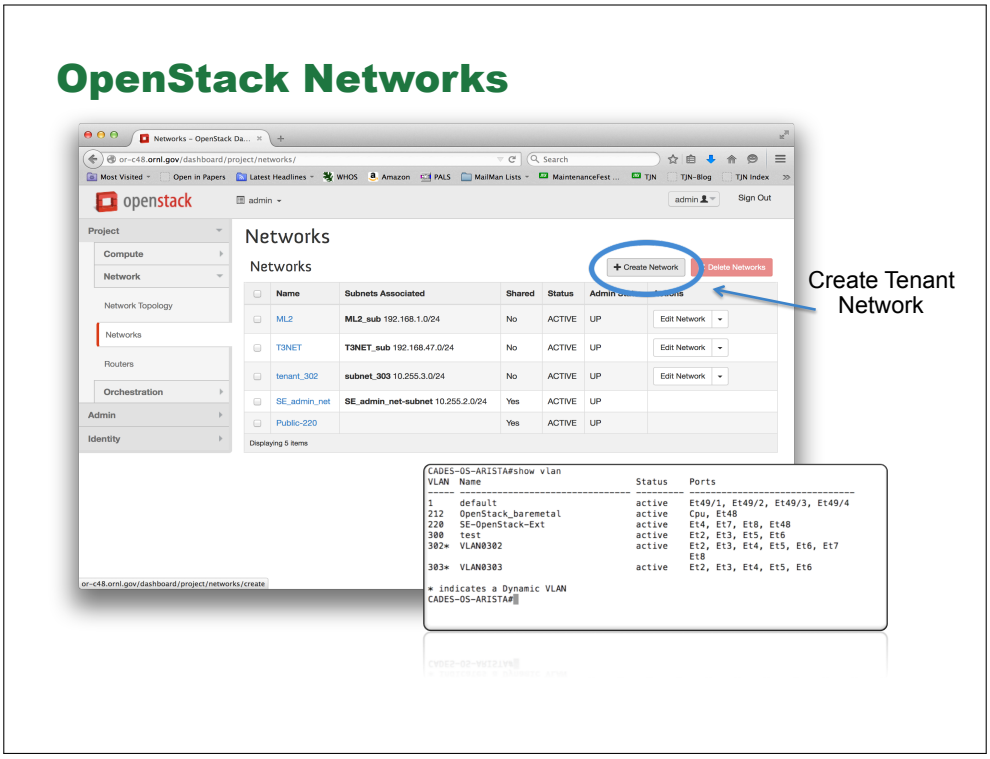


Figure D.1. Demo: OpenStack Networks

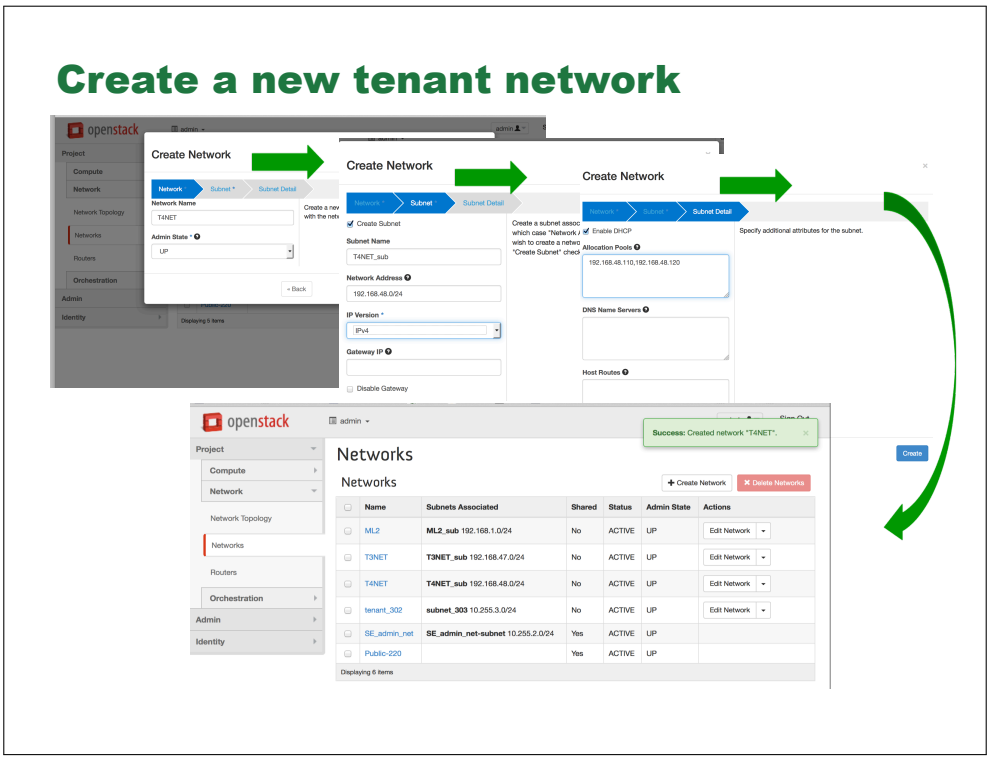


Figure D.2. Demo: Create a new tenant network (1/2)

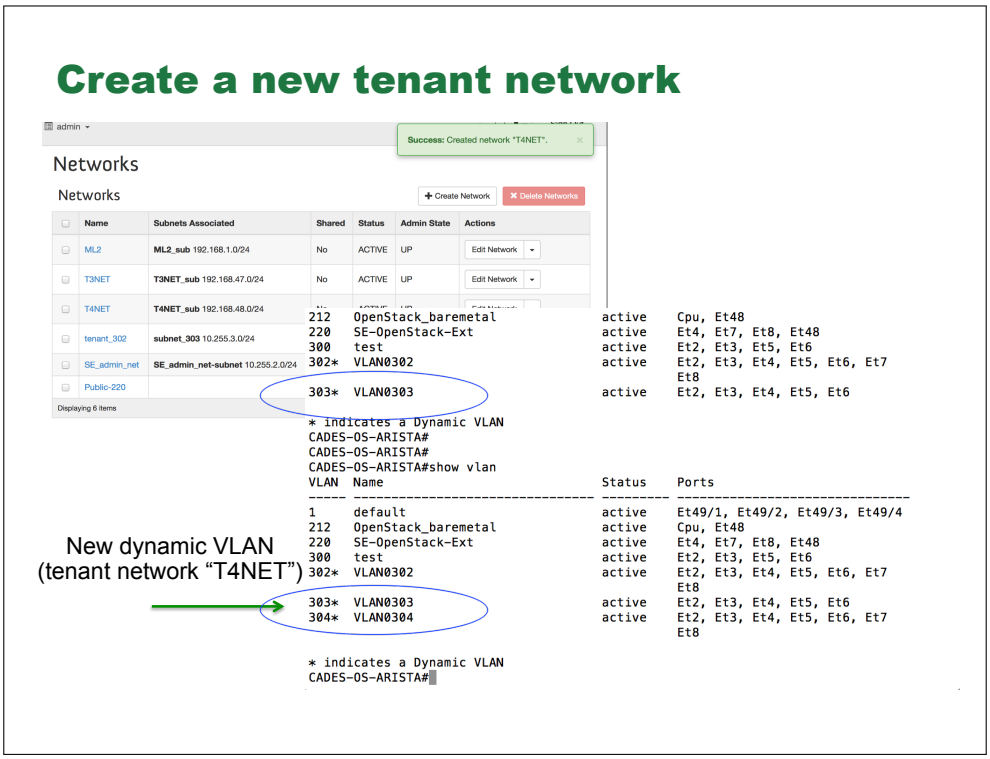


Figure D.3. Demo: Create a new tenant network (2/2)

