Security of OS-level virtualization technologies

Elena Reshetova[1], Janne Karhunen[2], Thomas Nyman[3], N. Asokan[4]

[1] Intel OTC, Finland
[2] Ericsson, Finland
[3] University of Helsinki, Finland
[4] Aalto University and University of Helsinki, Finland

Abstract. The need for flexible, low-overhead virtualization is evident on many fronts ranging from high-density cloud servers to mobile devices. During the past decade *OS-level virtualization* has emerged as a new, efficient approach for virtualization, with implementations in multiple different Unix-based systems. Despite its popularity, there has been no systematic study of OS-level virtualization from the point of view of security. In this paper, we conduct a comparative study of several OS-level virtualization systems, discuss their security and identify some gaps in current solutions.

1 Introduction

During the past couple of decades the use of different virtualization technologies has been on a steady rise. Since IBM CP-40 [19], the first virtual machine prototype in 1966, many different types of virtualization and their uses have been actively explored both by the research community and by the industry. A relatively recent approach, which is becoming increasingly popular due to its light-weight nature, is *Operating System-Level Virtualization*, where a number of distinct user space instances, often referred to as *containers*, are run on top of a shared operating system kernel. A fundamental difference between OS-level virtualization and more established competitors, such as Xen hypervisor [24], VMWare [47] and Linux *Kernel Virtual Machine* [28] (KVM), is that in OS-level virtualization, the virtualized artifacts are global kernel resources, as opposed to hardware. This allows multiple virtual environments to share a common host kernel and utilize underlying OS interfaces. As a result, OS-level virtualization incurs less CPU, memory and networking overhead, which is important not only for *High Performance Computing* (HPC), such as dense cloud configurations, but also for resource constrained environments such as mobile and embedded devices. However, the main disadvantage of OS-level virtualization is that each container can only contain a system of the same type as the host environment, e.g. Linux guests on a Linux host.

An important factor to take into account in the evaluation of the effectiveness of any virtualization technology is the level of *isolation* it provides. In the context of OS-level virtualization isolation can be defined as separation between containers, as well as the separation between containers and the host. In order to systematically compare the level of isolation provided by different OS-level virtualization solutions, one first needs to establish a common system model.

The goal of this study is to propose a generic model for a typical OS-level virtualization setup, identify its security requirements, and compare a selection of OS-level virtualization solutions with respect to this model. While other technologies as HW supported secure storage, various encryption primitives and specific CPU/memory features can enhance the security of OS-level virtualization solutions, they are left out of the scope of this paper due to the space limitations. To the best of our knowledge this is the first study of this kind that focuses on the security aspects of OS-level virtualization technologies. We base our analysis on information collected from the documentation and/or source code of the respective systems, wherever possible. As a result of this comparison section 6 identifies a number of gaps in the current implementation of Linux OS-level virtualization solutions.

2 System model

In Figure 1(a) we present a system model for a typical container setup. There are a number of containers $C_1 \ldots C_n$ that run on a single physical host machine. The OS kernel is shared among all the containers, but the extent of shared host user space depends on a concrete setup (see Table 1):

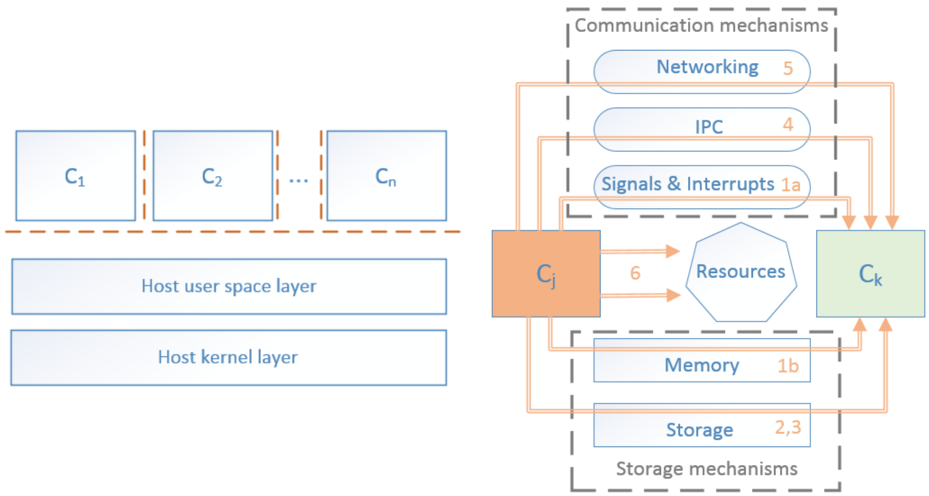
Full OS installation & management corresponds to the most common case when the host user space layer comprises a complete OS installation with the container management layer on top. In this case some host resources may be shared between the host and one or more containers via bind-mounts [25] or overlay filesystems [20]. Each container can be one of two types:

- *Application containers* have a single application or service instance running inside. They are most commonly used as sandboxes to contain damage in case an application or a service misbehaves.
- *System containers* have an entire OS user space installation and are commonly used for server consolidation, where a set of distinct physical servers are substituted with a single physical server running a number of distinct virtual environments.

