Dynamic Isolated Domains

Thomas Nyman

Helsinki October 6, 2014

MSc thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty		Laitos — Institution — Department
Faculty of Science		Department of Computer Science
Tekijä — Författare — Author		
Thomas Nyman		
Työn nimi — Arbetets titel — Title		
Dynamic Isolated Domains		
Oppiaine — Läroämne — Subject		
Computer Science		
Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	October 6, 2014	68 pages

Tiivistelmä — Referat — Abstract

Operating System-level Virtualization is virtualization technology based on running multiple isolated userspace instances, commonly referred to as *containers*, on top of a single operating system kernel. The fundamental difference compared to traditional virtualization is that the targets of virtualization in OS-level virtualization are kernel resources, not hardware. OS-level virtualization is used to implement *Bring Your Own Device* (BYOD) policies on contemporary mobile platforms. Current commercial BYOD solutions, however, don't allow for applications to be containerized dynamically upon user request. The ability to do so would greatly improve the flexibility and usability of such schemes.

In this work we study if existing OS-level virtualization features in the Linux kernel can meet the needs of use cases reliant on such dynamic isolation. We present the design and implementation of a prototype which allows applications in *dynamic isolated domains* to be migrated from one device to another. Our design fits together with security features in the Linux kernel, allowing the security policy influenced by user decisions to be migrated along with the application. The deployability of the design is improved by basing the solution on functionality already available in the mainline Linux kernel. Our evaluation shows that the OS-level virtualization features in the Linux kernel indeed allow applications to be isolated in a dynamic fashion, although known gaps in the compartmentalization of kernel resources require trade-offs between the security and interoperability to be made in the design of such containers.

ACM Computing Classification System (CCS):
K.6.5 [Management of Computing and Information Systems][Security and Protection]

Avainsanat — Nyckelord — Keywords
mandatory access control, isolation, Linux, migration, namespaces, security, policy, virtualization

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Acknowledgments

First and foremost, I wish to express my sincere appreciation to my advisors, Prof. Asokan, and Elena Reshetova for their guidance and advice throughout the course of this work. I would like to thank Dr. Sini Ruohomaa for her invaluable support and wish her further success in her future endeavors.

I also would like to extend my gratitude toward the faculty at the Department of Computer Science at the University of Helsinki. I have thoroughly enjoyed my time as a student at the university. They are too numerous to name here, but I would like to single out senior lecturer Arto Wikla. His teaching style and enthusiasm made a strong impression on me and I carry with me positive memories of his classes.

Last but not least I wish to thank my family — my Parents, for their love and encouragement, and my partner Kajsa, for her patience and caring. This would not have been possible without you.

This work was supported by the Intel Collaborative Research Institute for Secure Computing.

Contents

1 Introduction

In the 1960s, before multitasking operating systems, virtualization was developed for mainframe computers as a method to logically divide the mainframe resources among different applications. Since then, virtualization has become commonplace both on desktop, and warehouse scale computers. Virtualization has been instrumental in the emergence of the cloud-computing paradigm.

A relatively recent research direction is *Operating System-level Virtualization*. OS-level virtualization is based on running multiple isolated userspace instances, commonly referred to as *containers*, on a single operating system kernel. The fundamental difference compared to traditional virtualization is that the targets of virtualization in OS-level virtualization are kernel resources, not hardware. As a consequence, OS-level virtualization exhibits less overhead compared to traditional virtualization technologies, as guests can share a common instance of the operating system kernel. The challenge with OS-level virtualization lies in compartmentalizing kernel functionality in a way which ensures adequate isolation between, potentially untrusted, containers.

The principal motivations behind the development OS-level virtualization technology have been use cases in high-performance computing, shared hosting and high-availability environments. With this in mind, it is no surprise that OS-level virtualization technology is primarily found on Unix-like operating systems. However, the technological advances in mobile computing combined with the emergence of use cases such as *Bring Your Own Device* (BYOD) policies [MVH12], have lead to the application of OS-level virtualization technology also on mobile devices [ADH$^+$11, TMO$^+$12]. With regards to BYOD, OS-level virtualization can provide strong isolation with less overhead compared to traditional virtualization technologies. This is crucial on energy-constrained devices where increased CPU usage translates into reduced battery lifetime.

Containerization is, however, no panacea. Current commercial BYOD solutions are merely a stop-gap measure IT-departments can use to protect corporate assets on employee devices against risks that follow from the different sense of ownership users may have about personal devices, as opposed corporate-issued ones. This constitutes merely the low-hanging fruit as far as threats go. Compromise of the host kernel makes any protection containers may provide moot. Still, this does not mean that OS-level virtualization would not have a use as one layer in a more comprehensive solution to mobile security, possibly combined with other platform security measures.

Other challenges with OS-level virtualization on mobile devices lie in the area of usability. While it is perfectly natural for containers to appear as completely separate entities in a server setting, the users of modern mobile devices have become accustomed to "seamless" interaction between application on their device as well easy access to new apps. The compartmentalization provided by containerization needs to flexible enough to accommodate different interaction patterns and novel use cases. This leads to an inevitable trade-off between security and ease-of-use. To keep up with the demand of including new applications in existing workflows, applications need to be containerized dynamically upon user request.

As far as target platforms go, Linux-based operating systems constitute an appealing target for research in this particular area. Not only are Linux-based mobile platforms prominent, recent versions of the mainline Linux kernel also include a rich set of OS-level virtualization features. These features are modular by design, allowing them to be applied for process containment in a variety of ways and combinations. This brings a high-level of flexibility, but the sometimes the subtle interaction between these features makes applying them in a secure manner non-trivial. The purpose of this thesis is to study the use of current container-based isolation mechanisms in the Linux kernel, in particular their applicability to novel use cases in mobile computing. In particular, we make the following contributions:

- We describe the use of existing Linux kernel features for **dynamic application containerization**, and evaluate out setup with regards to the threat model described in Section 4.2.
- We describe the **design and implementation of a prototype system for application migration**, with the aim of preserving established security domains for applications between devices.

This thesis is organized as follows: Chapter 2 provides a study of OS-level virtualization features available in the mainline Linux kernel, as well as a historical overview of the development of process isolation mechanisms leading up to current container-based isolation. Chapter 3 describes the design goals behind the prototype, while Chapter 4 introduces the system and threat models we use as a basis for the design. Chapter 5 presents the design and implementation of the prototype itself. The methodology and results of the evaluation of the prototype is described in Chapter 6. Related work is discussed in Chapter 7. Finally, Section 8 summarizes our contributions and discusses open issues as well as directions for future work.

2 Background

In this section we give a brief overview of the origins of virtualization (Section 2.1), its use cases, and the development of operating system-level virtualization technology (Sections 2.3, 2.4). We also describe the current primitives for container-based virtualization available in the Linux kernel (Section 2.5) as well security features that are relevant to common use cases and our prototype (Sections 2.6, 2.7).

2.1 Virtualization

Conventional *Virtual Machines* (VMs) are efficient, isolated duplicates of the real machine they run on. The real machine or environment is known as the *host*, while the virtual machines or environments are referred to as *guests*. Virtual machines are created and run by a *Virtual Machine Monitor* (VMM). The VMM has three essential properties [PG74]:

Equivalence

The VMM provides an environment which is essentially identical to the original machine.

Efficiency

Programs run in the VM environment exhibit at worst minor performance penalties.

Safety

The VMM is in complete control of system resources.

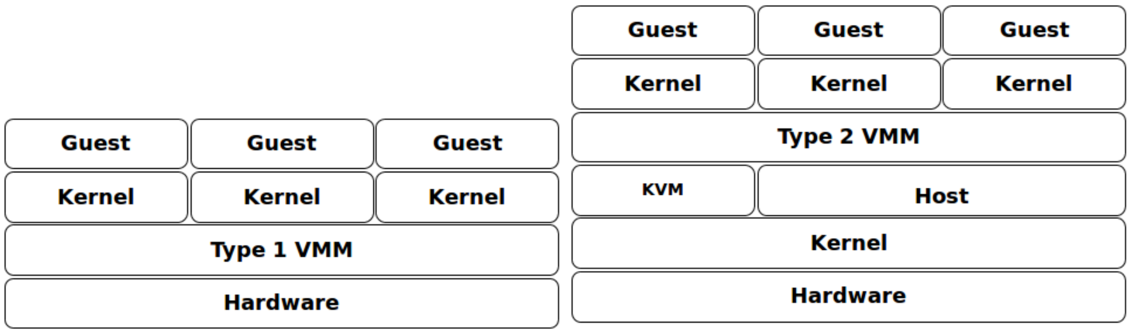
VMMs are traditionally classified into two types:

Type I (native, or bare metal)

Type I VMMS are run directly on the host hardware, mediating access between the hardware and a number of guests running on top of the VMM. The VMM may be implemented in either hardware or firmware. A schematic of a Type I VMM architecture is shown in Figure 1a.

Type II (hosted, or host-based)

Type II VMMs are software VMMs which run within a conventional operating system environment. Guest operating systems are run on top of the VMM. A schematic view of a Type II VMM architecture is shown in Figure 1b.



(a) Bare metal virtualization

(b) Host-based virtualization

Figure 1: VMM architectures

In both cases, the system software of each guest is a full-blown operating system in its own right, with distinct kernels and *userlands*[1]. The guest operating systems that run within each virtual machine either execute machine I/O instructions that the VMM emulates, or the operating system is modified to make system calls directly to the VMM. The latter approach is called *paravirtualization*.

The term *virtual machine* most likely originates from the IBM M44/44X experimental paging system project [Var97], which in the mid-60s emulated multiple virtual IBM 7044 machines on a single 7044 mainframe. The M44/44X was one of the first instances of *partial virtualization*, where the virtual machine emulates multiple instances of a significant portion of an underlying hardware environment. A key form of partial virtualization is *address space virtualization*, in which each virtual machine consists of an independent address space.

The experimentation with partial virtualization eventually led to the creation of the IBM CP-40 [Var97], the first system capable of *full virtualization*. In full virtualization, the underlying virtual machine emulates hardware to such a degree that an unmodified guest operating system designed for the same instruction set may be run

[1]Userland software refers to any software part of, or running on top of the operating system and not part of the operating system kernel.

in isolation within the virtual machine. The successor to CP-40, CP-67 was used in IBM's CP-67/CMS operating system for the System/360-67 mainframe. CP-67/CMS became the first widely available virtual machine architecture. CP-67 was the *Control Program* portion of the *Control Program / Cambridge Monitoring System* (CP/CMS) time-sharing operating system. In the CP/CMS architecture, the CP creates the virtual machine environment. Each virtual machine run a copy of CMS, a lightweight single-user operating system. By emulating a full, stand-alone computer for each user, CP/CMS could run any S/360 software in a time-sharing environment, not just applications specifically designed for time-sharing. In addition, CP/CMS achieved unprecedented time-sharing performance compared to contemporary multi-tasking operating systems. The CP/CMS virtual machine concept was also an important step in operating system design, as it greatly improved system reliability and security.

The CP-67/CMS system was later reimplemented for the System/370 mainframe. The successor, CP-370/CMS was never released as such, but became the foundation of IBM's *Virtual Machine Facility/370* (VM/370) operating system [Cre81], announced in 1972. The S/370 mainframe for use with VM/370 was the first system to provide *hardware-assisted virtualization*, i.e. architectural support that facilitates building a virtual machine monitor. In particular the processor architecture allowed the VMM to set the processor into *partial execution* mode [Olb78]. In this unprivileged mode, privileged instructions by the guest operating system are not executed directly, but generate a trap instruction which results in a context switch to the VMM. This greatly simplifies the implementation of the VMM and improves performance, as the VMM only needs emulate the traps to allow the correct execution of the guest operating system. This technique later became known as *direct execution* [BDR02].

In CP-67, certain model-dependent and diagnostic instructions were not virtualized. The diagnostic instruction `DIAG` was as used as a signal between a CMS instance and the CP. This is one of the earliest examples of a paravirtualization interface, which allowed the CMS to request the CP to perform filesystem operations and request other VM services directly, avoiding the overhead of of full emulation. In VM/370, the term *hypervisor* was used to refer to the virtual `DIAG` instruction handler[2]. The term has later become synonymous with VMM.

With the commoditization of microcomputers, in particular the emergence of the PC platform, rapidly decreasing hardware costs, and more sophisticated operating systems, mainframe virtualization architectures based on direct execution began to lose their appeal. The x86 architecture initially lacked hardware support for virtualization. Hence research on virtualization for the x86 architecture predating the virtualization extensions to the x86 instruction set focused on software-based techniques. The first commercial virtualization solutions for x86 were aimed at workstation computers, allowing a guest operating system to be run on top of a VMM running as a process on the host operating system. VMware workstation and Virtual PC, the best known virtualization solutions for x86 at the time, utilized *dynamic binary translation* [AA06] to achieve full virtualization support on the x86 architecture. Without hardware support for trap-and-emulate style direct execution the VMM would, whenever possible, run user-mode and virtual real mode code directly. When direct execution was not possible, as in the case of kernel-mode or real mode code, the VMM would resort to rewriting guest instructions to allow them to be executed in non-privileged mode. Binary translation also allows cross-platform virtualization, where the VMM is used provide compatibility between different processor architectures. This makes it possible to run entire operating systems written for a particular processor architecture to be run as a guest on top of a host operating

[2]http://www.zdnet.com/blog/btl/etymology-of-hypervisor-surfaces/1710

system written for another architecture. However, binary translation incurs considerable overhead, as no instructions may be executed directly prior to the binary translation.

