A Comparison of Secure Multi-tenancy Architectures for Filesystem Storage Clouds

Anil Kurmus[1], Moitrayee Gupta*[2], Roman Pletka[1], Christian Cachin[1], and Robert Haas[1]

[1] IBM Research - Zurich
{kur,rap,cca,rha}@zurich.ibm.com
[2] Department of Computer Science and Engineering, UCSD
m5gupta@cs.ucsd.edu

Abstract. A filesystem-level storage cloud offers network-filesystem access to multiple customers at low cost over the Internet. In this paper, we investigate two alternative architectures for achieving multi-tenancy securely and efficiently in such storage cloud services. They isolate customers in virtual machines at the hypervisor level and through mandatory access-control checks in one shared operating-system kernel, respectively. We compare and discuss the practical security guarantees of these architectures. We have implemented both approaches and compare them using performance measurements we obtained.

1 Introduction

Storage cloud services allow the sharing of storage infrastructure among multiple customers and hence significantly reduce costs. Typically, such services provide object or filesystem access over a network to the shared distributed infrastructure. To support multiple customers or *tenants* concurrently, the network-filesystem-access services must be properly isolated with minimal performance impact.

We consider here a *filesystem storage cloud* as a public cloud storage service used by customers to mount their own filesystems remotely through well-established network filesystem protocols such as NFS and the Common Internet Filesystem (CIFS, also known as *SMB*). Such a service constitutes a highly scalable, performant, and reliable enterprise network-attached storage (NAS) accessible over the Internet that provides services to multiple tenants.

In general, a cloud service can be run at any of the following increasing levels of multi-tenancy:

- *Hardware level*: server hardware, OS, and application dedicated per client.
- *Hypervisor level*: share server hardware, and use virtualization to host dedicated OS and application per client.
- *OS level*: share server hardware and OS, and run a dedicated application per client.
- *Application level*: share server hardware, OS, and application server among clients.

* Work done at IBM Research - Zurich.

Intuitively, the higher the level of multi-tenancy, the easier it seems to achieve a resource-efficient design and implementation; at the same time, though, it gets harder (conceptually and in terms of development effort) to securely isolate the clients from each other.

In this paper, we investigate a *hypervisor-level* and an *OS-level* multi-tenant filesystem storage cloud architecture, and compare them in terms of performance and security. The hypervisor-level multi-tenancy approach is based on hardware virtualization (with para-virtualized drivers for improved networking performance). We refer to this architecture as the *virtualization-based multi-tenancy (VMT) architecture*. The OS-level multi-tenancy approach uses mandatory access control (MAC) in the Linux kernel and is capable of isolating customer-dedicated user-space services on the same OS. Such an architecture may also leverage, for instance, OS-level virtualization technologies such as *OpenVZ* or *Linux Containers (LXC)*. We refer to this architecture as the *operating-system-based multi-tenancy (OSMT) architecture* in the remainder of this paper.

We have implemented both approaches on real hardware in the *IBM Scale-out NAS (SONAS)* [1] and the *IBM General Parallel Filesystem (GPFS)* [2] technologies. We used open-source components such as *KVM* [3] with *virtio* networking for virtualization and *SELinux* (`http://selinuxproject.org/`) for MAC.

Section 3 describes the architecture of a filesystem storage cloud and introduces the two designs. Section 4 defines an adversary model and discusses the security of both architectures according to this model. Section 5 presents the implementation and benchmark results. Related work is discussed in Section 6.

2 Background

One can distinguish the following categories of general-purpose storage clouds (ignoring storage clouds that provide database-like structures on content):

- *Block storage clouds*, with a block-level interface, i.e., an interface that allows the writing and reading of fixed-sized blocks. Examples of such clouds include *Amazon EBS*.
- *Object storage clouds*, composed of buckets (or containers) that contain objects (or blobs). These objects are referred to by a key (or name). The API is usually very simple: typically a REST API with *create* and *remove* operations on buckets and *put*, *get*, *delete*, and *list* operations on objects. Example of such storage clouds include *Amazon S3*, *Rackspace Cloudfiles*, and *Azure Storage Blobs*.
- *Filesystem storage clouds*, with a full-fledged filesystem interface, therefore referred to also as "cloud NAS." Examples of such clouds include *Nirvanix CloudNAS*, *Azure Drive*, and *IBM Scale-Out Network Attached Storage (SONAS)*.

Application-level multi-tenancy is sometimes also referred to as native multi-tenancy. Some authors consider it the cleanest way to isolate multiple tenants [4]. However, achieving multi-tenancy securely is very challenging and therefore not common for filesystem storage clouds. The reasons lie in the complex nature of this task: unlike other types of storage clouds, filesystem storage clouds possess complex APIs that have evolved over time, which leads to large attack surfaces. The vulnerability track record of these applications seems to confirm this intuition. CIFS servers were vulnerable

to various buffer-overflows (e.g., CVE-2010-3069, CVE-2010-2063, CVE-2007-2446, CVE-2003-0085, CVE-2002-1318, see `http://cve.mitre.org/`), format string vulnerability leading to arbitrary code execution (CVE-2009-1886), directory traversals (CVE-2010-0926, CVE-2001-1162), while NFS servers were also vulnerable to similar classic vulnerabilities as well as more specific ones such as filehandle vulnerabilities [5]. Moreover, adding multi-tenancy support into these server applications would require significant development (e.g., in order to distinguish between different authentication servers for specific filesystem exports) which will most likely result in new vulnerabilities. We discuss in Sections 3 and 4 architectures with lower levels of multi-tenancy. They effectively restrict the impact of arbitrary code execution vulnerabilities to the realm of a single tenant: by definition, this cannot be achieved with application-level multi-tenancy.