Lightweight management corresponds to the case where the host user space layer consists of merely a light-weight management layer used to initialize and run containers. This setup can be argued to be more secure, as it exhibits a reduced attack surface compared to a complete underlying host system. Again, each container can be one of two types:

- *Direct application/service setup* refers to the case when only a single application or service is installed in the container. It is more suitable for application isolation scenarios in which, for instance, a banking application is run in a separate container isolated from the rest of a less trusted OS running in another container.

– *Direct OS setup* refers to the case when a container runs an entire OS user space installation. It can provide an end-user the appearance of simultaneously running multiple OS instances, and is therefore well suited for the Multi-OS experience that allows end-users the ability to use applications and services from different OS variants on the same device.



(a) System model (b) Attacker model

Fig. 1. OS-level virtualization

Host user space layer	**Container**	
	Application/Service	*Full OS installation*
Full OS installation & management	Application container	System container
Lightweight management	Direct Application/Service setup	Direct OS setup

Table 1. Types of OS virtualization setups

The system model described above intentionally omits cases where containers $C_1 \ldots C_n$ are not independent, but arranged in a hierarchical structure. While some systems, such as FreeBSD jails [27], allow such setups, they are rarely used in practice and are therefore left beyond the scope of this paper.

3 Attacker model and security requirements

The attacker is assumed to have full control over a certain subset $\bar{C}$ of containers. The remaining set C is assumed to be in the control of legitimate users. We also assume that the attacker has a certain subset of superuser privileges. This is needed in order to allow a container's administrator to setup the desired settings, for example mounting new filesystems or performing a network configuration inside a container. The goals of the attacker can be classified as follows:

- **Container compromise:** compromise $C_k \in C$ by means of illegitimate data access, *Man-in-the-Middle* (MitM) attacks or by affecting the control flow of instructions executed in $C_k \in C$.
- **Denial of Service:** disturb normal operation of the host or $C_k \in C$.
- **Privilege escalation:** obtain a privilege not originally granted to a $C_j \in \bar{C}$.

The above goals can be achieved via different types of attacks that can be roughly classified into distinct groups based on the interfaces available in a typical Single UNIX Specification (SUS) compliant OS [44]. These attack groups can be further arranged into two classes based on the type of underlying mechanism: *attacks via communication mechanisms* and *attacks via storage mechanisms* (see Figure 1(b)). From this classification we derive a set of security requirements that each OS-level virtualization solution needs to fulfill. In the description below, numbers in parenthesis refer to arrows in Figure 1(b).

Separation of processes is a fundamental requirement that aims to isolate processes running in distinct containers to prevent $C_j \in \bar{C}$ from influencing $C_k \in C$ using interfaces provided by the operating system for process management, such as signals and interrupts (1a). In addition, it might be possible to directly access the memory of a process running in $C_k \in C$ by using special system calls, e.g. the *ptrace()* system call allows a debugger process to attach and monitor the memory of a debugged process (1b).

Filesystem isolation is required in order to prevent illegitimate access to filesystem objects belonging to $C_k \in C$ or the host (2).

Device isolation should protect device drivers shared between different containers and a host. Such drivers present another significant attack vector because they expose interfaces (3) to code running into the kernel space, which may be abused to gain illegitimate data access, escalate privileges or mount other attacks.

IPC isolation is needed in order to prevent $C_j \in \bar{C}$ from accessing or modifying data belonging to $C_k \in C$ being transmitted over different IPC channels (4). Such channels include traditional System V IPC primitives, such as semaphores, shared memory and message queues as well as POSIX message queues.

Network isolation aims to prevent attacks by $C_j \in \bar{C}$ via available network interfaces (5). In particular, an attacker can attempt to eavesdrop on or modify network traffic of the host or $C_j \in \bar{C}$, perform MitM attacks etc.

Resource management provides a way to limit the amount of resources available to each container depending on the system load. This is needed in order to prevent an attacker from exhausting physical resources available on a device, such as disk space or disk I/O limits, CPU cycles, network bandwidth and memory (6).

4 Comparison

At first, in this section we give a brief historical overview of the development OS-level virtualization solutions and then proceed to compare the chosen state-of-the-art examples in accordance to each of the security requirements described in the previous section.

4.1 Evolution of OS-level virtualization

The history of OS-level virtualization can be traced back to the usage of the *chroot()* system call in the Unix systems. While its primary goal has been to limit visibility of a filesystem root for a set of processes, *chroot()* has never been intended as a security mechanism. Nevertheless, it has often been used as a mechanism to limit filesystem access. Such usage is often referred to as *chroot() jails*. However since the *chroot()* environment is only limited to a filesystem and can be escaped by the privileged user [7], it cannot be used as it is in order to build a full OS-level virtualization solution.