Software-based virtualization enabled the widespread use of commodity hardware for contemporary use-cases of virtualization technology:

Alternate operating systems

Virtual machines allow an alternate operating system to be run as a guest operating system, without modifying the host operating system. Reasons for this include use of applications which are not supported on the host operating system, or evaluating the guest operating system without altering the host operating system setup.

Server consolidation

Multiple virtual machines can be run on one physical server, increasing the hardware utilization and energy efficiency in hosting environments. Virtual machines can also be provisioned as needed without the need to purchase additional hardware up-front. They can also be relocated from one physical host to another.

Fault containment

Virtual machines can be used prevent the propagation of a software failure. This could include studying malware or misbehaving software in a safe environment. When done, the VM can simply be discarded. Alternatively, multiple copies of the VM might exist, allowing a malfunctioning system to be replaced with a clean copy in high-availability environments.

Resource management

Virtual machines can also be used as a unit for resource allocation. VMs have enabled Internet hosting providers to provide customers with dedicated VM instances. Such *Virtual Private Servers* (VPSs) are for many purposes equivalent to a dedicated physical server, including allowing the customer superuser privileges on the guest operating system, but much more cost-effective, as they share the physical hardware with other VPS instances. The customer may be allocated a certain amount of resources, such as storage space and CPU cycles for their VPS, to use as they please, or they might billed according to the amount of resources used by their VM.

In the context of cloud computing, virtualization enables cloud computing platforms to dynamically adapt to workload changes to match the current demand as closely as possible [MH12]. This *elasticity* allows the cloud service provider to avoid over or under-provisioning, which in the end reduces their expenses while still allowing them to provide a certain level of service. A challenge in providing elasticity is the delay between the provision of a VM, until it is ready to use.

In the mid 2000s, Intel and AMD, the major x86 CPU manufacturers independently launched the AMD-V and Intel VT-x virtualization extensions for the x86 architecture. The first generations of chipsets supporting the virtualization extensions allowed trap-and-emulate style direct execution. However, with the advances in software-based virtualization, hardware-assisted virtualization on the x86 architecture alone didn't offer any significant improvement in performance. The second generation of chipsets added support for MMU virtualization, increasing the performance of virtualized system memory. Linux supports the x86 virtualization extensions through the *Kernel-based Virtual Machine*[3](KVM).

A fairly recent research direction with the aim of providing more lightweight virtualization compared to hardware or host-based virtualization is *operating system-*

[3]`http://www.linux-kvm.org`

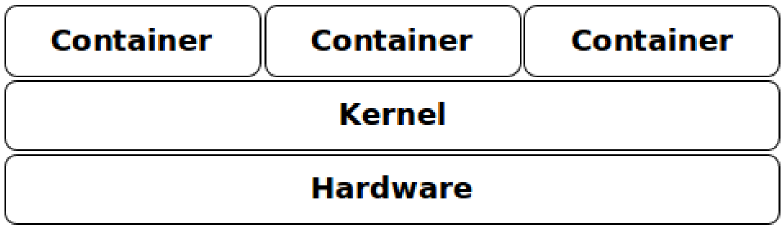


Figure 2: Container-based virtualization

level virtualization. OS-level virtualization revolves around running multiple isolated userspace instances, commonly referred to as *containers* on a single operating system kernel. A schematic of a container architecture is shown in Figure 2. A fundamental difference compared to the virtualization approaches mentioned above is that the object of virtualization is not hardware, but global kernel resources. With the rise of Linux-based mobile platforms, operating system-level virtualization mechanisms in the Linux kernel have received attention with regards to virtualization on mobile devices. The smaller resource footprint compared to traditional virtualization approaches make these more suitable for use on resource constrained devices.

2.2 The Dilemma of the Omnipotent Root

Traditional Unix systems make a distinction between two types of processes; *privileged* processes, and *unprivileged* processes. Privileged processes are able to bypass *all* regular access control checks made by the system kernel, whereas unprivileged processes are subject to access control checks based on the process' credentials.

Each process has certain identifiers associated with it; a *real User Identifier* (UID) and *Group Identifier* (GID), an *effective* UID and GID, a saved *Set-User-ID* (SUID) and *Set-Group-ID* (SGID), and possibly a number of supplementary GIDs. The effective UID and GID together with the supplementary group list usually act as a process' credentials. The real UID identifies the user who launched the process, and is considered the processes' owner. The real GID is determined by the primary group of that user. The saved SUID and SGID are the UID and GID respectively the process had when it began its execution (at the point of the last `exec()` call). Usually the real, effective and saved UIDs are the UID of the user who executed the program.

Traditionally, processes are considered privileged by the virtue of having the effective UID `0`. For convenience, this UID is by convention allocated to a superuser account, usually named *root*. The ability to bypass any access control checks is due to an effective UID of `0` being handled as a special case by access control checks in the kernel. These checks serve two purposes; firstly, they are used to uphold *Discretionary Access Control* (DAC) policies, which are used to protect filesystem objects (see Section 2.7), and second, they protect functionality offered by the kernel which, if unchecked, could be used to undermine the underlying security measures provided by the kernel, e.g. memory protection, hardware access control etc.

Unfortunately concentrating the privilege to a single point of contact, the superuser, makes adhering to the *principle of least privilege* [SS75] difficult. Delegation of privileges in traditional Unix relies on setting the *setuid* access control flag on program binaries. This has the effect of setting the effective UID to the *owner of the file* when the binary is executed, as opposed to the UID of the *owner of the process*. When set on root-owned binaries, the spawned process will be privileged with regards to all access control, regardless of the intended functionality of the application. This makes setuid binaries an attractive target for intruders looking to compromise the security of a system. Programming errors in setuid binaries, in particular ones that could be exploited for arbitrary code execution, are especially dangerous. The Unix C *Application Programming Interface* (API) provides means for privileged processes to drop its privileges, either temporary or permanently[4]. This occurs by toggling the effective UID between the processes real UID or saved SUID. However, in order to be effective, these mechanisms require the program to be structured in a way that minimize the portions of the program run with privileges, and ideally allow it to drop its privileges permanently as soon as possible. Even then, errors within the privileged portions remain as dangerous as before, leaving much to be desired with regards to avoiding over-privilege and the ability to compartmentalize functionality available to privileged processes.

Modern Unix-like operating systems try to address this disparity between privileged and unprivileged processes in various ways. Linux divides the privileges traditionally associated with superuser into distinct units, known as *capabilities*[5], which can be independently adjusted programmatically on a per-thread basis. Capabilities can also be set via file system attributes. Linux capabilities are based on a proposal part of the 1003.1e draft for the *Portable Operating System Interface of Unix* (POSIX) standard. Even though the draft was eventually withdrawn, the capability model employed by Linux is sometime referred to as *POSIX capabilities.*

Capabilities[6]allow for more fine-grained control of privileges, but this alone is not enough to meet the requirements of use cases such as server consolidation, where it is desirable to delegate some, but not all, administrative functions to less trusted parties, and simultaneously impose system-wide mandatory policies to system resources. The Unix access control model makes compartmentalization of this kind difficult, as security domain the superuser has privilege over encompasses the entire system.

[4]`http://pubs.opengroup.org/onlinepubs/007904975/functions/setuid.html`

[5]`https://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt`

[6]We note that a more common meaning for the term *capability* comes from *capability-based security* [Lev84], where it is used to refer to transferable, unforgeable tokens use to prove proper authorization access an object.

Fundamentally operating system-level virtualization is about compartmentalizing functionality available to processes. In Unix-like operating systems, mechanisms such as `chroot()` have for a long time been used to provide a modest level of compartmentalization for server applications. The problems with `chroot()` jails are well understood. We describe some of these in the following section, and more sophisticated approaches are discussed in the subsequent sections.

2.3 Chroot jails

The `chroot()`system call allows one to change the root directory of a running process and its children, limiting their visibility of the global filesystem hierarchy to a single subtree. The modified environment achieved via the `chroot()` system call is commonly called a *chroot jail.*

Chroot jails were never intended as a security mechanism, but as a development tool for building software in a confined tree, separated from the main directory structure [7]. Even so, the chroot mechanism has been used as a security measure to limit filesystem access. The classic example is the *File Transfer Protocol* (FTP) daemon, which is largely dependent on the security provided by a chroot jail to secure anonymous FTP access [Ker10, p.367].

While the chroot mechanism does provide a modest level of compartmentalization, it has serious shortcomings in terms of functionality and effectiveness. There are

[7]According to Dr. Marshall Kirk McKusick, (as reported by Kamp and Watson [KW00]): "According to the SCCS logs, the chroot call was added by Bill Joy on March 18, 1982 approximately 1.5 years before 4.2BSD was released. That was well before we had ftp servers of any sort (ftp did not show up in the source tree until January 1983). My best guess as to its purpose was to allow Bill to chroot into the /4.2BSD build directory and build a system using only the files, include files, etc contained in that tree. That was the only use of chroot that I remember from the early days."

several possible routes for unprivileged processes to break out of a chroot jail [Ker10, p.368]:

- The `chroot()` call does not change the working directory of the calling process. Thus, the call to `chroot()` is typically preceded or followed by a call to `chdir()`. If the directory change is omitted, the working directory of the process might reside outside the new root directory tree. If this is the case, the calling process, or child processes inheriting the parent's working directory, can use relative pathnames to access directories outside the chroot jail.

- If a process obtains an open file descriptor for a directory outside the chroot jail (by, for instance, opening the directory before the `chroot()` call or by receiving the file descriptor from another process outside the chroot jail via an Unix domain socket), then calling `fchdir()` with the open file descriptor will result in a current working directory outside the chroot jail.

By design, a privileged process can escape a chroot jail by setting its working directory to the altered root directory and change the root directory again by calling `chroot()` for a subdirectory [8]. Thus, the process has obtained a current working directory outside of the chroot jail. Now it can traverse the directory tree up to the original root directory.

Some BSD derivatives prevent this by changing the current directory to the new root directory upon a `chroot()` call if the process is already running with an altered root directory[9]. This prevents the current directory from being further up the directory tree than the new root directory.

[8]This method is documented in the Linux Programmer's Manual page on *chroot(2)* from September 20, 2010 (release 3.75).

[9]This behavior is documented in the OpenBSD Programmer's Manual page on *chroot(2)* from June 17, 2013 (release 5.4).

2.4 FreeBSD jails

The shortcomings of Unix discretionary access control facilities and `chroot()` in providing adequate compartmentalization of different security domains in shared hosting environments prompted the development of the *jail* [KW00] facility in the FreeBSD operating system. While the compartmentalization provided by `chroot()` is limited to filesystem visibility, the FreeBSD jail mechanism extends the compartmentalization to include processes and networking. Jails can be created by privileged processes by calling the `jail()` system call, but once a process has entered a jail, it can never leave.

Processes in a jail have several restrictions placed on them. Filesystem access is limited to subtree visible in the jail. The chroot mechanism is used to provide filesystem compartmentalization. It has, however been augmented to refuse backwards traversal across the first level chroot directory and disallow open file descriptors to directories at the point of the `chroot()` call.

Processes inside a jail cannot see or interact with processes outside the jail. Process visibility in FreeBSD is provided through the `procfs` file system and `sysctl` mechanism. Both only divulge information on processes in the same jail to a jailed process. To prevent *Inter-Process Communication* (IPC) across jail boundaries, access to System V IPC primitives is inhibited. This restriction may be lifted on a per-jail basis[10], but as System V primitives in FreeBSD share a single namespace across the host and jail environments, doing so will not only enable IPC within a jail, but also across jail boundaries.

Each jail is assigned a set of *Internet Protocol* (IP) addresses. Processes in the jail are unable to bind a socket with other IP addresses apart from the ones assigned to the jail. Attempts to bind using wildcard addresses silently use the jailed addresses instead. Alternatively binding to IPv4 or IPv6 accesses can be hindered altogether on a jail-by-jail basis. A jail can also be created with a virtual network stack of its own. This allows a jail modify its own (virtual) network configuration, interfaces, protocol addresses, routing tables etc. Typically sockets within a jail are restricted to IPv4, IPv6 and local (Unix) sockets. Access to protocol stacks without explicit support for jails, including raw, divert sockets used by the kernel packet diversion mechanisms, and routing sockets can be enabled via a configuration parameter.

The capabilities of privileged processes running inside a jail are severely restricted. Privileged processes are not allowed to modify kernel runtime parameters, such as `sysctl` settings, load kernel modules or create device nodes. Mounting and unmounting filesystems is prohibited by default, although the ability to mount certain "jail-friendly" filesystem types can be granted on a per-jail basis. File flags related to BSD `securelevel` security profiles cannot be modified.