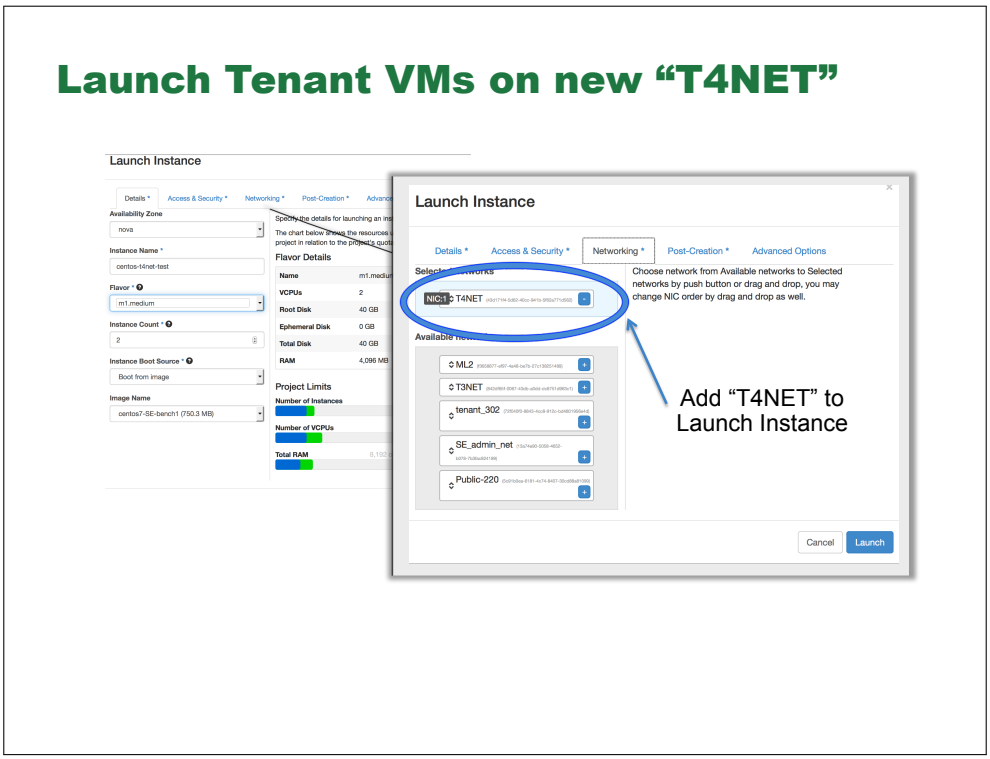


Figure D.4. Demo: Launch Tenant VMs on new “T4NET”