The need for a more robust and complete jail implementation in order to have separate virtual compartments on a single host has motivated the emergence of the **FreeBSD Jails** project [27] in 2000. The same need in Linux led to the **Linux-VServer** project [9] with its first release in 2003. In Linux-VServer, separate virtual environments are referred to as Virtual Private Servers (VPSs). **Solaris Zones/Containers** project [34] was started in 2004 in order to provide a commercial OS-level virtualization solution. The **OpenVZ** project [12], another open source OS-level virtualization solution for Linux, began in 2005. OpenVZ uses the term Virtual Environments (VEs) [46] to refer to the containers. Both the Linux-VServer and OpenVZ projects provide their own set of kernel patches that in turn create additional usage difficulties. Notably, the OpenVZ project was the first one to implement the *Checkpoint and Restart* (CR) functionality for VEs [29]. CR allows processes to be moved between different physical or virtual environments. This can be useful for cluster load-balancing or in high-availability environments, as well as an utility for software development and testing on different UNIX platforms. The **Linux Containers (LxC)** project [10] is the only currently available OS-level virtualization solution for Linux that consists only of a set of user space tools. This is possible because LxC utilizes only those virtualization features already integrated into the upstream Linux kernel. Another differentiating feature of LxC is the ability to use Linux Security Modules (LSMs) [40] to harden a container setup. Apparmor [1] and SELinux [37] profiles are officially supported, but in principle any LSM such as Smack [15] could be used. The **Cells** architecture [22] and the corresponding commercial Cellrox solution [2] are the only open source OS-level virtualization

	container structure	separate namespaces
pros	simplicity, convenience	flexibility, incremental introduction of containerization
cons	possible information duplication, less flexibility	increased complexity
used by	FreeBSD, SolarisZones, Linux-VServer, OpenVZ	Linux-VServer, OpenVZ, LxC, Cells

Table 2. Comparison of containerization approaches

solutions developed specifically for smartphones. The primary design goal behind Cells is to support the *Bring Your Own Device* (BYOD) policy [36] on the Android platform. BYOD allows one physical device to be used simultaneously for personal and business needs resulting in a need of rigid separation between these two environments in order to guarantee user privacy while conforming to enterprise policies. Similar to LxC, Cells utilizes upstream kernel features to isolate virtual phones. However, since Android has some non-standard Linux extensions, the developers of Cells had to implement a number of additional isolation mechanisms.

4.2 Separation of processes

The primary isolation mechanism required from any OS-level virtualization solution is that it is able to distinguish processes running in different containers from those running on the host, limit cross-container process visibility and to prevent memory and signaling-level attacks described in the section 3. The simplest solution to this problem is to embed a container identifier C_i into the process data structure and to check the scope and the permissions of all syscall invocations.

FreeBSD Jails, Solaris Zones, OpenVZ and Linux-VServer implementations follow this approach by linking a structure describing the container to the process data structure. However, unlike FreeBSD and Solaris, the data structures describing OpenVZ and Linux-VServer containers are not used to achieve process separation. They only store related container data such as resource limits and capabilities. Instead, OpenVZ, Linux-VServer, LxC and Cells use *process id (pid) namespaces* that are part of the mainline Linux kernel. A pid namespace is a mechanism to group processes in order to control their ability to see (for example via *proc* pseudo-filesystem) and interact (for example by sending signals) with each other. The pid namespaces also provide pid virtualization: two processes in different pid namespaces may have the same pid.

Having a separate structure describing a container and storing a pointer to it in the process task structure is a convenient way to have all the relevant information concerning the container in one place. However, the upstream Linux kernel has followed a different approach of grouping different kernel resources into separate namespaces and using these namespaces to build containers. This approach incurs additional complexity, but adds the flexibility to choose a combination of namespaces that best fits the desired use case. It also allows gradual introduction of namespaces to an existing system, like the mainline Linux kernel, which also helps in testing and verification of the implementation [17]. Furthermore, it avoids information duplication when both the process and the container structures have similar information. The pros and cons of these two approaches are summarized in Table 2.

In addition to the ability to isolate and virtualize process ids, the upstream Linux kernel also allows virtualization and isolation of the user and group identifiers with the help of *user namespaces*. Typically the root user has all the privileges to perform various system administration tasks and is able to override all access control restrictions. However, it is not desired that a root user running inside a container would be given the privileges of the host root user. Therefore, the Linux user namespace implementation interprets a given Linux capability as authorizing an action within that namespace: for example, the *CAP_SYS_BOOT* capability inside a container grants the authority to reboot that container and not the host. Moreover, many capabilities such as *CAP_SYS_MODULE* cannot be safely granted for container in any meaningful manner. When a process attempts to perform an action guarded by such capability, the kernel always checks if the process possesses this capability in the host user namespace. All Linux OS-level virtualization solutions support the option of starting a new user namespace for each container, but all the related configuration such as mapping the user identifiers between the host and the container must be done manually.