This paper targets the IBM SONAS [1] platform, which evolved from the IBM Scale-Out File Services (SoFS) [6]. IBM SONAS provides a highly scalable network-attached storage service, and therefore serves as a typical example of a filesystem storage cloud. IBM SONAS currently contains support for hardware-level multi-tenancy according to the architectures discussed in this work. Adding a higher-level of multi-tenancy is an important step to reduce the cost of a cloud-service provider.

3 System Description

Section 3.1 gives an overview of the general architecture of a filesystem storage cloud. Section 3.2 describes the MAC policies which are used in both architectures. Sections 3.3 and 3.4 introduce the two alternatives, detailing the internals of the interface nodes, the key element of the filesystem storage cloud architecture.

3.1 General Description

Figure 1 depicts the general architecture of a filesystem storage cloud that consists of the following elements:

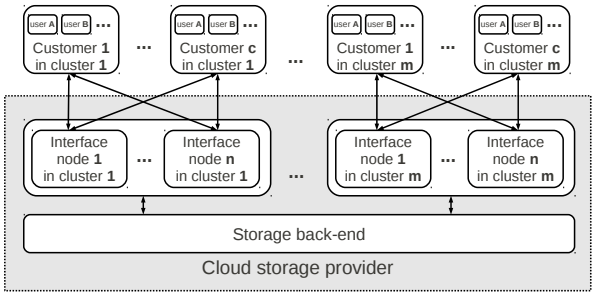


Fig. 1. General architecture of a filesystem storage cloud.

- Customers and users: A *customer* is an entity (e.g., a company) that uses at least one network file system. A customer can have multiple individual *users*. We assume that multiple customers connect to the filesystem storage cloud and that each customer has a separate set of users. Data is separated between users from distinct customers, and user IDs are in a separate namespace for each customer. Hence two distinct customers may allocate the same user ID without any conflict on the interface nodes or in the storage back-end.
- Interface nodes and cluster: An interface node is a system running *filer services* such as NFS or CIFS daemons. Interface nodes administratively and physically belong to the cloud service provider and serve multiple customers. A customer connects to the filesystem storage cloud through the interface nodes and mounts its filesystems over the Internet. Multiple interface nodes together form an interface cluster, and one interface node may serve multiple customers. A customer connects only to nodes in one interface cluster.
- Shared back-end storage: The shared back-end storage provides block-level storage for user data. It is accessible from the interface clusters over a network using a distributed filesystem such as GPFS [2]. It is designed to be reliable, highly available, and performant. We assume that no security mechanism exists within the distributed filesystem to authenticate and authorize nodes of the cluster internally.
- Customer boarding, unboarding, and configuration: Typically, interface nodes must be created, configured, started, stopped, or removed when customers are boarded (added to the service) or unboarded (removed from the service). This is performed by administration nodes not shown here, which register customer accounts and configure filesystems. Ideally, boarding and unboarding should consume a minimal amount of system resources and time.

As an example, a customer registers a filesystem with a given size from the filesystem storage cloud provider, and then configures machines on the customer site that mount this filesystem. The users of the customer can then use the cloud filesystem similar to how they use a local filesystem. Customers connect to the interface cluster via a dedicated physical wide-area network link or via a dedicated VPN over the Internet, ideally with low latency. The cloud provider may limit the maximal bandwidth on a customer link.

To ensure high availability and high throughput, a customer accesses the storage cloud through the clustered interface nodes. Interface nodes have to perform synchronization tasks within their cluster and with the back-end storage, generating additional traffic. An interface node has three network interfaces: one to the customer, one to other nodes in the cluster, and one to the back-end storage.

Dimensioning. The size of a filesystem storage cloud is determined by the following parameters, which are derived from service-level agreements and from the (expected or observed) load in the system: the number of customers c assigned to an interface cluster, the number of interface nodes n in a cluster (due to synchronization overhead, this incurs a trade-off between higher availability and better performance), and the number of clusters m attached to the same storage back-end.

Customer and user authentication. Describing customer authentication would exceed the scope of this work; in practice, it can be delegated to the customer's VPN endpoint in the

premises of the service-cloud provider. The authentication of users from a given customer also requires that customers provide a directory service that will serve authentication requests made by users. Such a directory service can be physically located on the customer's premises and under its administration or as separate service in the cloud. In either case, users authenticate to an interface node, which in turn relays such requests to the authentication service of the customer.

3.2 Mandatory Access Control Policies

We use mandatory access control on the filer services. In case of their compromise, MAC provides a first layer of defense on both architectures. For practical reasons, we have used SELinux. Other popular choices include *grsecurity RBAC* [7] or *TOMOYO*. These MAC systems limit the privileges of the filer services to those required, effectively creating a sandbox environment, by enforcing policies that are essentially a list of permitted operations (e.g., open certain files, bind certain ports, fork, ...).

As an example, the policies on the interface nodes basically permit the filer services to perform the following operations: bind on their listening port and accept connections, perform all filesystem operations on the customer's data directory (which resides on a distributed filesystem), append to the filer log files, and read the relevant service configuration files.