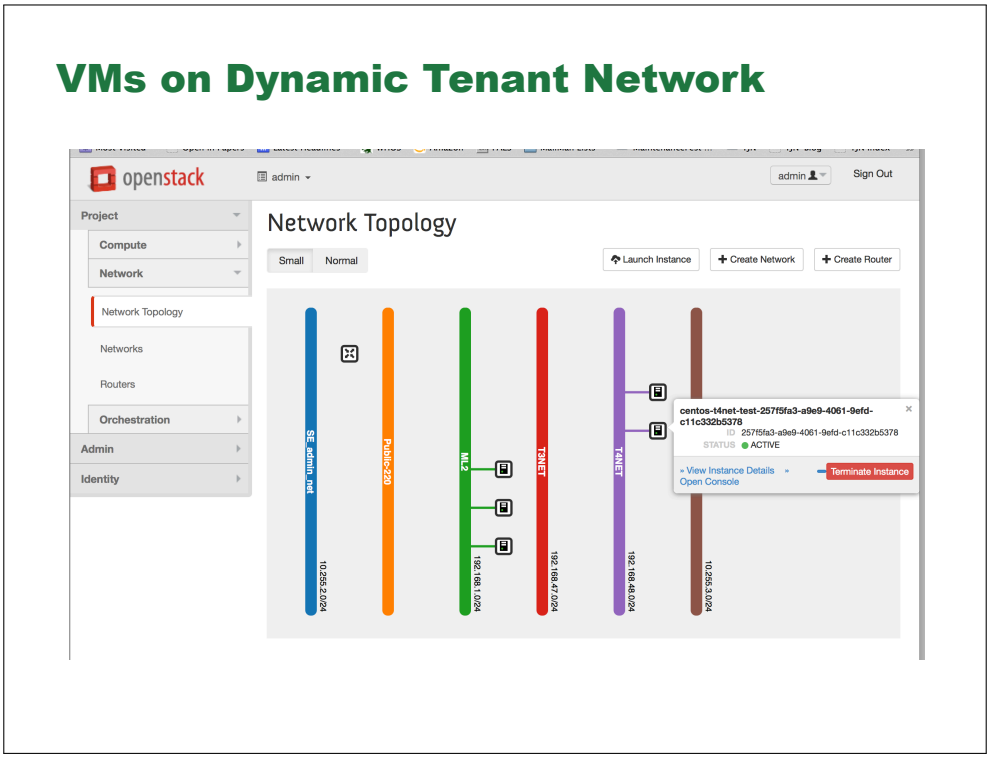


Figure D.5. Demo: VMs on Dynamic Tenant Network

Horizon VM Console & Ping Test

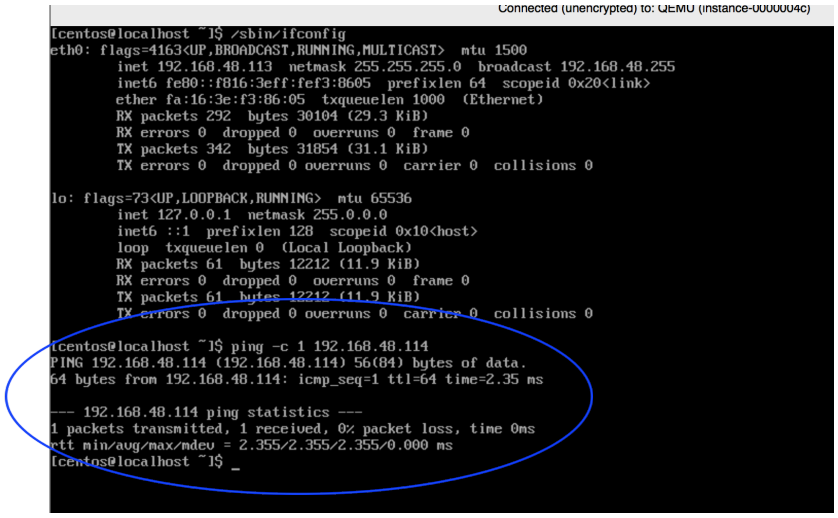


Figure D.6. Demo: Horizon VM Console & Ping Test

Details on Neutron Networks & Nova VMs