4.3 Filesystem isolation

The filesystem is one of the most important OS interfaces that allows processes to store and share data as well as to interact with one another. In order to prevent filesystem-based attacks described in section 3, it should be possible to isolate the filesystem between containers and to minimize the sharing of the data. The amount of sharing needed between the host and each container depends on the usage scenario. In the case of application isolation, it is not worthwhile to completely duplicate the OS setup inside a container and therefore some parts of the filesystem, such as common libraries, need to be securely shared with the host. On the other hand in the case of server consolidation, quite often it is best to completely separate the filesystems and create container filesystems from scratch.

All Linux-based OS virtualization solutions utilize a *mount namespace* that allows separation of mounts between the containers and the host. The design of upstream Linux mount namespaces[17] has been influenced by private namespaces [33] in Plan 9 from Bell Labs [32]. Namespaces in Plan 9 are file-orientated, and the principal purpose is to facilitate the customization of the environment visible to users and processes. Since all Linux based systems create each container within a new mount namespace, all the internal mount events are only effective inside the given container. However, it is important to underline that the mount namespace by itself is not a security measure. Running a container in a separate mount namespace does not give any additional guarantees concerning the data isolation between the containers since containers inherit the view of filesystem mounts from their parent and thus are able to access all parts of the filesystem similarly.

A typical approach for process filesystem access containment is by using the *chroot()* system call where process is bound within a subtree of the filesystem hierarchy. If desired, resources may be shared with the host by mounting them within the subtree visible inside the container. Since the *chroot()* system call [8] only affects pathname resolution, privileged processes (i.e. processes with the *CAP_SYS_CHROOT* privilege) can escape the chroot jail. This can be done for example by changing the root directory again via *chroot()* to a subdirectory relative to their current working directory. Of the virtualization solutions under comparison, only Cells relies on *chroot()* alone. Some systems, such as Linux-VServer utilize a *Secure chroot barrier* [9] to prevent processes in a VPS from escaping the modified environment.

Another approach, utilized by for instance LxC, is to not only modify the root directory for processes in a container, but modify the *root filesystem* as well. This can be achieved with the Linux specific *pivot_root()* system call [8], which is typically used during boot to change from a temporary root filesystem (e.g. an initrd) to the actual root filesystem. As its name suggests, the *pivot_root()* system call moves the mountpoint of the old root filesystem to a directory under the new root filesystem, and puts the new root filesystem at its place. When done inside a mount namespace, the old root filesystem can be unmounted, thus rendering the host root filesystem inaccessible for processes inside the container, without affecting processes belonging to the root mount namespace on the host system. At the time of writing, the implementation of *pivot_root()* also changes the root directory and current working directory of the process to the mountpoint of the new root filesystem if they point to the old root directory. OpenVZ relies on this behavior and uses the *pivot_root()* system call alone. However, as the behavior with regards to the current root directory and the current working directory remains unspecified, proper usage dictates that the caller of *pivot_root()* must ensure that processes with root directory or current working directory at the old root operate correctly regardless of the behavior of *pivot_root()*. To ensure this, LxC changes the root directory and current working directory to the mountpoint of the new root before invoking *pivot_root()*.

FreeBSD and Solaris also provide a sandbox-like environment for each jail/zone using similar *chroot()*-like calls that are claimed to avoid above mentioned security vulnerabilities [27], [34]. Mounting and unmounting of filesystems is prohibited by default for a process running inside a jail unless different *allow.mount.** options are specified.

A separate user namespace per container can further strengthen the filesystem isolation by mapping the user and group ids to a less privileged range of host uids and groups. Together with a mount namespace and a *pivot_root* environment it strengthens protection against filesystem-based attacks described in 3.

4.4 Device isolation

In Unix, device nodes are special files that provide an interface to the host device drivers. In classical Unix configurations, the device nodes are separated from the rest of the filesystem and their inodes are placed in the */dev* directory. In Linux's case, this task is usually performed by the udevd daemon process issuing the *mknod* system call upon receiving the event from the kernel. Device nodes are security-sensitive since an improperly exposed or shared device inside a container can lead to a number of easy attacks (see section 3). In the simplest example, if a container has an access to */dev/kmem* and */dev/mem* nodes, it is able to read and write all the memory of the host. Thus, in order to isolate containers from one another it is important to prevent containers from creating new device nodes and to make sure that containers are only allowed to access a "safe" set of devices listed below.

Purely virtual devices, such as pseudo-terminals and virtual network interfaces. The security guarantee comes from the fact that these devices are explicitly created for each container and not shared.

Stateless devices, such as *random*, *null* and others. Sharing these devices among all containers and the host is safe because they are stateless.

User namespace-aware devices. If a device supports verifying process capabilities in the corresponding user namespace, then it is safe to expose such device to a container, because the specified limitations will be enforced. The current 3.14-rc2 upstream kernel does not have any physical devices supporting this feature, but they are expected to appear in the future.

All compared systems allow the system administrator to define a unique set of device nodes for each container and by default create only a small set of stateless and virtual devices. In Linux, creation of new device nodes within containers can be controlled by limiting access to the *CAP_SYS_MKNOD* Linux capability and by ensuring that all mountpoints inside containers have the *nodev* flag set.