2.5 Resource namespaces

A kernel *resource namespace* [Ker13a] is an abstraction around a global system resource. Fundamentally, a namespace is a container for a set of identifiers. Identical identifiers may appear in multiple namespaces, but may refer to different resources in each namespace. In effect, processes within a namespace appear to access an isolated instance of the resource. Typically, the existence of such isolation is invisible to the processes within the namespace.

The principal application for resource namespaces are containers utilizing lightweight operating system-level virtualization. Common use cases for containers are VPS environments. When applied to a VPS environment, multiple distinct user-space software stacks are run in distinct containers on top of a shared operating system

[10]Jail parameters are documented in the FreeBSD System Manager's Manual page on *jail(8)* from October 12, 2013 (FreeBSD release 9.3)

kernel. This can be attractive in high-performance computing, as lightweight OS-level virtualization can provide less overhead and improved performance compared to traditional bare metal or host-based virtualization techniques.

Another use-case for resource namespaces is the implementation of process *Checkpoint and Restart* (CR) functionality [Bie06]. In CR, the objective is to store the execution state of a running process to a *process image* and restore it at a later point in time, continuing the execution of the process from the state stored in the image. Restoration can occur on another system than the one the process originated from. This is called *live migration*, and can be useful for load-balancing or in high-availability environments.

The challenge in CR lies not so much in storing the process state, but in restoring the process image, as a process might rely on possessing certain global resources, such as process IDs, IPC identifiers, filesystem paths etc. When a process is restored there is no guarantee that the process can reuse the same global identifiers, as they might be in use by other processes. The two main approaches to address this issue have been to either make sure global identifiers are unique across all machines in the computing cluster process migration might occur on, or to provide means of allowing the same set of identifiers to be repeated on the same system. A major challenge with the former approach is to allow the system to scale. Resource namespaces, in turn, make the latter possible.

Resource namespaces can also be used as an isolation measure between processes and system resources. While not strictly speaking a security mechanism, namespace isolation and access control are nevertheless related. They both prevent processes from having (unrestricted) access to system resources. Whereas conventional access control is based on an explicit *authorization* and is visible to process being controlled, namespace isolation prevents access by making only those resources that a process is allowed to access visible in its environment. However, as we shall see, namespace isolation is typically more coarse-grained than the corresponding access control measures. On the other hand, the isolation provided by namespaces has the advantage of being transparent to the process confined within the namespace.

Our primary interest in this thesis is the utilization of the Linux namespaces for resource isolation in combination with security features provided by the Linux kernel. While our use case involves the migration of applications, we limit ourselves to so called *cold migration*, i.e. the migrated application is required to have terminated before the migration occurs. We consider the problem of live migration orthogonal to our efforts; as long as the container setup provides an environment sufficiently independent from the host environment, support for a live migration system which serializes the state of a running application to filesystem objects can easily be incorporated as an add-on.

Current versions of the Linux kernel implement six different types of resource namespaces. Linux namespaces are instantiated when a process is created via the `clone()`[11] system call, or a process disassociates itself from its parent's namespace via the `unshare()` system call. The namespace to be unshared is in both cases identified via a bit flag defined in *bits/sched.h*. The creation of namespaces, with the exception of user namespaces, is a privileged operation.

Existing namespaces may be joined via the `setns()` system call. Linux namespaces are identified by file descriptors visible via the *proc filesystem* (procfs). In Unix-like operating systems this special filesystem provides information about existing processes represented as a file system hierarchy. It is typically mapped to a mount point at `/proc`. Each running process is represented by a directory under `/proc`, `/proc/<pid>`, where `<pid>` is the processes' numerical id. Each namespace supported by the kernel has an entry under the `/proc/<pid>/ns` directory. These

[11]Essentially `clone()` is a generalized version of the traditional Unix `fork()` system call, the primary, and traditionally only means of process creation.

Namespace	`clone()` flag	Kernel version
Mount namespaces	`CLONE_NEWNS`	2.4.19
UTS namespaces	`CLONE_NEWUTS`	2.6.19
IPC namespaces	`CLONE_NEWIPC`	2.6.19
PID namespaces	`CLONE_NEWPID`	2.6.24
Network namespaces	`CLONE_NEWNET`	2.6.24 — 2.6.29
User namespaces	`CLONE_NEWUSER`	2.6.23 — 2.6.29

Table 1: Linux namespaces

entries behave like file descriptors. The inode number of the file descriptor are unique to the namespace they represent. Hence, the inode number can be used to determine if two processes coexist in the same namespace.

The different namespaces and corresponding `clone()` flags are show in Table 1. The kernel version indicates the version of the Linux kernel in which the corresponding namespaces were introduced. *Mount namespaces* compartmentalize the visibility of mounted file systems. *UTS namespaces* allow for multiple host names and *Network Information Service* (NIS) domain names to be used on a single host. *IPC namespaces* control the visibility of IPC primitives not represented by file system objects. *PID namespaces* interact with procfs to cordon off visibility to process hierarchies, as well as allowing the same PIDs to be possessed by several processes (in different namespaces) simultaneously. Network namespaces compartmentalize access to networking primitives such as network interfaces, IP addresses and routing tables as well as port numbers. User namespaces compartmentalize UIDs and GIDs.

2.5.1 Mount namespaces

Mount namespaces [Ker13a] were the first namespaces to be implemented in the Linux kernel. This accounts for the, rather undescriptive, `NEWNS` (short for "new namespace") identifier assigned to namespace type constant. The design of Linux mount namespaces was influenced by the *Plan 9 from Bell Labs* operating system [Bie06].

The central design goals behind the Plan 9 [PPTT90] operating system was to integrate graphics and ubiquitous networking into a coherent, Unix-like framework. Similarly to traditional Unix systems, access to system services are provided through a single filesystem interface. In fact, Many facilities that under Unix are accessed through various ad-hoc interfaces like BSD sockets, *fcntl(2)*, and *ioctl(2)* are in Plan 9 accessed through ordinary read and write operations on special files analogous to Unix device files. Most system services, including the windowing system $8\frac{1}{2}$, are *file servers*, which provide special files or a directory tree representing resources they provide access to. As all mounted file servers export the same filesystem interface to users and client programs, access to each service looks the same regardless of the implementation behind them. Some might correspond to local filesystems, some to remote filesystems accessed over a network, some to instances of system servers running in userspace (like the windowing system), and some to kernel interfaces.

Plan 9 introduced the notion of *private namespaces* [PPT+92]. Every process can have its own view of the system's services by creating its own tree of file-server mounts. This allows for, for instance, `/dev/cons` always to refer to the user's terminal device and `/bin/date` to the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing `date`.

In a similar manner, Linux mount namespaces isolate the set of filesystem mounts visible to a group of processes. The `mount()` and `umount()` system calls only affect the mount namespace associated with the calling process. In contrast to Plan 9 namespaces, which cannot be joined by other means than direct inheritance, Linux provides the `setns()` system call, which can be used to join existing namespaces.

In Linux, mount namespaces interact interestingly with *bind mounts*, another filesystem feature influenced by Plan 9[12]. Bind mounts act as a sort of symbolic link at the filesystem level. In Linux they are implemented entirely within the *virtual filesystem* (VFS) layer, making them independent of any particular low level filesystem. Bind mounts allows to remount parts of the filesystem hierarchy at different mount points, making them visible at multiple points in the filesystem hierarchy simultaneously. When a mount operation occurs, it is possible to mark the mount and its submounts as *shared* or *slave* mounts. If a mount is marked shared, mounts and unmounts within the subtree propagate to any bind-mounted instances of the mount and vice versa. A slave bind mount receives propagated mounts and unmounts from its master (the original mount), but mounts within the slave are not propagated back. The propagation may occur across mount namespace boundaries, enabling schemes in which e.g. hot-pluggable mass storage devices may be made visible within unprivileged containers (with their own root filesystem) by mounting them within a shared mount point with the host.

2.5.2 UTS namespaces

The name of UTS namespaces [Ker13b] derive from the name of the structure passed to the `uname()` system call; `struct utsname`. There UTS stands for *Unix Timesharing System*, which is a term used for early Unix research systems developed at Bell Labs by Ken Thompson and Dennis Ritchie. The purpose of UTS namespaces is

[12]`http://plan9.bell-labs.com/magic/man2html/1/bind`

```c
struct utsname {
        char sysname[];     /* Operating system name (e.g., "Linux") */
        char nodename[];    /* Host name */
        char release[];     /* Operating system release (e.g., "3.13.0") */
        char version[];     /* Operating system version (e.g. build number etc.) */
        char machine[];     /* Hardware identifier */
        #ifdef _GNU_SOURCE  /* The domainname member is a GNU extension */
        char domainname[];  /* NIS or YP domain name */
        #endif
};
```

Listing 1: Definition of the `utsname` structure from the GNU C Library

to isolate two system identifiers returned by the `uname()` system call; `nodename` and `domainname`. The definition of `struct utsname` from the GNU C Library (glibc) is shown in Listing 1. In practice, this allows VPS containers to have a hostname and NIS domain name of their own.

The implementation of UTS namespaces is simple enough to function as an illustrative example of the changes to the mainline Linux kernel that the introduction of resource namespaces has required. Let us consider the `gethostname()` system call which is used to get the system hostname from kernel space into userspace. Like in the case of `uname()`, the information originates from an in-kernel representation of the `utsname` structure. Unlike `uname()`, `gethostname()` merely copies the `nodename` member, not the entire `utsname` structure.

Prior to kernel version 2.6.19, in which UTS namespaces were introduced, the `gethostname()` system call would access the `nodename` member of a global `utsname` structure, the `system_utsname`. Listing 2 shows a fragment of the implementation of the `gethostname()` system call prior to the introduction of UTS namespaces. From kernel version 2.6.19 onwards, two helper functions, `utsname()` and `init_utsname()`, were introduced. Consequently users of `system_utsname` were modified to use these helpers instead. Instead of accessing a global `system_utsname`,

```
asmlinkage long sys_gethostname(char __user *name, int len)
{
        int i, errno;
        ...
        i = 1 + strlen(system_utsname.nodename);
        ...
        /* Copy the nodename member from the global system_utsname to userspace */
        if (copy_to_user(name, system_utsname.nodename, i))
                errno = -EFAULT;
        ...
}
```

Listing 2: Fragment from the `gethostname()` system call prior to 2.6.19

```
static inline struct new_utsname *utsname(void)
{
        return &current->nsproxy->uts_ns->name;
}
```

Listing 3: Definition of `utsname()` helper function in 2.6.19

the `utsname()` helper returns a pointer to a `utsname` structure in the UTS namespace associated with the current process. Listing 3 shows the definition of `utsname()`. The `nsproxy` member in the `task_struct` for the `current` process is a structure which contains pointers to all namespaces associated with the process. In current kernel versions, the definition of the `gethostname()` system call is similar to the fragment shown in Listing 4. Access to the `utsname` structure occurs via the helper function.

The `gethostname()` example illustrates a pattern common to the currently implemented Linux namespaces; previously global resources are encapsulated in a per-process namespace. A handle to the namespace, and in extension any resources it encapsulates is maintained for each process.

```
asmlinkage long sys_gethostname(char __user *name, int len)
{
        int i, errno;
        struct new_utsname *u;
        ...
        u = utsname();
        i = 1 + strlen(u->nodename);
        ...
        if (copy_to_user(name, u->nodename, i))
                errno = -EFAULT;
        ...
}
```

Listing 4: Fragment from namespace-aware `gethostname()` system call

UTS namespaces might seem relatively harmless. However, access to them must be restricted to avoid scenarios where applications relying on the hostname are fooled into misbehaving because the system appears to have an unexpected hostname. For instance, applications might use the hostname as part of a lock file pathname. Running such an application inside a UTS namespace with an altered hostname can be used to circumvent the lock file, possibly leading to misbehavior in application instances running in different UTS namespaces.

2.5.3 IPC namespaces

IPC namespaces [Ker13a] allow processes to unshare Inter-Process Communication primitives and have a private set of primitives which are identified by other means than filesystem pathnames. Namely this includes System V IPC objects and POSIX message queues.

2.5.4 PID namespaces

The objects of isolation in PID namespaces [Ker13a, Ker13c, Ker13d] are process ID numbers. Processes in different PID namespaces may obtain the same PID. In addition, PID namespaces may be nested. A process a receives a PID for each layer of PID namespaces it resides in. A process is visible in its own namespace and all ancestors, although via a different PID number. A process in a particular namespace is not visible in child namespaces.

The first process created within a PID namespace receives a process ID of 1 within the namespace and becomes thus the `init` process for the namespace. In a similar manner to the `init` process in the root namespace of the host that has a special role in the system, so do `init` processes within a namespace. In particular, the `init` process of a namespace becomes the parent of processes that become orphaned within the namespace. In VPS environments, the `init` process for the namespace is also responsible for starting system daemons and other processes part of the environment within the namespace.