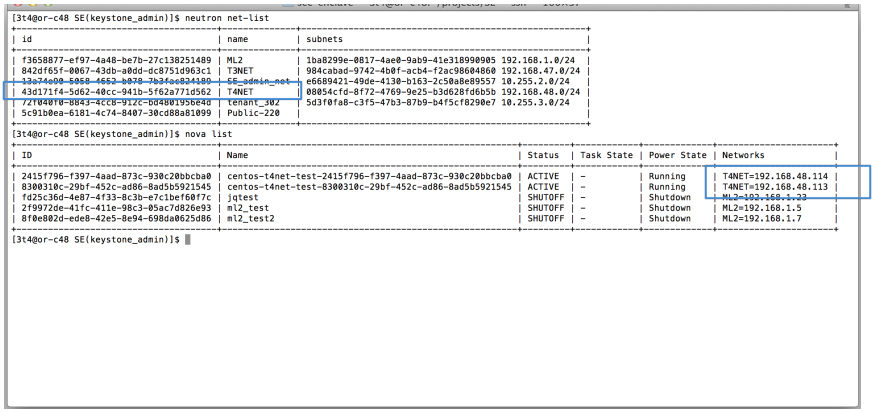


Figure D.7. Demo: Details on Neutron Networks & Nova VMs

Show Active VMs and Networks (Arista)

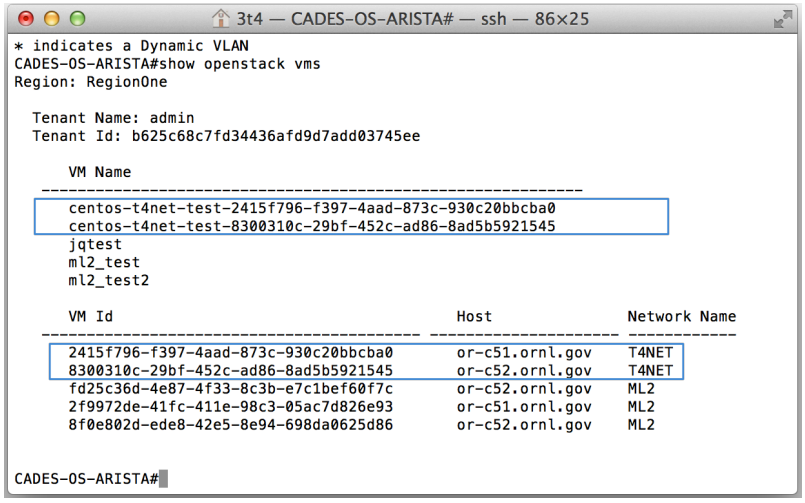
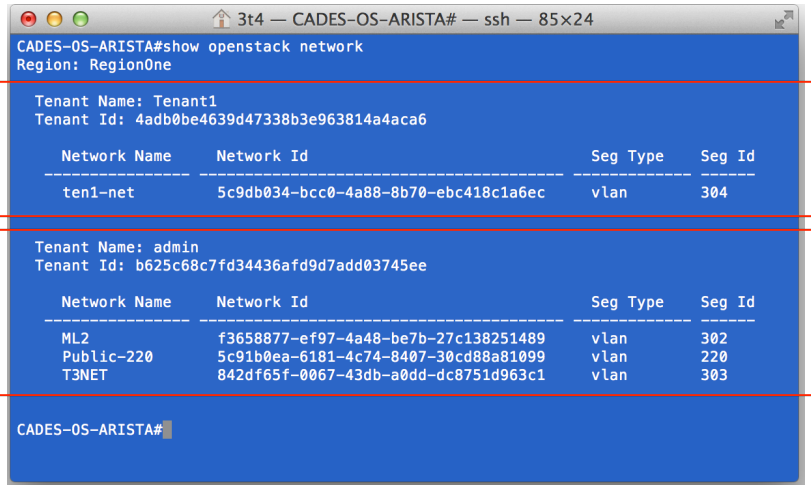