The biggest difference of the Cells implementation is the addition of a "*device namespace*" that attempts to make the Linux input/output devices namespace-aware. Cells assumes the host to have a single set of input/output devices and multiplexes access to the physical host device via virtual devices created in each container. One virtual device at a time is allowed to access physical devices, based on whether an application from a given container is "on the foreground" (ie. visible on the screen) or not. Security-wise such exclusive-access solution is comparable with the "purely virtual" devices category mentioned above and can be considered safe.

As mentioned above, Linux device drivers controlling physical devices are currently not namespace-aware and thus cannot be securely used inside containers. Quite commonly these devices assume only one controlling master host and require privileges that are hard to grant for a unprivileged container securely (unless the device is used exclusively by a single container). In other words, namespace support inside the device drivers would require extensive modifications to the existing driver code base.

	Layer 3 bind filtering	**Layer 3 VNI**	**Layer 2 VNI**
traffic shaping and policing	no	yes	yes
separate routing and filtering tables	no	no	yes
used by	FreeBSD Jails, Linux-VServer	Solaris Zones, OpenVZ	Solaris Zones, OpenVZ, LxC, Cells

Table 3. Comparison of network isolation

4.5 IPC isolation

In order to achieve IPC isolation between containers, processes must be restricted to communicate via certain IPC primitives only within their own container. If the filesystem isolation is done correctly (see section 4.3), then filesystem-based IPC mechanisms (such as UNIX domain sockets and named pipes) are automatically isolated because the processes are not able to access filesystem paths outside of their own container. However, the isolation of the rest of the IPC objects (such as System V IPC objects and POSIX message queues) requires additional mechanisms. In Linux these IPC objects are isolated with the help of the *IPC namespaces* that allow the creation of a completely disjoint set of IPC objects. Linux-VServer, OpenVZ, LxC and Cells all spawn a new IPC namespace for each container in order to achieve the required isolation.

In addition to using IPC namespaces, Cells also has to implement namespace support for the Binder system since it is the primary IPC mechanism on the Android OS. The solution [11] includes having a separate Context Manager for each IPC namespace that is able to resolve Binder addresses only in that namespace and therefore provide isolation of Binder addresses between different containers.

Solaris Zones follow a different approach to isolate IPC objects that are not filesystem path-based. A zone ID is attached to each object based on the zone ID of the process that creates it, and processes are not able to access objects from other zones. An exception is made only for an administrator in the global zone that can access and manage all the objects. FreeBSD simply blocks SysV IPC object-related system calls if such calls are issued from within a jail. The *allow.sysvipc* option allows SysV IPC mechanisms for jailed processes but lacks any isolation between jails.

4.6 Network isolation

The main goal of network isolation is to prevent network-based attacks described in section 3. Moreover, in order to fulfill the server consolidation use case and to be able to support applications that might conflict for the same type of network resource (such as binding to the same network port), it also needs to provide a virtualized view of the network stack.

Network isolation methods differ in terms of the OSI layer of the TCP/IP stack where the isolation is implemented (see Table 3 for a comparison between these implementations). FreeBSD and Linux-VServer implement network isolation on the Layer 3 with the help of bind filtering. They restrict a *bind()* call made from within a container to a set of specified IP addresses and therefore processes are only allowed to send and receive packets to/from these addresses. The benefit of such approach is the small amount of code that needs to be modified in the network implementation and a minimal performance overhead. However, the downside is that a lot of the standard networking functionality is not accessible for a process inside a container such as obtaining an address from the Dynamic Host configuration Protocol (DHCP), acting as a DHCP server or the usage of routing tables.

Another approach, supported by Solaris Zones and OpenVZ, provides a Layer 3 virtualized network interface (VNI) for each container. Compared to bind filtering this implementation is more flexible since it allows the configuration of different traffic control settings, such as traffic shaping and policing, from within the container. The Layer 3 implementation provided by OpenVZ is called *venet*, while Solaris uses the term *shared-IP zone*.

The third approach includes providing a Layer 2 virtualized network interface for each container with a valid Link layer address. This gives containers the ability to use many features that are not supported by the previous two solutions, such as DHCP autoconfiguration, separate routing information, filtering rules and IP network multipathing. This approach can also support a broader set of network configurations. However, the primary downsides include a performance penalty and the inability to control the container networking setup from the host. The latter can be important for the server consolidation case if the host administrator needs to be in the control of the overall network configuration. OpenVZ, Solaris, LxC and Cells all support the creation of the Layer 2 virtualized interfaces. On Linux platforms this feature is called virtual Ethernet (*veth*). On Solaris a similar configuration is named *exclusive-IP zone*.

The Linux Layer 2 network isolation is based on the concept of a *network namespace* that allows the creation of a number of networking stacks that appear to be completely independent. The simplest networking configuration for a container running in a separate network namespace includes a pair of virtually linked Ethernet (*veth*) interfaces and assigning one of them to the target namespace while keeping the other one in the host namespace. After the virtual link is established, interfaces can be configured and brought up [6].