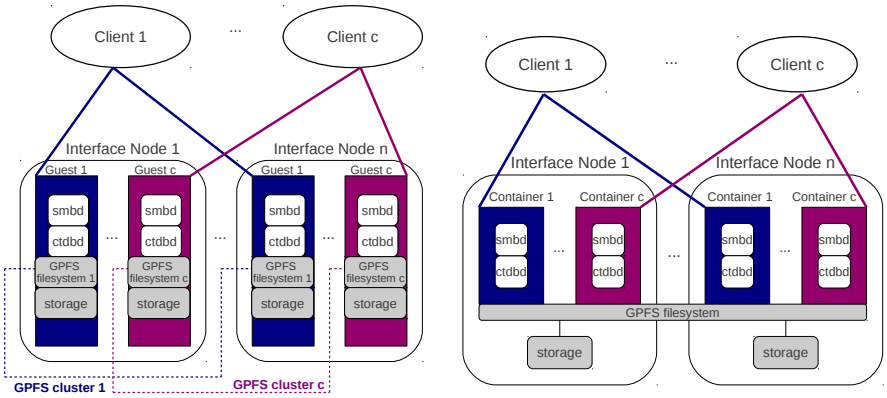
The protection provided by these policies can be defeated in two ways. One possibility is if the attacker manages to execute arbitrary code in kernel context (e.g., through a local kernel exploit), in which case it is trivial to disable any protections provided by the kernel, including MAC. The second possibility is by exploiting a hole in the SELinux policy, which is unlikely, but would be the case, for example, if a filer service were authorized to load a kernel module.

An important example of the benefit of these policies is the restriction of accessible network interfaces to the customer and intra-cluster network interfaces only. Another example is the impossibility for processes running in the security context of the filer services to write to files they can execute, or to use `mmap()` and `mprotect()` to get around this restriction. In practice, this means, for example, that an attacker making use of a remote exploit on a filer service cannot just obtain a shell and download and execute a local kernel exploit: the attacker would have to find a way to execute the latter exploit directly within the first exploit, which, depending on the specifics of the vulnerabilities and memory protection features, can be impossible.

Note that, because of the way MAC policies are specified — that is, by white-listing the permitted operations — these examples (network interface access denied, no read and execute permission) are a consequence of the policy and do not have to be explicitly specified, which encourages policies to be built according to the least privilege principle.

3.3 VMT Architecture

We now introduce the first architecture, called the *virtualization-based multi-tenancy (VMT) architecture*. It is based on *KVM* as a hypervisor and implements multi-tenancy by running multiple virtual interface nodes as guests on the hardware of one physical interface node. Such a filesystem storage cloud has a fixed number of physical interface



(a) VMT architecture (b) OSMT architecture

Fig. 2. Two architectures for multi-tenancy, shown for one interface cluster of each architecture.

nodes in every cluster, with each interface node running one guest for each customer. All guests that belong to the same customer form an interface-node cluster, which maintains the distributed filesystem with the data of the customer (labeled *GPFS cluster* in Figure 2(a), as explained below). Each virtual machine (VM) runs one instance of the required filer-service daemons, exporting the filesystems through the CIFS or NFS protocols, and has three separate network interfaces.

In terms of isolation, MAC can be applied at two levels in this architecture. The first level is inside a guest, for protecting filer services exposed to the external attackers, using the exact same policies as in the OSMT architecture[3] described below. The second level is on the host, with the idea of sandboxing guests (i.e., QEMU processes running on the host, in the case of KVM) by using multi-category security. Policies at this level do not depend on what is running inside the guests, therefore they can be applied to many virtualization scenarios. Such policies already exist and are implemented by sVirt (see `http://selinuxproject.org/page/SVirt`).

Figure 2(a) shows one of m clusters according to the VMT architecture, in which the distributed filesystem is GPFS and the daemons in each virtual machine are `smbd` (the Samba daemon, for CIFS) and `ctdbd` (the clustered trivial database daemon, used to synchronize meta-data among cluster nodes). They work together to export customer data using the CIFS protocol. The customers are also shown and connect only to their dedicated VM on each interface node. In terms of the dimension parameters from Figure 1, for every one of the m interface clusters, there are c *GPFS clusters*, each corresponding to a GPFS filesystem, and $c \cdot n$ guest virtual machines (n per customer) in this architecture.

When a new customer is boarded, it is assigned to a cluster and a configuration script automatically starts additional guests for that customer on all the physical interface nodes

[3] Except for the use of the multi-category security functionality (see Section 3.4): categories are not required in the VMT architecture as only one customer resides in each guest.

within this cluster. Furthermore, a new GPFS filesystem and cluster are created for the customer on the new virtual guests. Customer data can then be copied to the filesystem and accessed by users.

3.4 OSMT Architecture

The second architecture, called the *operating-system-based multi-tenancy (OSMT) architecture*, is based on a lightweight separation of OS resources by the kernel. OS-level virtualization of this form can be achieved using *containers*, such as *LXC*, *OpenVZ* or *Zap pods* [8] for Linux, *jails* [9] for FreeBSD, and *zones* [10] for Solaris. Containers do not virtualize the underlying hardware, and thus cannot run multiple different OSes, but create new isolated namespaces for some OS resources such as filesystems, network interfaces, and processes. Processes running within a container are isolated from processes within other containers, thus they seem to be running in a separate OS. All processes share the same kernel, hence, one cannot encapsulate applications that rely on kernel threads in containers (such as the kernel NFS daemon).

In our implementation, isolation is performed using *SELinux multi-category-security (MCS)* policies [11] for shielding the processes that serve a particular customer from all others. It is then sufficient to write a single policy for each filer service that applies for all customers by simply adjusting the category of each filer service and other related components inside a container (e.g., configuration files, customer data). This ensures that no two distinct customers can access each other's resources (because they belong to different categories). In comparison to the VMT architecture, the policies in the guest that contain the filer services, and the policies in the host that isolate the customers are now combined into a single policy which achieves the same goals.