In order to prevent the essential `init` process in the root PID namespace from being accidentally killed, only signals for which the process has established signal handlers are delivered to the `init` process. Similarly, the `init` process in child PID namespaces only receives signals for which it is has established signal handlers from processes within its namespace. However, contrary to the `init` in the root PID namespace, the `init` within a child namespace may receive signals from processes in ancestor namespaces. Unless the `init` process has established signal handlers for them, these are ignored as well, with the exception of `SIGKILL` and `SIGTERM`. These are forcibly delivered, and can be used by processes in ancestor PID namespaces to stop or terminate the child namespace `init` process. If the `init` process in a namespace terminates for some reason, the kernel proceeds to terminate all other processes within the namespace via `SIGKILL` signals, essentially shutting down the container in a VPS environment. This also has the effect of destroying[13] the, now empty, PID namespace.

Instances of the proc filesystem mounted from within a PID a namespace, display only information regarding processes inside the namespace (and nested namespaces) via `/proc/`*`pid`*`/`. While a procfs instance mounted from outside the PID namespace a process belongs to will display *pid* subdirectories for processes in another PID namespace, those PIDs will not be meaningful for processes in any other PID namespace. System calls made by processes always interpret PIDs in the context of the PID namespace in which they reside. As various utilities like `ps` rely on the procfs being mounted at the traditional `/proc` mount point, PID namespaces are usually used in combination with a mount namespace to allow the procfs for the namespace to replace the parent profcs mounted at `/proc`. Another possibility is to confine processes in a PID namespace to a `chroot()` jail (see Section 2.3) and mount a procfs at `/proc` within the confined directory hierarchy.

2.5.5 Network namespaces

Network namespaces [Ker14] encapsulate resources useful to containers from a networking perspective. Each network namespace contains a logical copy of of the network stack, with ts own network interfaces, IP addresses, IP routing tables, port numbers and `/proc/net` directory (as long a procfs is mounted from within a namespace, as with PID namespaces).

[13]An unusual corner case related to the dismantling of PID namespaces is that the namespace will not be destroyed as long as `/proc/`*`pid`*`/ns/pid` descriptor to it is held open. It is, however, impossible to create new processes in the namespace (via `setns()` or `clone()`), as the `init` process for the namespace is no more (`clone()` fails in this case with an `ENOMEM` error value to indicate that a PID cannot be allocated.)

Network namespaces can be configured from userspace via the `ip` utility[14]. By convention, configuration files that would ordinarily be stored under `/etc`, such as `/etc/resolv.conf`, can be made specific to a particular network namespace by storing them under `/etc/netns/`*<name>*, e.g. `/etc/netns/`*mynetns*`/resolv.conf`. In applications that are aware of network namespaces this search path will precede `/etc` when looking for global configuration files. For applications that are unaware of network namespaces, the namespace specific configuration files may be bind mounted over their regular counterparts under `/etc`. By utilizing mount namespace, this can be done without disrupting the behaviour of processes outside the namespace. The `ip` utility also supports processless network namespaces by exposing the namespace descriptor via `/var/run/netns/`*name*`/`. Processless network namespaces may be kept alive by keeping the corresponding file descriptor open.

2.5.6 User namespaces

The principal motivation for user namespaces [Ker13e] is to allow unprivileged users to safely unshare namespaces. Users will be privileged with respect to the new namespace, but restricted to resources they already own. User namespaces also provide separate limits and accounting for UIDs in different namespaces.

As with PID namespaces, user namespaces may be nested. The parent of a user namespaces is the user namespace of the process that creates the child user namespace. An exception to this rule is the initial host system user namespace, which has no parent. Each user namespace may have zero or more children.

The current implementation of user namespaces introduces two new types to represent in-kernel UIDs and GIDs; `kuid_t` and `kgid_t` [Cor12]. The purpose of the kernel UID and GID is to describe a process's identity on the host system, regardless

[14]Network namespace support was added to `ip` in version 3.0.0 of *iproute2*
`http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`

```c
typedef struct {
        uid_t val;
} kuid_t;
```

Listing 5: In-kernel UID definition

of any UIDs it may appear to have inside a user namespace. The namespace-specific IDs visible to userspace retain the integer types `uid_t` and `gid_t` as before. However, most privilege checks are done based on the kernel IDs.

The `kuid_t` and `kgid_t` types are simply C typedefs for single-field structures containing the corresponding integer UID or GID type (see the definition of `kuid_t` in Listing 5). This makes `kuid_t` and `kgid_t` type-incompatible with the integer UIDs and GIDs. This is intentional in order to cause mismatched operations between kernel IDs and userspace IDs cause compiler errors. The rational is that this will help to avoid a large portion of potential errors in kernel code that deals with user and groups IDs.

The userspace UIDs map one-to-one to kernel UIDs. The translation occurs at the boundary between kernel space and userspace. The mapping is established by writing mapping information to the `/proc/`*`pid`*`/uid_map` and `/proc/`*`pid`*`/gid_map` files corresponding to one of the processes in the user namespace the mappings affects. Initially the files are empty. Mapping rules are added to the files in the form of integer 3-tuples consisting of:

$$< ID{-}inside{-}namespace > \quad < ID{-}outside{-}namespace > \quad < length >$$

where $< ID - inside - namespace >$ together with $< length >$ defines a range of UIDs/GIDs inside the affected namespaces that are mapped to a corresponding range in the immediate parent namespace starting at $< ID{-}outside{-}namespace >$. Note that from the point of view of userspace processes, mappings are always done relative to two user namespaces, although from the point of view of the kernel the mappings are always flattened directly to and from kernel IDs, even with multiple layers of nested user namespaces.

Mapping rules may be added by processes either inside the affected namespace, or inside the immediate parent user namespace. In addition, the process performing the mapping must have write access to `/proc/`*`pid`*`/uid_map` and `/proc/`*`pid`*`/gid_map`, which are owned by the UID that created the user namespace the process identified by *`pid`* belongs to. The process performing the mapping must also have the `CAP_SETUID` (for `uid_map`, or `CAP_SETGID` for `gid_map`) capability in the affected user namespace.

Mappings may only be established once per user namespace, i.e. only one write can be performed to any mapping files belonging to a particular user namespace. If the process performing the mapping possesses the `CAP_SETUID` (for UID mappings) or `CAP_SETGID` (for GID mappings) in the *parent* namespace, it is allowed to define mappings to arbitrary user or group IDs in the parent user namespace by performing a single write to the corresponding mapping file that may contain multiple newline-delimited triples. If the process does not possess the necessary capabilities in the parent namespace (as in the case of the initial process in a newly created user namespace), it is only allowed to write a single line to each mapping file to map its UID or GID inside the namespace to its corresponding effective UID or GID in the parent namespace. This arrangement guarantees that privileged processes inside a user namespace remain unprivileged with regards to the parent namespace, i.e. they cannot elevate their privileges via an ID mapping unless they already possessed the corresponding privileges in the parent user namespace.

A problem with this arrangement as described so far is that it does not lend itself to server consolidation, which is perhaps the most important use case for the namespace isolation mechanisms in the first place. The reason is that in server consolidation, it is often desirable to allow an unprivileged user with regards to the host system effectively full privileges in a container assigned to them, including the ability to create additional user accounts. The problem we encounter in the above arrangement is twofold; firstly, the user initializing the container, i.e. creating the namespaces lacks the privilege to add mappings for other than the initial UID and GID inside the container to his UID and GID on the host; secondly, the user lacks knowledge of suitable ID ranges outside the namespace to use as targets for mapping distinct IDs inside the namespace to. These issues are solved in userspace by introducing the notion of *subordinate UIDs* and *subordinate GIDs*. These are IDs, typically above `100000` that can be allocated to users, either manually, or automatically upon user creation by the root user[15]. The allocations are stored in `/etc/subuid` or `/etc/subgid` respectively. These can be mapped to UIDs in containers the owning user creates via the use of the setuid programs `newuidmap` and `setgidmap`. This userspace support allows unprivileged users to establish ID mappings from a container user namespace to subordinate IDs assigned to them.

2.6 Resource Control

In addition to resource namespaces and access control, *resource control* provides a third approach to limiting the access processes have to system resources. It concerns itself with the utilization CPU, memory, disk I/O and other finite resources that are consumed by processes. It is instrumental in preventing *Denial of Service* conditions due to resource exhaustion.

[15]Version 4.2 of the shadow tool suite adds support for subordinate ID assignment. Manual assignment can be done using command line options added to the `usermod` utility, and automatic assignment can occur based on appropriate stanzas added to the `login.defs` configuration file. The `newuidmap` and `newgidmap` programs are also provided by the shadow tool suite. `http://pkg-shadow.alioth.debian.org/`

Linux *Control Groups* (cgroups) provide a mechanism for partitioning processes into hierarchical groups that constitute units of resource control. All processes in a control group are bound by similar resource consumption criteria. Often this criteria may be inherited by the parent group. As such, control groups provide a unified interface to resource control scaling from single processes to entire userspace instances, as in the case with containers.

Control groups support *resource limiting*, e.g. memory consumption limits that may not be exceeded; *prioritization*, e.g. the assignment of a certain share of the CPU throughput to a particular group; *accounting* that allows the use of system resources to be measured for e.g. billing purposes. Control groups can also act as the unit for freezing processes for the purposes of CR.

2.7 Discretionary and Mandatory Access Control

The core access control model in Linux is based on Unix *Discretionary Access Control* (DAC). In Unix DAC, the owner of a filesystem object is allowed set the security policies associated with the object. In other words, the access control policies for filesystem objects are at the *discretion* of their respective owners. Unix DAC is typically implemented via a bit mask associated with the inode of a filesystem object. The bit mask represents a simple *access control list* (ACL), with each bits corresponding to a certain type of privilege, e.g. *read*, *write*, *execute* etc. Traditionally, distinct access rights can be assigned to the *owner* of the object, users in a certain *group* associated with the object, and *other* users not in one of the previous two categories.

The security requirements for many use cases require more fine-grained access control than what traditional Unix DAC can provide, including the ability to specify a centrally controlled access control policy, which cannot be overridden by users. This type of access control is commonly referred to as *Mandatory Access Control* (MAC). MAC is typically formalized as an access control matrix which constrains the ability of a *subject* to access or generally perform some sort of operation on a target *object*.

The necessary infrastructure for MAC in Linux is provided by the *Linux Security Modules* (LSM) framework. The LSM framework consists of a series of hooks in kernel code at points where access control decisions are made. LSMs implement these hooks, providing access control decisions based on their individual access control schemes. In MAC LSMs, the subjects are typically individual processes, as opposed to users.

MAC LSMs in Linux can be categorized into two groups; *label* and *path*-based. In label-based LSMs, all subjects (processes) and objects on the system (such as files) are assigned *security labels*, typically stored as *extended filesystem attributes* (xattrs). All interaction between subjects and objects is subject to review by the LSM, which consults its security policy to determine if the access should be allowed.

Security Enhanced Linux (SELinux) is currently the most widely deployed label-based LSM. It was originally developed at the Trusted Systems Research division of the United States National Security Agency (NSA) for the purposes hardening Linux for use in government and military systems which manage classified information. SELinux is an implementation of the *Flux Advanced Security Kernel* [SSL+99] (FLASK) operating system security architecture. SELinux provides a feature-rich policy definition language used by software maintainers and administrators to formulate a system policy. SELinux also provides a rigorously structured reference policy that is commonly used as a basis for the system policy shipped with Linux distributions using SELinux such as Fedora and Red Hat Enterprise Linux distributions.

SELinux policy configuration requires expertise in the behavior of applications subject to MAC. SELinux policies also tend to be quite large. The Fedora SELinux policy for instance defines over 700 distinct classes of subjects, over 3000 classes of objects and close to 100 000 access rules. As a result, SELinux has been criticized for being too complex for many use cases. The *Simple Mandatory Access Control* (SMACK) LSM was developed as an alternative to SELinux. SMACK also features a label-based scheme providing a basic form of MAC, but with a much simpler grammar for policy definition. SMACK supplied the MAC for the *Mobile Simplified Security Framework* [KREA11] (MSSF) used in the MeeGo mobile platform, and its successor Tizen [16].

In contrast, path-based LSMs do not make access control decisions based on on-disk security labels (e.g. xattrs), but policy definition is done based on filesystem pathnames. This has the benefit of policies which are more easily amendable to support different access control policies for the different appearances of the same object (e.g. symbolic links, bind mounts etc.), but may require additional controls to avoid circumventing the policy by means of such alternate pathnames. Path-based LSMs in Linux are AppArmor[17] and TOMOYO[18].

The development of TOMOYO was also in part motivated by the perceived complexity of SELinux policy configuration. Most MAC LSMs support a *permissive mode*, in which policy violations are logged, but not prevented. This can be useful during policy configuration. A distinguishing feature of TOMOYO is a *"learning mode"* which allows a policy to be automatically generated based on the observed behavior of processes.

[16] `https://www.tizen.org/`

[17] `http://wiki.apparmor.net`

[18] `http://tomoyo.sourceforge.jp/`

AppArmor is a successor to security extensions originally developed for Immunix, a commercial Linux distribution specializing in host-based application security. Immunix Inc., which was eventually acquired by Novell also developed AppArmor for Novell's SUSE Linux. AppArmor is currently maintained by Canonical Inc., and is the default LSM used by Ubuntu.