Linux provides multiple ways for connecting containers to the physical networks. One option is connecting the *veth* interface and the host physical interface by using a virtual network bridge device. Another option is to utilize routing tables to forward the traffic between virtual and physical interfaces. When a virtual bridge device is used, all container and host interfaces are attached to the same link layer bridge and thus receive all link layer traffic on the bridge. However, in the case of the route configuration, containers are not able to communicate with each other unless a network route is explicitly provided. Also in the latter

	rlimits	***cgroups***
scope	per process, inheritable	per process group, inheritable
managed resources	memory(limited), CPU(limited), filesystem, number of threads	memory, CPU, block I/O, devices, traffic controller
action when limit is reached	resource request denial and process termination	resource request denial, possibility to have a custom action
used by	Linux-VServer, Cells	OpenVZ, LxC, Linux-VServer, Cells

Table 4. Comparison of Linux resource management mechanisms

case, the container addresses are not visible for outsiders like in the bridged mode. Another way of providing network connectivity for containers is to use the *MACVLAN* interface [10] that allows each container to have its own separate MAC address. MACVLAN can be set to operate in a number of modes. In a PRIVATE mode containers cannot communicate with each other or the host making it the strictest isolation setup. The BRIDGE mode allows containers to communicate with one another, but not with the host. The last mode is the Virtual Ethernet Port Aggregator (*VEPA*) mode that by default isolates containers from one another, but leaves a possibility to have an upstream switch that can be configured to forward packets back to the corresponding interface. Currently LxC is the only solution that can support all the *macvlan* modes.

4.7 Resource limitation

A good virtualization solution needs to provide support for limiting the amount of primary physical resources allocated to each container in order to prevent containers from carrying out denial of service attacks described in section 3.

Since the 9.0 release FreeBSD utilizes Hierarchical Resource Limits (RCTL) to provide resource limitation for users, processes or jails [13]. RCTL supports defining an action in case a specified limit is reached: deny new resource allocation, log a warning, send a signal (for example SIGHUP or SIGKILL) to a process that exceeded the limit or to send a notification to the device state change daemon.

Solaris implements resource management for zones using a number of techniques that can be either applied to a whole zone or to a specific process inside a zone. Resource partitioning, called *resource pools*, allows defining a set of resources, such as a physical processor set, to be exclusively used by a zone. A dynamic resource pool allows to adjusting the pool allocations based on the system load. Resource capping is able to limit the amount of the physical memory used by a zone.

The traditional way of managing resources on BSD-derived systems is the *rlimits* mechanism that allows specifying soft and hard limits for system resources for each process. Cells and Linux-VServer utilize *rlimits* to do resource management for containers. However, the main problem of rlimits is that it does not allow specifying limits for a set of processes or to define an action when a limit is reached. Also the CPU and memory controls are very limited and do not allow specifying the relative share of CPU time, number of virtual pages resident in RAM or physical CPU or memory bank allocations.

In an attempt to address some of these limitations, OpenVZ and Linux-VServer have implemented custom resource management extensions, such as new limits for the maximum size of shared and anonymous memory or new CPU scheduler mechanisms. In addition both virtualization solutions added the possibility to specify resource limits per container.

Linux Control Groups (cgroups) [3] is a relatively new mechanism that aims to address the downsides of *rlimits*. It allows arranging a set of processes into hierarchical groups and performs resource management for the whole group. The CPU and memory controls provided by *cgroups* are rich, and in addition it is possible to implement a complex recovery management in case processes exceed their assigned limits. LxC, Linux-VServer, OpenVZ and Cells provide a way to use cgroups as a container resource management mechanism.

Table 4 presents a comparison of different aspects between *rlimits* and *cgroups*. A combined use of these mechanisms allows protecting the container from a set of DoS attacks directed towards the CPU, memory, disk I/O and filesystem (*rlimits* combined with *filesystem quotas*). However, the future direction is to aggregate all resource management to *cgroups*, and allow *rlimits* to be changed by a privileged user inside a container[5].

5 Related work

A number of previous studies have compared different aspects of the OS-level virtualization to other virtualization solutions. Padala et al. [31] analyze the performance of Xen vs. OpenVZ in the context of server consolidation. Chaudhary et al. [26], Regola et al. [35] and Xavier et al. [41] perform comparisons of different virtualization technologies for HPC. Yang et al. [42] study the impact of different virtualization technologies for the performance of the Hadoop framework [45].

The Capsicum sandboxing framework [38] introduced in FreeBSD 9 isolates processes from global kernel resources by disabling system calls which address resources via global namespaces. Instead, resources are accessed via capabilities which extend Unix file descriptors. Linux has a similar mechanism, called *seccomp* [18], that allows a process to restrict a set of systems calls that it can execute. Both Capsicum and *seccomp* require modifications to existing applications.