Figure D.8. Demo: Show Active VMs and Networks (Arista)

Separate example with different Tenants*



*** Note: Separate experiment that shows multiple networks for different Tenants (Tenant1 & admin).**

Figure D.9. Demo: Separate example with different Tenants

Terminate VM Instances

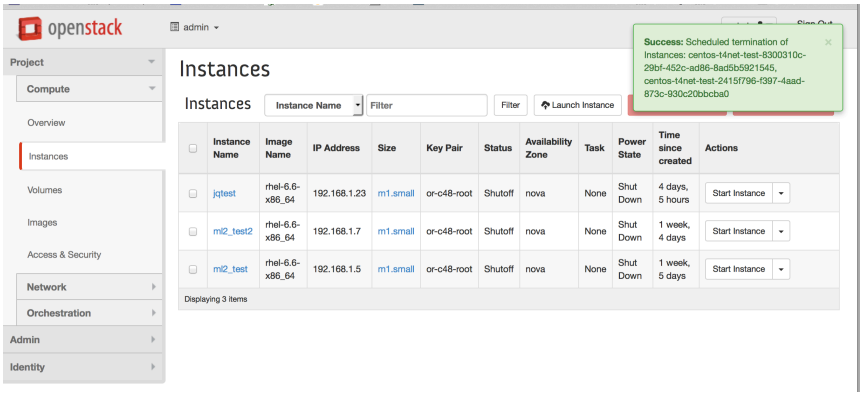


Figure D.10. Demo: Terminate VM Instances

Show Terminated VMs Are Gone (Arista)

VMS Gone but Network remains (VLAN304)

```
----- -------------------------------- --------- -------------------------------
1     default                          active    Et49/1, Et49/2, Et49/3, Et49/4
212   OpenStack_baremetal              active    Cpu, Et48
220   SE-OpenStack-Ext                 active    Et4, Et7, Et8, Et48
300   test                             active    Et2, Et3, Et5, Et6
302*  VLAN0302                         active    Et2, Et3, Et4, Et5, Et6, Et7
                                                 Et8
303*  VLAN0303                         active    Et2, Et3, Et4, Et5, Et6
304*  VLAN0304                         active    Et2, Et3, Et4, Et5, Et6

* indicates a Dynamic VLAN
CADES-OS-ARISTA#show openstack vms
Region: RegionOne

  Tenant Name: admin
  Tenant Id: b625c68c7fd34436afd9d7add03745ee

   VM Name    VM Id                                Host            Network Name
   ---------- ------------------------------------ --------------- ------------
   jqtest     fd25c36d-4e87-4f33-8c3b-e7c1bef60f7c  or-c52.ornl.gov  ML2
   ml2_test   2f9972de-41fc-411e-98c3-05ac7d826e93  or-c51.ornl.gov  ML2
   ml2_test2  8f0e802d-ede8-42e5-8e94-698da0625d86  or-c52.ornl.gov  ML2


CADES-OS-ARISTA#
```

Figure D.11. Demo: Show Terminated VMs Are Gone (Arista)