In addition, a *change-root (chroot)* environment is installed, whose only purpose is to simplify the configuration of the isolated services and the file labeling for SELinux. We refer to such a customer isolation domain as a *container* in the remainder of this work. A container dedicated to one customer on an interface node consists of a *chroot directory* in the root filesystem of the interface node, which contains all files required to access the filesystem for that customer. All required daemons accessed by the customer run within the container. Because of the chroot environment, the default path names, all configuration files, the logfile locations, and so on, are all the same or found at the same locations for every customer; this is implemented through read-only mount binds, without having to copy files or create hard links. This approach makes our container-based setup amenable to automatic maintenance through existing software distribution and packaging tools.

This form of isolation does not provide new namespaces for some critical kernel resources (process identifiers are global, for instance); it does not allow a limitation of memory and CPU usage either. However, it causes a smaller overhead for isolation than hypervisor-based virtualization does.

Figure 2(b) shows an interface cluster following the OSMT architecture, in which the distributed filesystem is GPFS, shared by all containers within a cluster. Each container runs a single instance of each of the `smbd` and `ctdbd` daemons, accessed only by the corresponding customer. In the terms of the dimension parameters from Figure 1, for every one of the m interface clusters, there exists *one* GPFS filesystem (only one per

cluster), n kernels (each kernel is shared by c customers), and $c \cdot n$ containers (n per customer).

Customer boarding is done by a script that creates an additional container on every interface node in the cluster, and a data directory for that customer in the shared distributed filesystem of the interface cluster. The daemons running inside the new containers must be configured to export the customer's data directory using the protocols selected. No changes have to be made to the configuration of the distributed filesystem on the interface nodes.

4 Security Comparison

In this section, we discuss the differences between the VMT architecture and the OSMT architecture from a security viewpoint. Because we aim to compare the two approaches, we only briefly touch on those security aspects that are equal for the two architectures. This concerns, for instance, user authentication, attacks from one user of a customer against other users of the same customer, and attacks by the service provider ("insider attacks" from its administrators). These aspects generally depend on the network filesystem and the user-authentication method chosen, as well as their implementations. They critically affect the security of the overall solution, but are not considered further here.

4.1 Security Model

We consider only attacks by a malicious customer, i.e., attacks mounted from a user assigned to one customer against the service provider or against other customers. In accordance with the traditional goals of information security, we can distinguish three types of attacks: those compromising the confidentiality, the integrity, or the availability of the service and/or of data from other customers.

Below we group attacks in two categories. First we discuss *denial-of-service (DoS) attacks* targeting service availability in Section 4.3. Second, we subsume threats against the confidentiality and integrity of data under *unauthorized data access* and discuss them in Section 4.4.

We assume that the cloud service provider is trusted by the customers. We also disregard customer-side cryptographic protection methods, such as filesystem encryption [12] and data-integrity protection [13]. These techniques would not only secure the customer's data against attacks from the provider but also protect its data from other customers. Such solutions can be implemented by the customer transparently to the service provider and generally come with their own cost (such as key management or the need for local trusted storage).

4.2 Comparison Method

An adversary may compromise a component of the system or the whole system with a certain *likelihood*, which depends on the vulnerability of the component and on properties of the adversary such as its determination, its skills, the resources it invests in an attack and so on. This likelihood is influenced by many factors, and we refrain from assigning

numerical values or probabilities to it, as it cannot be evaluated with any reasonable accuracy [14, Chap. 3–4].

Instead we group all attacks into three sets according to the likelihood that an attack is feasible with methods known today or the likelihood of discovering an exploitable vulnerability that immediately enables the attack. We roughly estimate the relative severity of attacks and vulnerabilities according to criteria widely accepted by computer emergency readiness teams (CERTs), such as previous exploits or their attack surfaces. Our three likelihood classes are described by the terms *unlikely*, *somewhat likely* and *likely*.

In Section 4.4 we model data compromise in the filesystem storage cloud through graphical *attack trees* [15]. They describe how an attacker can reach its goal over various paths; the graphs allow a visual comparison of the security of the architectures.

More precisely, an attack tree is a directed graph, whose nodes correspond to states of the system. The initial state is shown in white (meaning that the attacker obtains an account on the storage cloud) and the exit node is colored black (meaning that the attacker gained unauthorized access to another customer's data). A state designates a component of the system (as described in the architecture) together with an indication of the security violation the attacker could have achieved or of how the attacker could have reached this state.

An edge corresponds to an attack that could be exploited by an attacker to advance the state of compromise of the system. The intermediate nodes are shown in various shades of gray, roughly corresponding to the severity of the compromise. Every attack is labeled by a likelihood (unlikely, somewhat likely, or likely), represented by the type of arrow used.

4.3 Denial-of-Service Attacks

Server crashes. An attacker can exploit software bugs causing various components of an interface node to crash, such as the filer services (e.g., the NFS or CIFS daemon) or the OS kernel serving the customer. Such crashes are relatively easy to detect and the service provider can recover from them easily by restarting the component. Usually such an attack can also be attributed to a specific customer because the service provider maintains billing information of the customer; hence the offending customer can easily be banned from the system.

Both architectures involve running dedicated copies of the filer services for each customer. Therefore, crashing a filer service only affects the customer itself. Although the attack may appear likely in our terminology, we consider it not to be a threat because of the external recovery methods available.