While there are OS-level virtualization solutions such as ICore [4] and Sandboxie [14] in existence for Microsoft Windows as add-on solutions, we have left

[5] documentation in source code of `http://lxr.linux.no/#linux+v3.13.5/kernel/sys.c#L1368`

them out this paper's scope due to their closed nature. Authors are not aware of any OS-level virtualization solutions for Mac OSX or iOS.

In addition to the OS-level virtualization solutions under comparison in this study, researchers have developed a number of other technologies. An attempt by Banga et al. [23] to do fine-grained resource management lead to creation of a new facility for resource management in server systems called *Resource Containers*. Zap [30] allows to group processes into *Process Domains* (PODs) that provide a virtualized view of the system and support for CR. An OS-level virtual machine architecture for Windows is proposed by Yu et al. [43]. A partial OS-level virtualization is provided by the PDS environment by Alpern et al. [21]. Wessel et al. [39] propose a solution for isolating user space instances on Android similar to the Cells/Cellrox. The solution by Wessel et al. has a special focus on security extensions, such as remote management, integrity protection and storage encryption.

6 Discussion and Conclusions

All compared systems implement core container separation features in terms of the memory, storage, network and process isolation. However, while the initial innovation around containers happened on FreeBSD and Solaris, the mainline Linux has caught up in terms of features and the flexibility of the implementation. Linux is likely to have a complete user space process environment virtualization in course of time. Given the scale of deployment of Linux and the maturity of its OS-level virtualization features, we focus on Linux in the rest of this section.

Table 5 summaries the state of the OS-level virtualization supported by the current upstream Linux kernel. The first row shows how each type of isolation discussed in section 4 can be achieved using the currently available techniques. The second row presents a number of gaps that are briefly described below.

Security namespaces. In order to reduce security exposure and adhere to the *principle of least privilege*, many OSs provide an integrated mandatory access control (MAC) mechanism. MACs can be used to strengthen the isolation between different containers and the host, as well as to enforce MAC policies for processes inside containers. The latter is especially important when the container has a full OS installation, because it usually comes with pre-configured MAC policies. Therefore, OS-level virtualization solutions should support the ability to use the common MAC mechanisms in the underlying host kernel to enforce independently defined (container-specific) MAC policies. However, currently none of the compared solutions fulfills this requirement. Linux kernel developers plan to address this limitation in the future by introducing a *security namespace* that would make LSMs container-aware.

IPC extensions. While IPC namespaces and filesystem isolation techniques cover most of the inter-process communication methods available on Linux, exceptions exist. For example *Transparent Inter-process Communication* (TIPC) [16] is not currently covered. TIPC is a network protocol that is designed for an inter-

<table>
<tr><th></th><th>separation of processes</th><th>filesystem isolation</th><th>IPC isolation</th><th>device isolation</th><th>network isolation</th><th>resource limitation</th></tr>
<tr><td rowspan="2">achieved by</td><td>pid ns</td><td colspan="3">mount ns, pivot_root</td><td>network ns</td><td>rlimits</td></tr>
<tr><td colspan="2">user ns</td><td>ipc ns</td><td>cgroups device controller, exclusive device usage</td><td>veth, macvlan</td><td>cgroups</td></tr>
<tr><td>open problems</td><td colspan="2">security ns</td><td>IPC extensions</td><td>device ns, (pseudo)random devices, hotplug support</td><td></td><td>uncompleted cgroups</td></tr>
</table>

Table 5. Summary of OS-level virtualization in upstream Linux kernel

cluster communication. Usage of such methods would break the IPC isolation borders between containers and if the given features are not needed, they should be disabled from the kernel configuration.

Device namespaces. As discussed in section 4.4, secure access to device drivers from within a container remains an open problem. One way to approach it would be to create a new namespace class (a *device namespace*) and group all devices to belong in their own device namespaces in hierarchical manner, following the generic namespace design pattern. Given this, only processes within the same device namespace would be allowed to access devices belonging in it. However, since the core of such functionality would resemble more access/resource control than a fully featured namespace, it was initially decided to implement the functionality as a separate *cgroups* device controller. The discussions defining the full notion of the device namespace and its functionality continue in the kernel community [5].

(Pseudo)random number generator devices. In section 4.4 we have stated that using stateless devices such as */dev/random* or */dev/urandom* are secure within containers due to their stateless nature. This means that even if two containers share the same device, they cannot predict or influence the output from another device node within another container. However, it is important to note that exposing blocking devices, such as */dev/random*, poses a DoS possibility. A malicious container can exhaust all available entropy and block the */dev/random* from being used in all other containers and the host, making it impossible to perform cryptographic operations requiring random input. If only non-blocking */dev/urandom* is exposed, there is a theoretical possibility that a malicious container can predict the random output for another container or a host. A complete solution would be to implement a separate random device per namespace or even introduce a namespace for (pseudo)random number generators.

Hotplug support. The desktop Linux relies heavily on the dynamic nature of the device nodes. Once the new devices are plugged in to the system, kernel generates an *uevent* structure notifying the user space about the presence of the new hardware. As briefly explained in section 4.4, *Uevent* is typically handled by the *udevd* daemon which configures the device for the system use. Traditionally it has also created the corresponding device node after the device setup. As far as containers are concerned, this setup is risky and complicated - containers should not be allowed to configure the hardware and/or have permissions for creating the new device nodes. As a result, the safe device hotplug for containers remains an open problem.