3 Design Goals

The research question behind this thesis is to study the feasibility to utilize existing Linux kernel features to enable the ability to migrate the data and settings of user installed applications between distinct, general purpose devices. We consider this functionality to be useful for both personal devices and shared environments. Usage scenarios involving personal devices include setting up a new device purchased by a user, as well as rental devices, which remain in a user's possession only for a limited time. Shared environments may include *In-Vehicle Infotainment* (IVI) systems or smart displays which would allow users to temporarily provision application to the devices. In these cases a user would typically only have a transitory relationship with the device. Since the same device may allow multiple users to interact with it, possibly simultaneously, providing suitable isolation of applications becomes especially important. Different device stakeholders may also wish to place restrictions on the use of the device, which might differ from the restrictions on the device from where the migrated application originates.

When used for server consolidation, containers are typically used to house entirely self-contained environments, which share only the underlying operating system kernel. Even with potentially untrusted containers, assuming that host kernel compromise is not viable, this kind of setup poses very few points of contact to other containers. Also, migration of such an environment to another physical server is straightforward, especially when the underlying hardware and host operating system setups are largely homogeneous. If the underlying storage is network-based, the migration can be a simple manner of shutting down the container on one host, and starting it on another.

In this work we consider deployment scenarios where it is infeasible for containers to be entirely self-contained. Consider for instance application containers, i.e. containers that only have a single application or service instance running inside them are commonly used for fault containment. When dealing with user-facing applications, it is desirable to allow applications to be containerized dynamically upon user request without the need for users to possess the expertise to configure the container environment themselves. Such applications may need to interact with host services, e.g. applications with graphical user interfaces should still be allowed to draw to the screen, receive input etc., even though it is desirable that access to a subset of resources remain strongly compartmentalized.

While container migration in server environments is well understood, the migration of dynamically established applications containers that share various resources with the host or other containers poses additional challenges. The origin and target environment can exhibit heterogeneity, while it would be desirable for users that the containerized application would behave in the same manner as it did prior to the migration. Specifically, our problem statement is twofold:

- Can existing operating system-level virtualization mechanisms in the Linux kernel be used to set up, on demand, isolated security domains for applications?
- Can such dynamically established security domains be migrated from one device to another?

It should be noted that an application design pattern fairly common in modern mobile platforms are applications that store user data and preferences in the cloud, allowing the data to be retrieved when the user installs the application on a new device. This approach, however, comes with a number of drawbacks. Firstly, these kinds of cloud synchronization solutions are application or platform specific, and in the latter case requires use of platform specific libraries. Secondly, such applications tend to require continuous network connectivity during use in order synchronize user data with the cloud. Thirdly, these kinds of solutions can raise privacy concerns regarding the information stored in the cloud.

3.1 Requirements

In order to address both security and usability aspects in our design, we have identified the following requirements:

Isolation: Dynamically set up domains should be isolated from each other and other applications present on the device. However, the isolation mechanisms in place should not restrict the use of common operating system services (provided the applications inside the container possess the appropriate permissions).

Authentication: It should only be possible to migrate application containers to a device that belongs to a user or is temporarily in use by a user (such as a rental device). Furthermore application migration should only be done upon explicit user consent. If applications are allowed to be migrated to to a third party device without user consent, security issues such as privacy violation, abuse of credentials etc. might follow.

Policy migration: Although security policies associated with the migrated application container must be preserved, the existing security policies of the target device must take precedence during the migration. Some devices might have security policies established by other entities than the user. Mechanisms such as content protection schemes might disallow user applications certain permissions. Circumvention of such security measures by migrating applications with more lenient security policies on other devices must not be possible.

Full cleanup: It must be possible to fully remove a migrated application container and all associated data from a device. Failure to completely wipe out application data might lead to privacy violations etc., especially in the case of rental devices.

Interoperability: Applications should be able to to be migrated and continue operation without modification and without relying on uncommon frameworks or programming languages.

Deployability: In order to make the adoption of the system feasible, the core operating system of the target devices must not be modified. Even the same OS from different vendors can differ and core parts of the OS should not be interchanged, since this might introduce software flaws, leading to system instability or security issues. Furthermore requiring significant changes to the operating system is sure to hamper the adoption of such systems.

Performance: As the ultimate target of the application migration scheme are mobile devices with limited resources in terms of battery capacity, processing power, and storage space, the mechanisms facilitating application migration should be sufficiently lightweight in order to be fast and efficient on contemporary mobile devices.

Usability: From a usability perspective, the migration process should be easy to learn, easily remembered as well as efficient and satisfying to use. Users should not be prompted for any information that is possible to obtain from the previous setup on the originating device, such as permissions, application preferences etc.

3.2 Assumptions

For the purposes of this thesis we limit ourselves to cold migration, i.e. the migrated application is halted before the migration occurs. We consider the problem of live migration orthogonal to our efforts; as long as the container setup provides an environment sufficiently independent from the host environment, support for a live migration system which serializes the state of a running application to filesystem objects can easily be incorporated as part of future work.

If users continue to use a migrated application container on both devices (i.e. the source and target device), the state of the applications will diverge. A previously migrated application container along with its state should be migratable back to the original device. In such cases synchronization of diverged application states becomes a challenge. For out purposes we consider a usage scenario where only one canonical copy of a container exists at any time. When the container is migrated, it ceases to be available on the source device.

In our design, the integrity of the container relies in part on the integrity and confidentiality of the transfer of data from one device to another. We leave securing this transfer outside the scope of this thesis, although we acknowledge that this is an integral part of the security of the scheme if it were to be deployed in a real-world scenario.

4 System and Threat Model

4.1 System Model

In this section we explain the specifics our target system model. The model described here is generalized in such a way that it may apply to many different variants of Linux-based operating systems. Our design is intended to allow for integration into systems which fit this general model. The main reference points used in the formulation of the system model are the freedesktop.org[19] base platform for desktop software that has become a *de facto* standard in modern desktop GNU/Linux

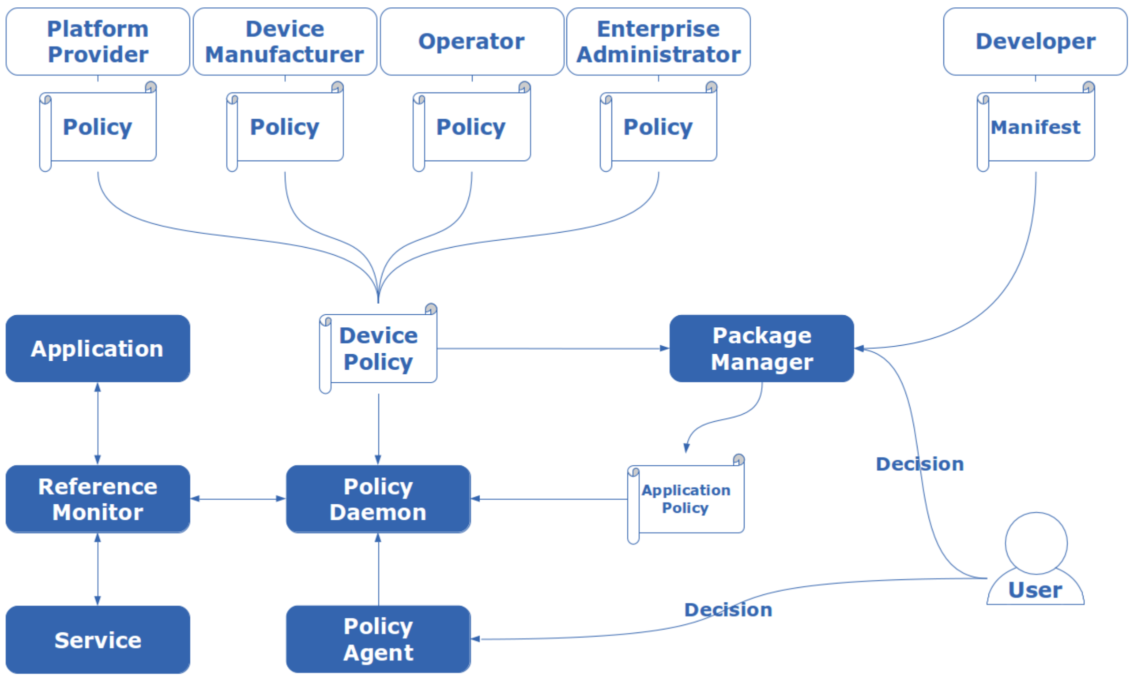


Figure 3: System model

distributions, and the MSSF which plays a somewhat similar role on some mobile Linux-based platforms. An overview of the components that comprise the system model are shown in Figure 3. These are described in detail below.

Permissions Modern application platforms use permission systems to protect access to resources that warrant protections, such as hardware capabilities and personal data. Prime examples of permission use can be found in prominent mobile platforms such as Android[20] and iOS[21]. However, the wide disparity in how permissions are presented to users and how users are involved in permission assignment is evidence of the fact there is no consensus on the best way to design such permission system [FEF+12].

For our purposes we consider a permission system where applications can acquire permissions in three ways:

[19]http://freedesktop.org
[20]https://www.android.com/
[21]https://www.apple.com/ios/

- Automatically at install time.
- At install time as a result of explicit user consent.
- At runtime as a result of explicit user consent.

If allowed by the device policy the user is allowed to grant a *blanket permission* when prompted for consent on a particular access control decisions. In this case, the user is not prompted again for consent on that particular permission and application pair, but the permission is granted permanently.

Device Policy: The device security policy is the collected state of the system security framework. It is influenced by the needs of different stakeholders. For contemporary mobile platforms, this includes the platform provider, device manufacturer, mobile operator and in some cases an enterprise administrator. The device policy is not always managed centrally.

For our purposes, the device policy consists of a list of permissions that may be assigned to applications. The device policy defines if a permission is allowed or denied automatically, or if the permission is granted only after being approved by the user. If a permission is not defined in the device policy, it is denied by default to any application which might request it.

Package manager: The package manager is responsible of overseeing the installation of applications. The package manager also grants any automatically granted permissions to the application, as per the device policy, and prompts the user to confirm any permissions that require his or her consent. The result is an application policy, which accurately depicts the permissions the application has been granted on a particular system.

Reference monitor: Most contemporary Linux-based systems include a framework for IPC at a higher level of abstraction than traditional Unix System V IPC primitives. This usually involves a message bus architecture, such as in the case of D-Bus in freedesktop.org-compliant systems, or a component object model, such as in the case of the Android Binder [FCH+11]. Both of these facilitate *Remote Procedure Calls* (RPC) between applications. The reference monitor is responsible for mediating access on the IPC channel.

We note that adding support for allowing LSMs to mediate IPC occurring over the message bus is being pursued by several parties. At the time of writing, active efforts are being made in order to enable mediation of D-Bus IPC by AppArmor [22,23] and SMACK[24]. Similarly in Android, SELinux allows userspace object managers to retrieve the SELinux context of the calling process in RPC occurring over Binder. It has even been proposed that the SELinux context of the caller should be passed with each Binder call[25] in order to avoid potential *Time-of-Check Time-of-Use* (TOCTOU) race conditions[26]in cases where the calling process context changes during the time between the Binder call and the context query is made. The *X Access Control Extension* (XACE) allows the access control of X11 display server graphics objects in a manner similar LSMs in the Linux kernel. XACE support for SMACK is also being done [Sha09].

Policy daemon and agent: The policy daemon decides if intercepted access attempts are allowed or not. In systems where the underlying policy is managed by an LSM, the policy daemon might not have knowledge of the policy itself, but consult the LSM when policy decisions are to be made. In our system model we describe the policy daemon as a separate entity, but it is entirely possible that the responsi-

[22]`https://blueprints.launchpad.net/ubuntu/+spec/security-o-apparmor-dbus`

[23]`https://wiki.ubuntu.com/SecurityTeam/Specifications/Oneiric/AppArmorDbus`

[24]`https://bugs.freedesktop.org/show_bug.cgi?id=47581`

[25]`https://code.google.com/p/android/issues/detail?id=72971`

[26]CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition
`http://cwe.mitre.org/data/definitions/367.html`

bilities of the reference monitor and policy daemon are placed in a single software component.

The policy agent is the part visible to end users. It's responsible for interacting with the user when the user is involved in access control decisions.

4.2 Threat model

In our threat model we assume that a certain subset of containers are fully controlled by an attacker, while the remaining containers are legitimate. Attacker goals include:

Compromise of Legitimate Containers or Host: Means of compromise include illegitimate access to information belonging to legitimate containers or host including file system resources, information about running processes, network traffic etc. More serious threats include the ability to mount *Man-in-the-Middle* (MitM) attacks on network communications or IPC, or the ability to affect the control flow of programs executed in a legitimate container or the host.

Privilege Escalation: The ability to obtain privileges not originally granted to processes in the container. This includes both applications permissions, as well as operating system capabilities, which could potentially allow the attacker to override the protections set in place by the permission system.

Denial of Service: The ability to disrupt the normal operations of the host or legitimate containers. Means of DoS include resource exhaustion of CPU, memory or persistent storage.

An earlier version of our threat model appears in the survey by Reshetova et.al. [RKNA14].