Note that non-malicious faults or random crashes of components are not a concern because all components are replicated inside an interface cluster, which means that the service as a whole remains available. Crashes due to malicious attacks, on the other hand, will affect all nodes in a cluster as the attacker can repeat its attack.

Furthermore, any server crash has to be carried out remotely and therefore mainly affects the network stack. It appears much easier for a local user to crash a server, in contrast. For this, the attacker must previously obtain the privilege to execute code on the interface node, most likely through an exploit in one of the filer services. However,

when attackers have obtained a local account on an interface node, they can cause much more severe problems than simply causing a crash (Section 4.4). Therefore we consider a locally mounted DoS attack as an acceptable threat.

In the *VMT architecture*, a kernel crash that occurs only inside the virtual machine dedicated to the customer does not affect other customers, which run in other guests — at least according to the generally accepted view of virtual-machine security. However, the effects on other guests depend on the kind of the DoS attack. A network attack that exploits a vulnerability in the upper part of the network stack (e.g., UDP) most likely only crashes the targeted guest. But an attack on lower-layer components of the hypervisor (e.g., network interface driver), which run in the host, can crash the host and all guests at once. Moreover, additional vulnerabilities may be introduced through the hypervisor itself.

In the *OSMT architecture*, an attacker may crash the OS kernel (through a vulnerability in the network interface driver or a bug in the network stack), which results in the crash of the entire interface node and disables also the service to all other customers. Thus, the class of DoS attacks targeted against the OS kernel has a greater effect than in the VMT architecture.

Resource exhaustion. An attacker can try to submit many filesystem requests to exhaust some resource, such as the network between the customers and the interface nodes, the network between interface nodes and the resource cluster, or the available CPU and memory resources on the interface nodes. Network-resource exhaustion attacks affect both our designs in the same way (and more generally, are a common problem in most Internet services); therefore, we do not consider them further and discuss only the exhaustion of host resources.

In the *VMT architecture*, hypervisors can impose a memory bound on a guest OS and limit the number of CPUs that a guest can use. For example, a six-CPU interface node may be shared by six customers in our setup. Limiting every guest to two CPUs means that the interface node still tolerates two malicious customers that utilize all computation power of their dedicated guests, but continues to serve the other four customers with two CPUs.

The impact of a resource-exhaustion attack with a container setup in the *OSMT architecture* depends on the container technology used and its configuration.

In our study, we use a container technology (SELinux and chroot environment) that cannot restrict the CPU used or the memory consumed by a particular customer. Given proper dimensioning of the available CPU and memory resources with respect to the expected maximal load per customer, however, a fair resource scheduler alone can be sufficient to render such attacks harmless.

With more advanced container technology, such as LXC (based on the recent cgroup process grouping feature of the Linux kernel), it is possible to impose fine-grained restrictions on these resources, analogously to a hypervisor. For instance, the number of CPUs attributed to a customer and the maximally used CPU percentage can be limited for every customer.

4.4 Unauthorized Data Access

We describe here the attack graphs in Figures 3 and 4 as explained in Section 4.2. Some attacks are common and apply to both architectures; they are described first. We then present specific attacks against the VMT and OSMT architectures. Each attack graph includes all attacks relevant for the architecture.

Common attacks. *Filer service compromise.* Various memory corruption vulnerabilities (such as buffer overflows, string format vulnerabilities, double frees) are notorious for allowing attackers to execute arbitrary code with the privileges of the filer service. However, protection measures such as address space layout randomization, non-executable pages, position-independent executables, and stack canaries, can render many attacks impossible without additional vulnerabilities (e.g., information leaks). This is especially true for remote attacks, in which the attacker has very little information (e.g., no access to `/proc/pid/`) and less control over memory contents (e.g., no possibility of attacker-supplied environment variables) than for local attacks. For these reasons, we categorize these attacks as "somewhat likely."

Complementing the aforementioned attacks that permit arbitrary code execution, confused deputy attacks [16] form a weaker class of attacks. In such an attack, the attacker lures the target application into performing an operation unauthorized to the attacker without obtaining arbitrary code execution. Directory traversal, whereby an attacker tricks the filer service into serving files from a directory that should not be accessible, is a famous example of such attacks in the context of storage services (e.g., CVE-2010-0926, CVE-2001-1162 for CIFS). Clearly, such attacks leverage the privileges of the target process: a process that has restricted privileges is not vulnerable. Therefore, they form a weaker class of attacks: preventing unauthorized data access to an attacker who has compromised the filer service through arbitrary code execution also prevents these attacks. Furthermore, confused deputy attacks are very unlikely to serve as a stepping stone for a second attack (e.g., accessing the internal network interface), which would be required to access another tenant's data in both architectures here. Consequently, we do not consider confused deputy attacks any further.

Kernel compromise. We distinguish between remote and local kernel attacks. The reasoning in the previous paragraph concerning the lack of information and memory control is essentially also valid for remote kernel exploits. However, for the kernel, the attack surface is much more restricted: typically network drivers and protocols, and usually under restrictive conditions (e.g., LAN access). Recently, Wi-Fi drivers have been found to be vulnerable (CVE-2008-4395), as well as the SCTP protocol (CVE-2009-0065) both of which would not be used in the context of a filesystem storage cloud. For these reasons, we categorize these attacks as "unlikely." In contrast to remote exploits, we categorize local kernel exploits as "somewhat likely" given the information advantage (e.g., `/proc/kallsysms`) and capabilities of a local attacker (e.g., mapping a fixed memory location). Many recently discovered local kernel vulnerabilities confirm this view.