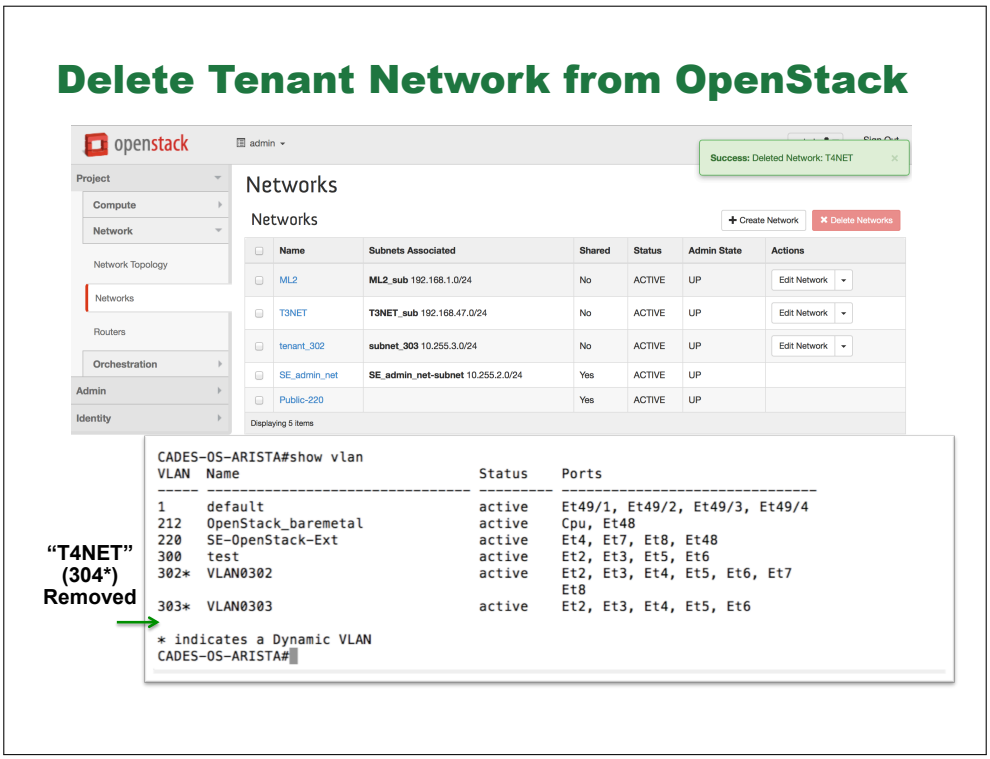


Figure D.12. Demo: Delete Tenant Network from OpenStack

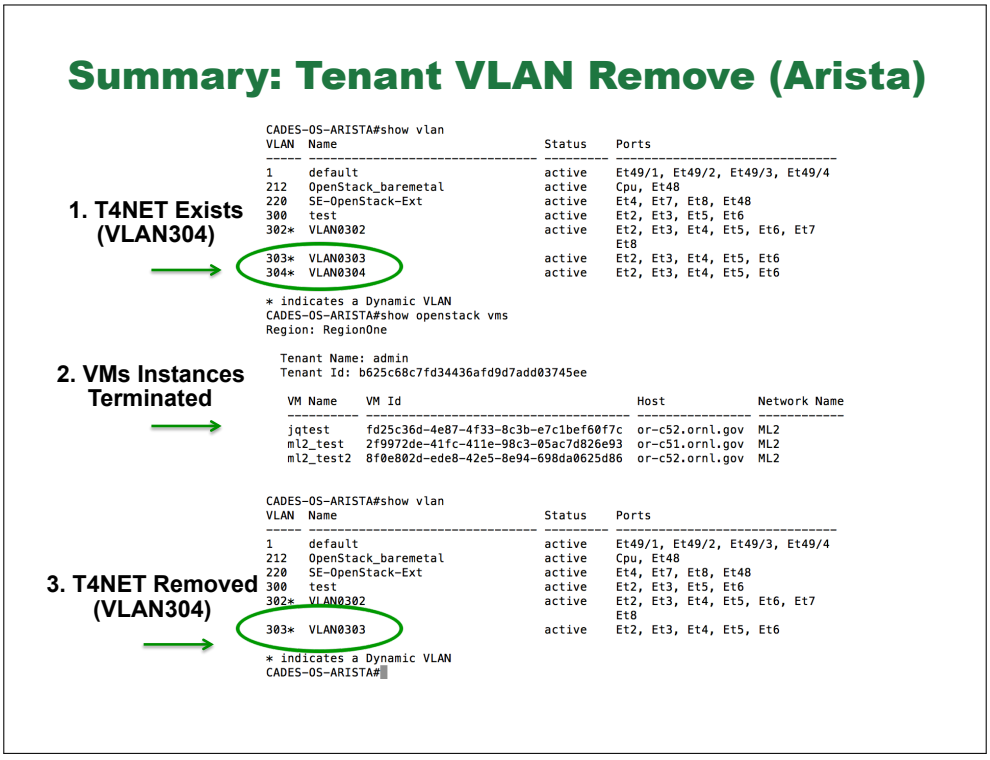


Figure D.13. Demo: Summary - Tenant VLAN Remove (Arista)