Uncompleted *cgroups*. As was mentioned in the section 4.7, the current goal of the upstream Linux is to integrated all features supported by *rlimits* into the *cgroups* resource management. However this has not been done yet and currently remains the work in progress.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable suggestions in order to improve the paper.

References

1. AppArmor project wiki. `http://wiki.apparmor.net/index.php/Main_Page`.
2. Cellrox project. `http://www.cellrox.com/`.
3. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.
4. iCore project page. `http://icoresoftware.com/`.
5. Linux Containers mailing list. `http://lists.linuxfoundation.org/pipermail/containers/2013-September/033466.html`.
6. Linux Network Namespaces. `http://www.opencloudblog.com/?p=42`.
7. Linux Programmer's Manual page on chroot(2) from 20.9.2010 (release 3.35).
8. Linux Programmer's Manual pages (release 3.35).
9. Linux-VServer project. `http://linux-vserver.org`.
10. LxC project. `http://linuxcontainers.org/`.
11. Namespace support for Android binder. `http://lwn.net/Articles/577957/`.
12. OpenVZ project. `http://openvz.org`.
13. RCTL. `https://wiki.freebsd.org/Hierarchical_Resource_Limits`.
14. Sandboxie project page. `http://www.sandboxie.com/`.
15. Smack project. `http://schaufler-ca.com/home`.
16. TIPC project. `http://tipc.sourceforge.net/`.
17. Biederman. Multiple Instances of the Global Linux Namespaces. In *Linux Symposium*, pages 101–112, 2006.
18. Corbet. Seccomp and sandboxing. `http://lwn.net/Articles/332974/`.
19. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, pages 483–490, 1981.
20. Edge. Another union filesystem approach. `https://lwn.net/Articles/403012/`.
21. Alpern et al. PDS: a virtual execution environment for software deployment. In *VEE*, pages 175–185, 2005.

22. Andrus et al. Cells: a virtual mobile smartphone architecture. In *ACM SOSP*, pages 173–187, 2011.
23. Banga et al. Resource containers: A new facility for resource management in server systems. In *OSDI*, pages 45–58, 1999.
24. Barham et al. Xen and the art of virtualization. *ACM SIGOPS OSR*, pages 164–177, 2003.
25. Bhattiprolu et al. Virtual servers and checkpoint/restart in mainstream Linux. *ACM SIGOPS OSR*, pages 104–113, 2008.
26. Chaudhary et al. A comparison of virtualization technologies for HPC. In *AINA*, pages 861–868, 2008.
27. Kamp et al. Jails: Confining the omnipotent root. In *SANE*, page 116, 2000.
28. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.
29. Mirkin et al. Containers checkpointing and live migration. In *Linux Symposium*, pages 85–92, 2008.
30. Osman et al. The design and implementation of Zap: A system for migrating computing environments. *ACM SIGOPS OSR*, pages 361–376, 2002.
31. Padala et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.
32. Pike et al. Plan 9 from Bell Labs. In *UKUUG*, pages 1–9, 1990.
33. Pike et al. The Use of Name Spaces in Plan 9. In *5th workshop on ACM SIGOPS European workshop*, pages 1–5, 1992.
34. Price et al. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA*, pages 241–254, 2004.
35. Regola et al. Recommendations for virtualization technologies in high performance computing. In *IEEE CloudCom*, pages 409–416, 2010.
36. Shim et al. Bring Your Own Device (BYOD): Current Status, Issues, and Future Directions. 2013.
37. Smalley et al. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.
38. Watson et al. Capsicum: Practical Capabilities for UNIX. In *USENIX*, pages 29–46, 2010.
39. Wessel et al. Improving Mobile Device Security with Operating System-Level Virtualization. In *Security and Privacy Protection in Information Processing Systems*, pages 148–161. 2013.
40. Wright et al. Linux security module framework. In *Linux Symposium*, pages 604–617, 2002.
41. Xavier et al. Performance evaluation of container-based virtualization for high performance computing environments. In *PDP*, pages 233–240, 2013.
42. Yang et al. Impacts of Virtualization Technologies on Hadoop. In *ISDEA*, pages 846–849, 2013.
43. Yu et al. A feather-weight virtual machine for windows applications. In *VEE*, pages 24–34, 2006.
44. The Open Group. *The Single UNIX Specification: Authorized Guide to Version 4*. 2010. `http://www.unix.org/version4/theguide.html`.
45. Kizza. Virtualization Infrastructure and Related Security Issues. In *Guide to Computer Network Security*, pages 447–464. 2013.
46. Kolyshkin. Virtualization in Linux. *White paper, OpenVZ*, 2006.
47. Rosenblum. VMwares Virtual Platform. In *Hot Chips*, pages 185–196, 1999.