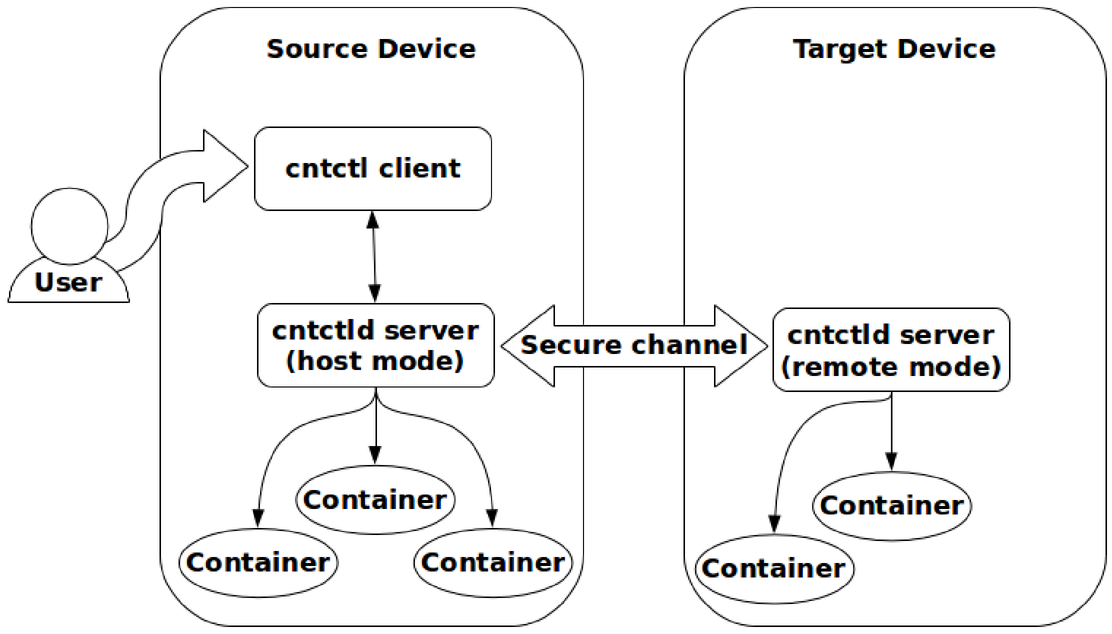


Figure 4: Architecture

5 Design and Implementation

In this section we present our design and describe the implementation of a proof-of-concept prototype.

5.1 Architecture

Our design consists of a client-server architecture comprising of a client (`cntctl`), and a server daemon (`cntctld)`). Each server instance operates in one of two distinct modes of operation we call *host mode* and *remote mode*. In host mode, the `cntctld` server provides a number container management services to local clients, including the creation of containers and the execution of containerized application. In remote mode, the server principally acts as an endpoint for container migration for remote clients. An overview of the architecture is shown in Figure 4. Container migration in the other direction (back from the target to source device) uses a symmetric setup.

The `cntctl` client implements the user-visible interface for container management. This component is entirely unprivileged, and needs a server instance operating in host mode to perform privileged operations, such as namespace creation on its behalf. Container creation consists of the host mode server spawning an initial process in the container (henceforth referred to as the *container init*) which together with the host mode server coordinates the setup of the containerized environment, and finally executes the target applications placed inside the container.

The target environment for our prototype enforces applications permissions using SMACK. The device policy defines a number of SMACK domains that correspond to security domains applications may be granted access to. For the purposes of our prototype we consider a *Three Domain Model*[27]. The top-level security domains are subdivided into *peer domains* corresponding to distinct permissions (e.g. `System::Audio` for audio access). In our prototype the host mode server is responsible for the loading container specific policy rules in accordance with the device policy to the SMACK policy maintained in the kernel. In a real-world deployment we see that this will occur in collaboration with the system components described in Section 4.1.

5.2 Filesystem compartmentalization

To minimize the overhead with regards to filesystem resources that must be made available to the container (e.g. shared libraries and other dependencies) our design allows such resources to be shared between the host and a guest container. In order to avoid information about processes on the host or other containers to be accessible from inside a container, and to prevent processes inside the container from

[27]`https://wiki.tizen.org/wiki/Security:SmackThreeDomainModel`

potentially influencing processes outside the container, we confine the container to a subtree where host resources are made available via the use of an *overlay filesystem*[28].

Overlay filesystems are a way to provide a unified view of two directory trees (sometimes referred to as *branches*) as a single hierarchy. The directory trees involved in the union are designated as the '*upper*', and '*lower*' branch. When a particular pathname exists in both branches, the corresponding filesystem object in the upper branch is visible in the union, while the object in the 'lower' branch is either hidden or, in the case of directories, merged with the corresponding object in the 'upper' branch. The lower branch may be read-only, as the overlay file system provides copy-on-write semantics for operations on the union. When a file residing on the 'lower' branch would be modified, the operations results in a *copy-up* of the file to the 'upper' branch. The modifications are always made to the copy in the 'upper' branch. Removal of files and directories in the union are recorded to the 'upper' branch as *whiteouts*, which cause the corresponding pathname to in the 'lower' branch to be ignored in the union. The whiteout itself is also hidden.

The particular overlay filesystem implementation utilized in our prototype is *OverlayFS*[29]by Miklos Szeredi. OverlayFS is characterized as a 'hybrid' approach to overlay filesystem because unlike its predecessors, such as *Unionfs*[30] and *Aufs*[31], which provide virtual filesystems with persistent inode renumbering and dynamic branch management, OverlayFS merely modifies pathname lookups. This occurs mainly in the `readdir()` system call. As a result, filesystem objects that appear in the union do not all appear to belong to that filesystem. Non-standard behavior includes any file locks obtained before a copy-up not applying to the copied up file as that is essentially a new file appearing with the same pathname. Similarly the

[28]The term 'overlay filesystem' is somewhat of a misnomer, as most implementations in fact operate on directory subtrees that may very well exists on the same filesystem.

[30]http://unionfs.filesystems.org/

[31]http://aufs.sourceforge.net/

inode number of the file may change as a result of a copy-up, as well as the device number if the branches exist on different file systems. We expect that for many applications these differences can be ignored. OverlayFS mounts are also limited to two branches, although several OverlayFS mounts may be nested in order to unify more than two directory trees. The level of nesting is currently limited to a stack of two OverlayFS mounts by an arbitrary value hardcoded into the kernel.

We also utilize the fact that modern Linux distributions, including Debian and Fedora, have introduced `/run` to store files that contain runtime information which does not require preserving across reboots. The `/run` directory is a mount point for a *tmpfs* instance, a temporary file storage facility that appears as a mounted file system, but stores files in volatile memory instead of persistent storage. This location replaces several existing locations described in the *Filesystem Hierarchy Standard*[32], including `/var/run`, `/var/lock`, and `/dev/shm`[33]. For backwards compatibility, these directories are provided by symbolic links pointing to appropriate locations under `/run`.

In our design, each container is assigned a subtree in the filesystem hierarchy that is a result of the union of the host root filesystem and a writable branch which acts as a copy-on-write overlay for the container. This makes files on host root filesystem visible, and even writable inside the container with the necessary privileges, but any

[31] `https://git.kernel.org/cgit/linux/kernel/git/mszeredi/vfs.git/?h=overlayfs.current`

[32] `http://www.pathname.com/fhs/pub/fhs-2.3.html`

[33] Recently there has also been attempts to switch `/tmp` to tmpfs as well. Reasons for this include minimizing the wear on *Solid-State Drives* (SSDs) utilizing flash storage technology with a limited number of write cycles. These attempts have, however, been met with a fair amount on controversy. Counter-arguments revolve around the added complexity caused by breaking the register/cache/memory/disk hierarchy. For instance, as tmpfs size is limited to a certain percentage of main memory, it is not suitable for storing (very) large files. For the purposes of this thesis we assume a that `/tmp` is backed by a tmpfs and treat it like other transient file storage.

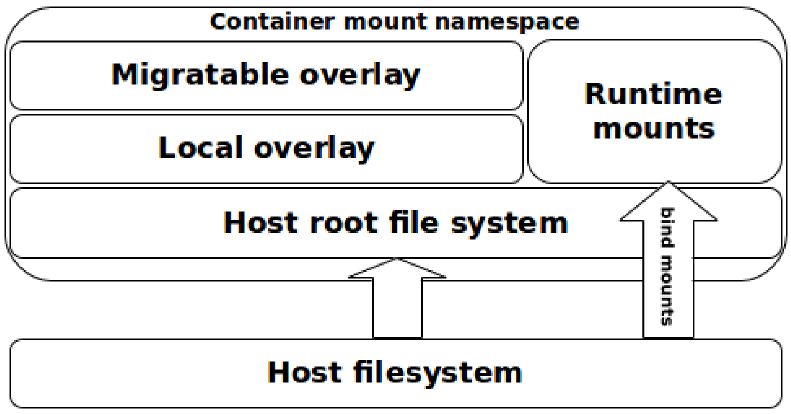


Figure 5: Container filesystem compartmentalization

changes made to the root filesystem are stored by the copy-on-write overlay, and thus do not affect the actual host root filesystem. Transient files stored in `/run` are not visible to the container, as they exists on a separate filesystem and thus not made visible by OverlayFS. The copy-on-write overlay also allows us to isolate files for migration. We use another, nested overlay for local, non-migratable container specific files (see Section 5.7). The local overlay remains read-only. Additional locations, for which copy-on-write semantics are undesirable can be made available to the container via bind mounts. All this occurs before the initial process in the container is spawned. A schematic view of the setup is shown in Figure 5

We note that as an alternate approach, the read-only root filesystem for the container can be constructed by bind mounting desired locations from the host to locations where the conventional mount points have been prepared. This precludes the need for overlay filesystems, but also prevents writing to shared locations from inside the container.

In order to prevent processes from accessing filesystem resources outside the subtree designated for the container, the container init process employs the Linux specific `pivot_root()` system call in combination with the conventional `chroot()` system call and a distinct mount namespace created for the container to render the host root filesystem inaccessible to processes placed in the container.

The `pivot_root()` system call is typically used during a Linux boot sequence to change from a temporary root filesystem (e.g. an initrd) to the actual root filesystem on a block device. As its name suggests, the *pivot_root()* system call moves the mount point of the old root filesystem to a directory under the new root filesystem, and places the new root filesystem at its place. When done inside a distinct 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, or any other mount namespaces 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 mount point of the new root filesystem if they point to the old root directory. This prevents kernel threads from keeping the old root directory busy with their root and current working directory. It seems that the intention is to keep changes to the process directories separate from changes to the root filesystem, so in the future there may be a mechanism for kernel threads to explicitly relinquish any access to the filesystem. This would allow this mechanism to be removed from `pivot_root()`[34]. As the root directory and working directory may or may not change as a result of calling `pivot_root()`, proper usage dictates that he 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, we change the root directory and current working directory to the mount point of the new root filesystem subsequent to invoking `pivot_root()`, then proceed to perform a lazy unmount of the old root filesystem. This makes the mount point immediately unavailable for new accesses,

```
/* Change the current working directory (cwd) of the calling process
 * to the new root directory. */
if (chdir(rootfs) == -1) { /* Abort if change of cwd fails */ }

/* put_old contains a relative path to a location under the new rootfs
 * used as the destination for the old rootfs after the pivot. */
if (pivot_root(".", put_old) == -1) { /* Abort of the pivot fails */ }

/* Depending on the implementation of pivot_root(), the root directory
 * and cwd of the caller may or may not change. Calling chroot() (and
 * subsequently chdir()) here makes sure that the root directory (and
 * cwd) change regardless of whether pivot_root() has changed the root
 * directory or not. */
if (chroot(".") == -1) { /* Abort if the change of root directory fails }

/* Perform lazy unmount of the old rootfs. This makes the mount point
 * immediately unavailable for new accesses. */
if (umount2(put_old, MNT_DETACH) == -1) { /* Abort if lazy unmount fails */ }
```

Listing 6: filesystem compartmentalization system call sequence

but the actual unmount is delayed until a time when the mount point ceases to be busy. The corresponding sequence of system calls is shown in Listing 6.

5.3 Process compartmentalization

Process compartmentalization is provided by a separate PID namespace created for each container. This, combined with the mount namespace used for filesystem compartmentalization allows a separate procfs instance to be mounted at `/proc` inside the container. As described in Section 2.5.4, profcs instances mounted from within a PID namespaces only provide information on processes within that namespace. An IPC namespace can also easily be combined with the setup described so far. It

[34]This design rationale is documented in the Linux Programmer's Manual page on *pivot_root(2)* from July 13, 2012 (release 3.73).

requires no additional setup apart from adding the appropriate flag to the `clone()` call when the container init process is spawned.