SELinux bypass. The protection provided by SELinux can be bypassed in two ways. One of them is by leveraging a mistake in the security policy written for the application: if the policy is too permissive, the attacker can find ways to get around some restrictions. An

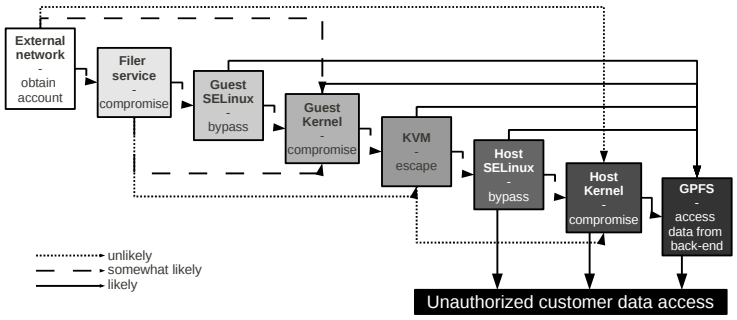


Fig. 3. Attack graph for the VMT architecture.

example of such a policy vulnerability was found in sVirt [17]: an excessively permissive rule in the policy allowed an attacker in the hypervisor context to write directly to the physical drive, which the attacker can leverage in many ways to elevate his privileges. The second option for bypassing SELinux is by leveraging a SELinux implementation bug in the kernel. An example of such a vulnerability is the bypass of NULL pointer dereference protections. The Linux kernel performs checks when performing `mmap()` to prevent a user from mapping addresses lower than `mmap_min_addr` (which is required for exploiting kernel NULL pointer dereferences vulnerabilities). SELinux also implemented such a protection (with the additional possibility of allowing such an operation for some trusted applications). However, the SELinux access control decision in the kernel would basically override the `mmap_min_addr` check, weakening the security of the default kernel (CVE-2009-2695). For these reasons, we categorize these attacks as "somewhat likely."

Attacks against the VMT architecture. *VM escapes.* Although virtual machines are often marketed as the ultimate security isolation tool, it has been shown [18, 19] that many existing hypervisors contain vulnerabilities that can be exploited to escape from a guest machine to the host. We assume these attacks are "somewhat likely".

Filer service compromise: NFS daemon and SELinux. Apart from the helper daemons, which represent a small part of the overall code (e.g., `rpc.mountd`, `rpc.statd`, `portmapd`), most of the `nfsd` code is in kernel-space. This means it is not possible to restrict the privileges of this code with a MAC mechanism in the sense that a vulnerability in this code might directly lead to arbitrary code execution in kernel mode. The authors of [20] tried to implement such a protection within the kernel but this approach cannot guarantee sufficient isolation of kernel code simply because an attacker with `ring 0` privileges can disable SELinux. We categorize this attack as "somewhat likely."

Attacks against the OSMT architecture. *Container escapes.* As mentioned in 3.4, we have implemented what we refer to as containers using a chroot environment. As is widely known, a chroot environment does not effectively prevent an attacker from escaping

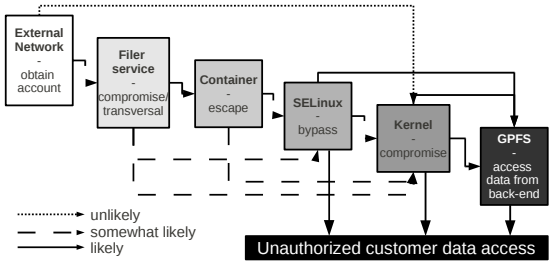


Fig. 4. Attack graph for the OSMT architecture.

from the environment and provides limited isolation. For completeness, we include a container-protection layer which corresponds to the chroot environment (without SELinux) in Figure 4, and marked it as "likely" to be defeated. However, containers such as LXC do implement better containment using the *cgroups* feature of Linux. While these technologies have a clean and simple design, it is still likely that some vulnerabilities allowing escapes can be found, especially because they are very recent (one such current concern regards containers mounting a `/proc` pseudo-filesystem).

4.5 Conclusion

A high-level comparison of Figures 3 and 4 shows that the VMT architecture has many more layers and could lead to the conclusion that the VMT approach provides better security. However, we also have to take into account the various attacks: most notably, it is possible that an attacker uses the internal network interface effectively for customer data access, and that this network interface is accessible from within the guest VMs (which is required, because the distributed filesystem service runs in the guest). The possibility of this attack renders other layers of protection due to VM isolation much less useful in the VMT architecture.

In other words, a likely chain of compromises that can occur for each scenario is

- for VMT:
  1. attacker compromises filer service, obtaining local unprivileged access,[4]
  2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,
  3. attacker accesses files of a customer through the distributed filesystem (assuming no authentication or authorization of nodes and no access control on blocks).
- for OSMT:
  1. attacker compromises filer service, obtaining local unprivileged access,
  2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,

[4] In the case of a kernel NFS daemon, it is possible that the attacker directly obtains `ring 0` privileges and can therefore skip the next step, however this is less likely.

3. attacker accesses files of a local co-tenant, or through the distributed filesystem for other customers.

Although it is expected that hypervisor-level multi-tenancy can, in general, be a better security design than OS-level multi-tenancy, we have seen in this section that in the case of a filesystem storage cloud and under our assumptions (i.e., that in a distributed filesystem, each individual node is trusted), no solution was clearly more secure than the other. Both solutions could be used to achieve an acceptable level of security.

5 Performance and Scalability Evaluation

In this section, we present a performance evaluation comparing the VMT and OSMT architectures for given customer workloads. Our experimental filesystem storage cloud setup is based on the IBM SONAS [1] product, on which we implemented the two multi-tenant architectures in the interface nodes, using the same physical infrastructure. We ran the benchmarks on both setups using the same customer workload based on the CIFS protocol, and measured various system metrics. Besides measuring the performance of the two architectures, the evaluation allows us to compare the scalability of the architectures.

In this section, we use the term *client* to refer to a single user belonging to a customer (each customer has one user, and we refer to it as client).

5.1 Experimental setup

We experiment with two storage back-end configurations in our benchmark.

1. RAM-disks directly on the interface node. This allows us to observe the performance of the systems in the absence of bottlenecks due to physical disk-access limitations, which is useful for analyzing the scalability of the interface node itself.
2. Actual disk-based storage, in the form of a direct-attached DS3200 storage subsystem, which allows us to evaluate performance in a realistic setup.

The experimental setup consists of two interface nodes forming a two-node GPFS cluster, and one client node. All network connections are 1 GbE links. To measure the performance of a single interface node, we connect the client node to only one of the interface nodes. Thus, all client traffic goes to a single interface node over a single 1 GbE link. Although the second interface node receives no direct client traffic, it is included in the benchmark setup to have a realistic 2-node GPFS cluster setup.

Figure 5 shows the setup and the server configurations. All servers run RHEL Server 5.5 with kernel version 2.6.18-194. The IBM SONAS version used on the interface nodes is 1.5.3-20. On the DS3200 storage subsystem, we use a single 5+1 RAID5 array with a total capacity of 2 TB using 15k RPM SAS drives. The storage subsystem is attached to only one of the interface nodes — the same node that is connected to the client.

In the VMT setup, we use *KVM* and *libvirt* [21] to create and manage virtual machines on the interface nodes. Each interface node has one virtual machine for each customer, and all the virtual machines belonging to a specific customer across all the interface

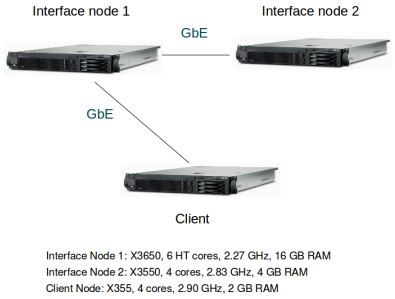


Fig. 5. Experimental setup.

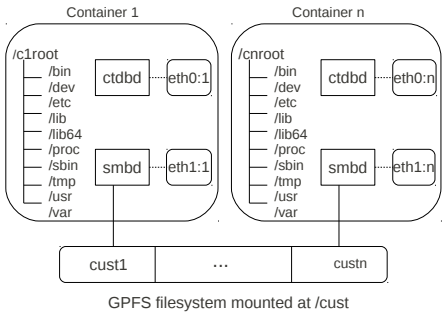


Fig. 6. Setup of containers in an interface node.

nodes are clustered together, with a single filesystem containing that customer's data. Each virtual machine has two virtual Ethernet interfaces assigned, which are bridged to the corresponding interfaces on the host; one interface is used for GPFS-cluster synchronization traffic, and the other for client traffic. In the RAM-disk configuration, the total RAM-disk allocation on each interface node is evenly divided between all the virtual machines running on that node. Each virtual machine uses a single RAM-disk for the creation of the distributed filesystem. We make sure that the total physical memory allocation in the VMT setup is exactly the same as in the OSMT setup. In the disk-based configuration, the 2 TB SAS array is evenly divided into partitions, which are then exposed to the virtual machines as raw devices through virtio interfaces. Each virtual machine uses the raw device for the creation of the distributed filesystem.

The OSMT setup on the two interface nodes consists of dedicated "chroot directories" for each container in the root filesystems of the interface nodes. Each container is assigned two aliased Ethernet interfaces: one for GPFS-cluster synchronization traffic and the other for client traffic. All the network interfaces assigned to the containers are created as aliases of the host interfaces. We use aliases to simulate an environment in which each customer has a dedicated secure access channel to the interface cluster. In an actual customer environment, this secure channel would take the form of a VPN. Figure 6 shows the OSMT setup for a single interface node. The interface node shown here is part of a GPFS cluster. The GPFS filesystem extends to the other interface nodes that are part of the same cluster. For the RAM-disk configuration, a single RAM disk is used on each interface node and a single GPFS filesystem is created using the RAM disks from both interface nodes. This filesystem is used by all the containers, with specific data directories allotted to each customer. For the disk-based configuration, the entire disk array is used to create a single filesystem, which is then used in the same way as in the RAM-disk setup.

5.2 Tools Used in the Benchmarks

The standard fileserver benchmark used widely to evaluate the performance of Windows fileservers is *Netbench*. However, Netbench runs only on the Windows platform and

requires substantial hardware for a complete benchmark setup. Under Linux, the Samba software suite provides two tools that can be used to benchmark SMB servers using Netbench-style I/O traffic, namely *dbench* and *smbtorture BENCH-NBENCH*. Both tools read a loadfile provided as input to perform the corresponding I/O operations against the fileserver being benchmarked. The dbench tool can be run against both NFS and SMB fileservers, the smbtorture tool is specific to SMB fileservers. We used the smbtorture BENCH-NBENCH tool because it offers more control over various aspects of the benchmark runs than the dbench tool, and also because our benchmark focuses solely on client access using the CIFS protocol.

The loadfile used in our benchmark consisted mainly of file creation operations, 4 KiB reads, 64 KiB writes, and some metadata queries.