As described in the previous section of filesystem isolation, we also took care not to expose transient files which might contain process runtime information or communication primitives visible as filesystem objects, such as Unix-domain sockets, FIFOs and shared memory used for IPC within the container. To finish this off, environment variables which expose the (from within a container meaningless) locations of such IPC sockets may be pruned from the environment inside the container. Explicit exceptions can be made in cases where it is desired to allow applications within containers access to host services. For instance, we make communications sockets for the X11 display server and PulseAudio sound system[35]available inside the container along with the environment variables that define their location. This is achieved by bind-mounting the corresponding sockets to their respective locations within the container subtree. This allows applications inside the container to draw to the screen and play audio in a regular fashion. Access to the communication sockets are still mediated by the permission system, as described earlier.

5.4 User compartmentalization

In our prototype, the containerized applications are run with the UID of the user who created the container. Multiple concurrent users can in principle be supported by running multiple instances of the host mode daemon. This approach, however, does not lend itself well to usage models where the user does not necessarily have a Unix user account on the target device. Examples of such usage scenarios are IVI systems or smart displays which act as shared computing environment with which the user has only a fleeting encounter with. For these kinds of environments we suggest a

[35]`http://www.freedesktop.org/wiki/Software/PulseAudio/`

model where multiple users are served by the same server instance running under a dedicated UID with a certain number of subordinate UIDs assigned to it. A distinct user namespace is created for each container, and UIDs inside user namespaces are mapped to distinct subordinate UIDs.

5.5 Network compartmentalization

Linux provides several possible approaches to provide networking for a container running in a distinct network namespace. A new created network namespace is initially empty, except for a *loopback interface.* Additional interfaces must be assigned to the network namespace via a kernel interface exposed through the *netlink* protocol. However, each network interface may only be present in a single network namespace at any time. In hosts with multiple physical interfaces, one or more physical interfaces may be dedicated to a container. In devices where this is not practical, the container must be assigned a virtual interface. We will discuss the use of two different types of virtual interfaces: *Virtual Ethernet* and *MACVLAN* interfaces.

A Virtual Ethernet device consists of a pair of virtual interfaces that act essentially as a pipe, i.e. Ethernet frames transmitted one peer are received by the other. When used with containers, a pair of Virtual Ethernet devices is created on the host. One peer is assigned to the container network namespaces, while the peer remaining on the host is connected to a virtual bridge. This can be used to either create virtual networks between container by linking them via different bridges, or provide a link to the outside network by binding the bridge to bound to a physical interface.

Whereas *Virtual Local Area Networks* (VLANs) allow a single network interface to be mapped to multiple virtual networks, MACVLAN interfaces perform the opposite role, allowing a single physical interface (usually referred to as the *lower device*) to be mapped to multiple virtual interfaces, each with its own MAC address. MACVLAN interfaces can operate in three distinct modes:

Private: In this mode the interface cannot communicate with any other endpoints on the lower device i.e. all incoming frames are dropped if their source MAC address matches one of the MACVLAN interfaces present on the host. This isolates containers from each other, preventing direct communication between containers, but not from the outside network.

Bridge: This modes provides MACVLAN endpoints bound to the same lower device a bridge that allows the to communicate with each other or the host directly without transmitting frames through the physical link.

Virtual Ethernet Port Aggregator (VEPA): In VEPA mode, transmitted frames are always sent out via the lower device, even if they are destined to other endpoints bound to the same device. If the physical switch on the receiving complies with the 802.1Qbg standard for *Edge Virtual Bridging*[36], it may act as a *Reflective Relay* (a.k.a "*Hairpin switch*"), allowing it to transmit the frame back on the same link it received it on. This not only allows containers to communicate with each other and the host, but has the added benefit of allowing network level policies to be enforced by the switches (e.g. DHCP filtering). Unfortunately this mode of operation is not widely supported yet.

5.6 Device compartmentalization

Each container is created with a distinct `/dev` with a minimal set of device nodes (in particular `/dev/null`, `/dev/zero`, `/dev/full`, `/dev/random` and `/dev/urandom`). In addition, we share certain device nodes with the host by bind-mounting them to

[36]http://www.ieee802.org/1/pages/802.1bg.html

appropriate locations on container creation. For instance, by making `/dev/dri/card0` we available enable hardware accelerated graphics within the container.

It is important that the devices that are visible to containers are chosen with care, especially in cases when processes within the container are allowed privileges. For instance, access to the device node corresponding to the block device housing the host root filesystem would allow a suitably privileged process inside the container to circumvent the filesystem compartmentalization described in Section 5.2.

5.7 Permission compartmentalization

Placing the container in a distinct SMACK domain without affecting existing policies on the device poses some challenges. At the time of writing, the Linux MAC LSMs have not yet been made aware of containers. In our prototype, we instead utilize the nested overlay filesystem to allow the relabeling of native binaries executed within the container. This allows both an containerized and non-containerized version of the application to co-exist on the host. The relabeling consists of appending an identifier unique to the container to the existing label. The host mode daemon is responsible for relabeling SMACK policy rules pertaining to the containerized application and loading them to the enforced policy. This particular functionality requires the host mode daemon to obtain privileges to override existing SMACK labels and modify the enforced policy.

5.8 Application migration

Application migrations consists of simply transferring the contents of the copy-on-write overlay to the target device. This can occur over a network connection, or some other medium. We have also experimented with a usage scenario where users carry the migratable containers with them on portable USB drivers. In this case, the containers were encrypted on-disk using a passphrase known to the users.

The overlay also contains a description of permissions assigned on the host device and the origin of the particular permission. When the containerized environment is recreated on the target device, the container specific policy is reconciled with the device policy on the target device according to the following rules:

1. If the device policy always allows a permission, it is granted automatically.
2. If the device policy always denies a permission, or the permission is not defined by the policy, it is never granted for any reason.
3. If the device policy disallows a blanket prompt for a permission, the user is prompted for consent upon each startup of the container.
4. If the device policy allows a blanket prompt for a permission, *and* the user has consented to a blanket prompt on the source device, the permission is granted automatically.

6 Results and Evaluation

Our design employs resource namespaces and mandatory access controls to confine applications within dynamically isolated domains based on the notion of containers. Resource namespaces provide the necessary building blocks to compartmentalize operating system resources available to containerized applications. The MAC based on the Linux LSM framework provides the basis of the permission system in our prototype. In this section we describe how each of requirements identified in Section 3.1 were taken into account in the design and how well out withstands against the threat model described in Section 4.2.

Isolation: Although Linux resource namespaces provide a flexible toolset for resource isolation, we believe our design is evidence of the fact that the evaluation of the security of container-based virtualization solutions should in detail take into account resources crossing namespace boundaries. Due to our goal of allowing containers to share resources with the host system, we needed to consider many venues of potential information leakage on the filesystem layer, including transient runtime files and host device nodes. In this light, we believe an approach to container construction where the containerized environment is initialized as empty as possible, and resources to be shared with the host are introduced one by one in a controlled manner is imperative in avoiding unintended information leakage.

Authentication: We did not explicitly address the issue of authentication in our design because existing solutions for demonstrative authentication, such as those widely-deployed in Bluetooth and Wifi protected setup [SVA09] can be layered on top of our design.

Policy migration: In our prototype we explored the migration of policies in the context of two devices with potentially conflicting device policies. Our scheme allows the policy associated with a migrated container to retain user policy decisions to the extent allowed by the local device policy.

Full cleanup: In our design, the copy-on-write overlay isolates all files subject to migration. Therefore, removing the files in the copy-on-write overlay should be sufficient to remove all potentially sensitive data associated with the container from a device after successful migration.

Interoperability: Since our design relies modifying the environment containerized applications are run in, rather than modifying the applications themselves, we believe our approach can be applied to a wide variety of applications. For applications which depend on services running on the device for persistent storage (e.g.

databases etc.), the container could be extended to include private instances of necessary dependencies. For the purposes of our prototype we only considered relatively self-contained applications which relied on files for persistent storage, but in principle our design could be extended to compartmentalize interdependent applications as the cost of a higher overall resource footprint.

Deployability: In this work, we have intentionally limited ourselves to existing features available in the mainline kernel with the exception of OverlayFS, which at the time of writing has been proposed for inclusion in Linux 3.18[37]. Furthermore, we have based our design on a system model which fits real-world Linux-based systems.

`https://lkml.org/lkml/2014/9/29/350`

Performance:

We evaluated the performance of our design with a small number of unmodified applications. Of these we wish to note especially *Extreme Tux Racer*[38]an OpenGL[39] racing game which requires 3D acceleration to operate properly.

Bind-mounts proved to be very flexible in allowing devices and IPC sockets to be shared across container boundaries. This allowed us to containerize applications utilizing hardware-accelerated OpenGL graphics with no human-perceivable degradation in performance.

We note that our prototype exhibits increased overhead in filename lookups due to the design of OverlayFS. For the applications we evaluated, this overhead turned out to be negligible, but it is possible that this may be an issue for applications which perform a large number of repeated filename lookups.

Compromise of Legitimate Containers or Host: From a security perspective exposing certain device nodes to containers poses a major challenge as device drivers

[39]https://www.opengl.org/

[39]`http://sourceforge.net/projects/extremetuxracer/`

pose and significant attack vector by exposing uncompartmentalized interfaces to code running in kernel space.

Privilege Escalation: Our design allows mandatory access control policies to be enforced for containerized applications. This forms the basis for the permission system considered in the system model described in Section 4.1. Permissions that can be granted to an application is limited by the local device policy. The policy reconciliation used in the migration scheme respects the device policy.

Our design relies on system services supporting RPC, such as D-Bus and the X11 display server being able to enforce access control policies based on callers security context in order to avoid so called *Confused Deputy*[40]attacks.

The lack of container-awareness in LSMs poses a challenge in our design. The need for compartmentalization of both device drivers and LSM policies has been known for some time [Bie06]. Our overlay-based relabeling is only a workaround for this deficit. Path-based MAC LSMs allow for some additional flexibility in this regard, but with a separate set of drawbacks.

Denial of Service: Although we also exposed `/dev/random` and `/dev/urandom` in our container setup, as these devices access a global entropy pool exposing these poses a couple or risks that are not easily addressed. For `/dev/random` there exists theoretical risk that a malicious container might be able to predict the output for another container or host [DPR$^+$13]. In addition, as `/dev/urandom` employs blocking semantics, there exists a very real risk that a malicious container may attempt to exhaust all available entropy in the entropy pool and thus mount a DoS on cryptographic applications relying on high-entropy random number generation for e.g. key generation.

[40]CWE-441: Unintended Proxy or Intermediary ('Confused Deputy')
`http://cwe.mitre.org/data/definitions/441.html`

Our prototype also does not currently provide resource control, making it susceptible to resource exhaustion attacks. In Section 2.6 we account for kernel features that can be used to address this.

7 Related Work

Zap [OSSN02] is a system for process migration for Linux. Zap introduces the notion of *PrOcess Domains* (PODs), which are used to contain migratable processes. A POD decouples processes from dependencies on the host operating system and other processes by providing processes a virtualized view of the operating system by associating virtual identifiers with operating system resources. The approach bears some resemblance to Linux namespaces, although Zap predates the Linux namespace implementation by several years. Zap virtualization is achieved by intercepting system calls and translating physical resource names in arguments and returns values to virtual names and vice versa.

During migration, a POD along with all processes it contains is suspended and the state of the POD written to a file. This checkpoint file can subsequently be used to restart the POD along with its processes and restore the state at time of the checkpoint. In order to reduce the amount of data that has to been transferred during a migration, Zap leverages distributed filesystems such as NFS to store the filesystem visible within a POD and mounts the filesystem within the POD's virtual filesystem hierarchy.

Linux-VServer[41] is another solution for compartmentalizing the userspace environment into multiple distinct units. It is mainly aimed at providing VPS environments for server consolidation. Linux-VServer is not based on mainline Linux namespaces, but distributed as a set of kernel patches that extend existing kernel structures to make them aware of Linux-VServer *contexts*. Each context hides processes outside its scope and prevents interaction between processes in different contexts. This is achieved by providing isolation of shared memory and IPC primitives, user and process IDs, Unix pseudo-teletype devices and sockets. Linux-Vserver also adds a per-context capability mask that limits the capabilities available to processes within a context. It also provides Plan 9-style private namespaces for filesystem isolation, chroot barrier support and a *unification* mechanism for sharing files between distinct contexts utilizing a shared filesystem. The purpose of unification is to reduce the overall resource consumption via the use of hard links. If the file is modified in a particular context, the link is broken to avoid the changes from being visible in other contexts.

OpenVZ[42] is a open source, container-based virtualization solution for Linux, commercialized by Parallels, Inc. OpenVZ is the basis for the commercial Parallels Cloud Server which supports both traditional and container-based virtualization for Internet hosting services. OpenVZ is notable, because the team behind it has made significant contributions to the operating system-level virtualization mechanisms merged in the mainline kernel. As a result, the OpenVZ tools are able to operate, although with reduced functionality, on a upstream Linux kernel.

One of the distinguishing features of OpenVZ is CR functionality that is supported via an *in-kernel* implementation. Efforts were made to merge CR support into the upstream Linux kernel. However, the considerable complexity of the in-kernel implementation eventually lead to the rejection of the OpenVZ CR patches.

Not deterred by the rejection of the in-kernel CR implementation, the OpenVZ team took another approach. *Checkpoint/Restore In Userspace*[43] (CRIU) is a users pace tool for Linux which allows running applications to be suspended, and checkpointed to the filesystem as a collection of files. These files can later be used to restore

[41]`http://linux-vserver.org`

[42]`http://openvz.org`

the application to the state at the point it was suspended, and continue execution. CRIU differs from the OpenVZ CR feature in that it is, as the name implies, mainly implemented in userspace, with only minor changes to the kernel exposing various internal kernel resources to be read and modified from userspace.

With the inclusion of resource namespaces in the mainline Linux kernel, a number of efforts to provide containers based on these have been proposed. The most prominent of these is the LXC Linux Containers project[44]. LXC provides an userspace interface for the mainline kernel operating system-level virtualization features. It can make use of the kernel resource namespaces, AppArmor and SELinux MAC profiles, seccomp, chroots, capabilities and control groups. LXC consists of the *liblxc* C library with language bindings for several other programming languages, and a set of a command line tools for managing containers.

The *libvirt* virtualization API[45]is a toolkit with the goal of providing a common interface to the difference virtualization capabilities supported in recent versions of Linux and other operating system. It supports both OpenVZ and LxC containers, as well as a number of traditional hypervisors such as the KVM/QEMU hypervisor on Linux, the XEN hypervisor on Linux and Solaris, the FreeBSD *bhyve*[46] hypervisor, the VMware ESX and GSX hypervisors and the Microsoft Hyper-V hypervisor. The *libvirt* API provides operations to provision, create, modify, monitor, control, migrate and stop the virtualized domains, depending on the capabilities of the underlying virtualization solution.

A recent user of operating system-level virtualization primitives in Linux is *Docker*[47], a platform for automating the deployment of distributed applications in cloud environments. In the Docker deployment model, system administrators can provide

[43]`http://criu.org`

[44]`http://linuxcontainers.org`

[46]`http://bhyve.org/`

[46]`http://libvirt.org/`

standardized environments to development, QA, or operations personnel in the form of Docker containers. Docker containers consist of a filesystem image used to set up the application to be deployed along with any dependencies. The Docker Engine provides packaging tools to automate the creation of containers to a high degree. Docker containers also allow differences in in the underlying infrastructure, such as the specific OS variant (distribution), to be abstracted away. The containers can then be deployed on any system which provides the Docker Engine.

Docker can use LXC, either on its own or via *libvirt* as a back-end for the execution environments used to run containers, but LXC was recently superseded by *libcontainer*[48] specifically developed for Docker, as the default back-end. The Docker *libcontainer* is a Go library which offers an interface to the kernel's without dependencies such as LXC. It has support for namespaces, control groups, capabilities, AppArmor profiles, network interfaces and firewalling rules.

Cells [ADH$^+$11] is a virtualization architecture for Android, which allows multiple virtual phones to run concurrently on a single physical phone. Only one virtual phone, the foreground virtual phone, is displayed at a time, while other virtual phones are invisible in the background.

Virtual phones are isolated from each other by the means of resource namespaces. Cells introduces a new kernel-level mechanism, *device namespaces*[49], which provide hardware resource multiplexing and isolation. Unlike other kernel namespaces, device namespaces do not virtualize identifiers, but are used by device drivers or kernel subsystems to tag data structures or register callback functions, which are called when a device namespace changes state. Each virtual phone in Cells is running in a device namespace of its own. Callbacks triggered when a virtual phone changes between foreground and background state allow devices to respond differently de-

[47]`https://www.docker.com/`

[48]`https://github.com/docker/libcontainer`

pending on whether a virtual phone is in the foreground or in the background. The implementation of device namespaces in cells provides either wrappers around existing device drivers, modifies kernel subsystems to take into account device state or modifies individual device drivers to take device namespace state into account.

In 2011, the company Cellrox[50] was founded to commercialize Cells. The Cellrox multi-persona platform is marketed as an enterprise mobility solution enabling BYOD. It adds remote management support to Cells that allows corporate IT departments to manage the corporate persona on their employees' devices. Other commercial BYOD solutions include Samsung KNOX™[51]and VMware Horizon Mobile™[52]

Current technology in the field of IVI connectivity include MirrorLink™[53], which is a device interoperability standard for integration between smartphones and vehicle infotainment systems. In the MirrorLink™ architecture, applications are hosted and run on the smartphone while the driver and passengers interact with the applications via IVI system peripherals such as steering wheel controls, dashboard buttons or touch screens. MirrorLink™ utilizes well-established, non-proprietary technologies such as IP, USB, Wi-Fi, Bluetooth, Real-Time Protocol (RTP, for audio) and Universal Plug and Play (UPnP). The Virtual Network Computing (VNC) protocol is used to replicate the phone display in the car navigation display and communicate user input back to the mobile device.

The MirrorLink™ architecture is influenced by the state of current IVI systems, which are not able to match the functionality of contemporary smartphones. To-

[49]`https://github.com/Cellrox/devns-patches`

[50]`http://www.cellrox.com/`

[52]`http://www.vmware.com/mobile-secure-desktop/overview`

[52]`https://www.samsung.com/global/business/mobile/platform/mobile-platform/knox/`

[53]`http://www.mirrorlink.com/technology`

day, the GENIVI®[54]industry consortium drives the broad adoption of Linux-based operating systems, middleware and platforms for automotive IVI systems.

8 Conclusion

In this thesis we present the detailed design of a container-based isolation solution for Linux comprised of isolated dynamic security domains. Its defining features are the use of mandatory access control to strengthen the isolation provided by the resource namespaces. We also present a proof-of-concept prototype which utilizes dynamic isolated domains for application migrations which takes into account access control policies associated with the migrated application.

Our design shows that existing OS-level virtualization features in the Linux kernel can be used for dynamic application confinement. In an attempt to improve the deployability of the solution, we intentionally restricted ourselves to kernel features already available in contemporary Linux variants. Our evaluations identifies a number of gaps in the design, which are not easily addressed without appropriate kernel primitives not currently readily available. The major gaps in the compartmentalizations capabilities of Linux which are visible in our design are the lack of device and security namespaces. Both are topics of ongoing work by the Linux kernel community.

It is easy to draw parallels between the development of hardware virtualization technology in the 60s and the development of OS-level virtualization in Unix-like operating system. Currently there exists many systems that are capable of compartmentalizing a partial set of operating system resources. Despite its limitations Linux namespaces are one of the more mature solutions currently available.

[54]http://www.genivi.org

References

AA06 Adams, K. and Agesen, O., A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, New York, NY, USA, 2006, ACM, pages 2–13, URL `http://doi.acm.org/10.1145/1168857.1168860`.

ADH+11 Andrus, J., Dall, C., Hof, A. V., Laadan, O. and Nieh, J., Cells: a virtual mobile smartphone architecture. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, 2011, ACM, pages 173–187, URL `http://doi.acm.org/10.1145/2043556.2043574`.

BDR02 Bugnion, E., Devine, S. and Rosenblum, M., System and method for virtualizing computer systems, December 17 2002. URL `http://www.google.com/patents/US6496847`. US Patent 6,496,847.

Bie06 Biederman, E. W., Multiple instances of the global linux namespaces. *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, July 2006, pages 101–111.

Cor12 Corbet, J., A new approach to user namespaces, `http://lwn.net/Articles/491310/`, 2012. [Retrieved 11.09.2014].

Cre81 Creasy, R. J., The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25,5(1981), pages 483–490. URL `http://dx.doi.org/10.1147/rd.255.0483`.

DPR+13 Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D. and Wichs, D., Security analysis of pseudo-random number generators with input: /dev/random is not robust. *Proceedings of the 2013 ACM*

SIGSAC Conference on Computer &#38; Communications Security, CCS '13, New York, NY, USA, 2013, ACM, pages 647–658, URL http://doi.acm.org/10.1145/2508859.2516653.

FCH$^+$11 Felt, A. P., Chin, E., Hanna, S., Song, D. and Wagner, D., Android Permissions Demystified. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, 2011, ACM, pages 627–638, URL http://doi.acm.org/10.1145/2046707.2046779.

FEF$^+$12 Felt, A. P., Egelman, S., Finifter, M., Akhawe, D. and Wagner, D., How to ask for permission. *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, Berkeley, CA, USA, 2012, USENIX Association, pages 7–7, URL http://dl.acm.org/citation.cfm?id=2372387.2372394.

Ker10 Kerrisk, M., *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, first edition, 2010.

Ker13a Kerrisk, M., Namespaces in operation, part 1: namespaces overview, http://lwn.net/Articles/531114/, 2013. [Retrieved 05.10.2014].

Ker13b Kerrisk, M., Namespaces in operation, part 2: the namespaces api, http://lwn.net/Articles/531381/, 2013. [Retrieved 05.10.2014].

Ker13c Kerrisk, M., Namespaces in operation, part 3: Pid namespaces, http://lwn.net/Articles/531419/, 2013. [Retrieved 05.10.2014].

Ker13d Kerrisk, M., Namespaces in operation, part 4: more on pid namespaces, http://lwn.net/Articles/532748/, 2013. [Retrieved 05.10.2040].

Ker13e Kerrisk, M., Namespaces in operation, part 5: User namespaces, `http://lwn.net/Articles/532593/`, 2013. [Retrieved 05.10.2014].

Ker14 Kerrisk, M., Namespaces in operation, part 5: Network namespaces, `http://lwn.net/Articles/580893/`, 2014. [Retrieved 05.10.2014].

KREA11 Kostiainen, K., Reshetova, E., Ekberg, J.-E. and Asokan, N., Old, new, borrowed, blue –: A perspective on the evolution of mobile platform security architectures. *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, New York, NY, USA, 2011, ACM, pages 13–24, URL `http://doi.acm.org/10.1145/1943513.1943517`.

KW00 Kamp, P.-H. and Watson, R. N. M., Jails: Confining the omnipotent root. *In Proc. 2nd Intl. SANE Conference*, 2000.

Lev84 Levy, H. M., *Capability-Based Computer Systems.* Butterworth-Heinemann, Newton, MA, USA, 1984.

MH12 Mao, M. and Humphrey, M., A performance study on the vm startup time in the cloud. *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, Washington, DC, USA, 2012, IEEE Computer Society, pages 423–430, URL `http://dx.doi.org/10.1109/CLOUD.2012.103`.

MVH12 Miller, K., Voas, J. and Hurlburt, G., Byod: Security and privacy considerations. *IT Professional*, 14,5(2012), pages 53–55.

Olb78 Olbert, A. G., Extended control program support: Vm/370: A hardware assist for the ibm virtual machine facility/370. *SIGMICRO Newsl.*, 9,3(1978), pages 8–25. URL `http://doi.acm.org/10.1145/1096532.1096534`.

OSSN02 Osman, S., Subhraveti, D., Su, G. and Nieh, J., The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36,SI(2002), pages 361–376. URL `http://doi.acm.org/10.1145/844128.844162`.

PG74 Popek, G. J. and Goldberg, R. P., Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17,7(1974), pages 412–421. URL `http://doi.acm.org/10.1145/361011.361073`.

PPT+92 Pike, R., Presotto, D., Thompson, K., rickey, H. and Winterbottom, P., The use of name spaces in plan 9. *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, New York, NY 10036, USA, 1992, ACM Press, pages 1–5, URL `http://doi.acm.org/10.1145/506378.506413`.

PPTT90 Pike, R., Presotto, D., Thompson, K. and Trickey, H., Plan 9 from bell labs. *In Proceedings of the Summer 1990 UKUUG Conference*, 1990, pages 1–9.

RKNA14 Reshetova, E., Karhunen, J., Nyman, T. and Asokan, N., Security of operating system virtualization technologies. To appear in Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings. Springer, 2014.

Sha09 Shaufler, C., SMACK and the Application Ecosystem, September 2009. URL `http://www.linuxfoundation.jp/news-media/videos/2009/10/lpc-2009-smack-and-application-ecosystem`. Remarks by Casey Shaufler at the Linux Plumbers Conference, Portland, OR [Retrieved: 02.10.2014].

SS75 Saltzer, J. H. and Schroeder, M. D., The protection of information in computer systems. *Proceedings of the IEEE.*

SSL+99 Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. and Lepreau, J., The flask security architecture: System support for diverse security policies. *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, Berkeley, CA, USA, 1999, USENIX Association, pages 11–11, URL `http://dl.acm.org/citation.cfm?id=1251421.1251432`.

SVA09 Suomalainen, J., Valkonen, J. and Asokan, N., Standards for security associations in personal networks: a comparative analysis. *IJSN*, 4,1/2(2009), pages 87–100. URL `http://dx.doi.org/10.1504/IJSN.2009.023428`.

TMO+12 Tiwari, M., Mohan, P., Osheroff, A., Alkaff, H., Shi, E., Love, E., Song, D. and Asanović, K., Context-centric security. *Proceedings of the 7th USENIX conference on Hot Topics in Security*, HotSec'12, Berkeley, CA, USA, 2012, USENIX Association, pages 9–9, URL `http://dl.acm.org/citation.cfm?id=2372387.2372396`.

Var97 Varian, M., Vm and the vm community: Past, present, and future. *HARE 89 Sessions 9059–61.*