To collect system metrics from the interface and client nodes during the execution of the benchmark, we used the *System Activity Report (SAR)* tool from the Red Hat *sysstat* package. The SAR tool collects, reports and saves system activity information, such as CPU and memory usage, load, network and disk usage, by reading the kernel counters at specified intervals.

5.3 Benchmark Procedure

We collected and analyzed the following system metrics on the interface nodes (in the case of the VMT architecture, metrics were collected on the host):

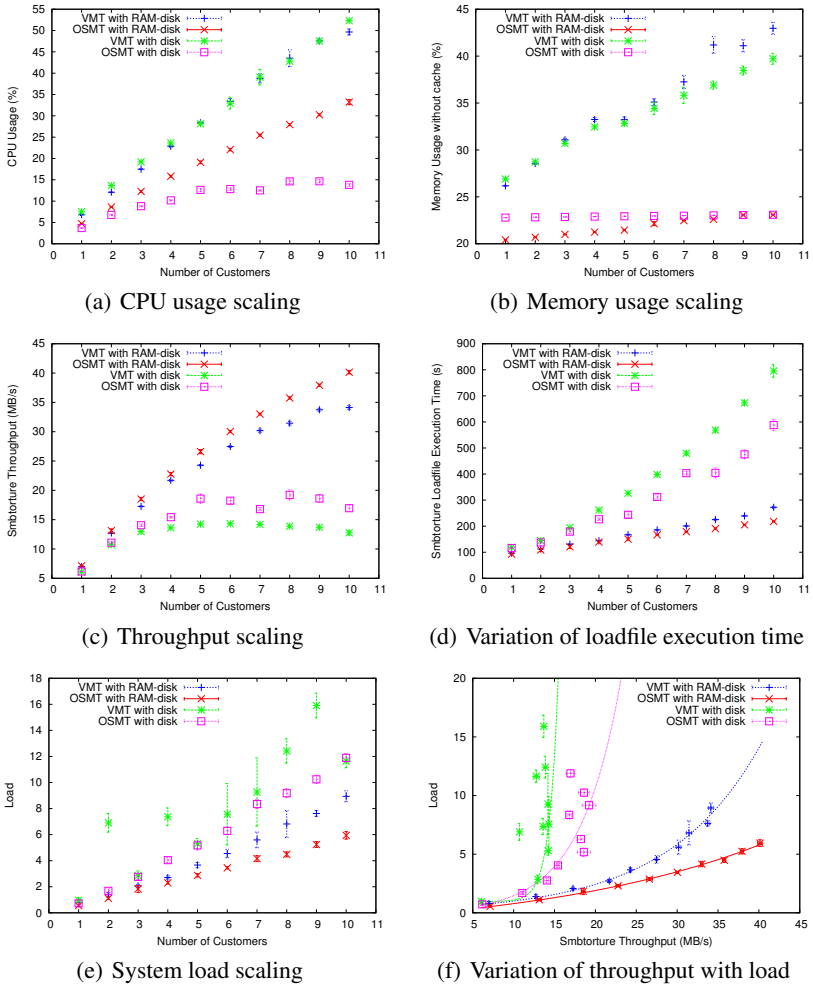
1. *CPU usage*: We used the *%idle* values reported by SAR to compute the *%used* values in each interval.
2. *System load*: We used the one-minute system load averages reported by SAR. Note that on Linux systems, the load values reported also take into account processes that are blocked waiting on disk I/O.
3. *Memory usage*: We recorded the memory usage both with and without buffers. In both cases, we excluded the cached memory. In the VMT setups, we also excluded the cached memory on the virtual machines running on that host.

In addition to these system metrics, we also recorded the throughput and loadfile execution time reported by smbtorture on the clients. We performed 10 iterations for each benchmark run — caches were preconditioned by 2 dry runs. We then computed 95% confidence intervals using the t-distribution for each metric measured.

5.4 Results

The graphs in this section show the variation of a particular system metric on both the VMT and the OSMT architecture. For each architecture, we show the variation of the metric on the RAM-disk-based setup as well as the disk-based setup. Note that each customer is simulated with a single user (one client).

Figure 7(a) shows the variation of CPU usage as a function of the number of customers. The overall CPU usage is much lower in the OSMT architecture, for both the RAM-disk and the disk setup. In the OSMT architecture, the CPU usage is significantly lower when we use disk storage and flattens above 5 customers because more cycles are



(a) CPU usage scaling (b) Memory usage scaling

(c) Throughput scaling (d) Variation of loadfile execution time

(e) System load scaling (f) Variation of throughput with load

Fig. 7. Benchmark results

spent waiting on disk I/O than in the RAM-disk-based setup. Hence, the performance of the disks dominates the overall throughput. For the VMT architecture, however, the CPU usage is about the same in both setups and therefore starts impacting the overall throughput when the number of customers is higher. For 10 customers, this culminates in a difference of about 77% in CPU usage between the two architectures when disk storage is used.

The variation of memory usage with the number of customers is shown in Figure 7(b). In the OSMT architecture, the memory usage remains relatively constant irrespective of the number of customers. In the VMT architecture, however, the memory usage grows almost linearly with the number of customers. We explain the less-than-2% discrepancy in the increase between the disk- and RAM-disk-based OSMT setup by the varying buffer size requirements with respect to the latency of the medium: in the disk setup,

access time is slower and the buffers reach their maximum size already for a single customer. For 10 customers, there is a difference of about 45% in the memory usage of the two architectures when disk storage is used.

Figures 7(c) and 7(d) show the variation of throughput and loadfile execution time. As expected, the loadfile execution times are the lowest on the RAM-disk setups. The throughput reported by smbtorture is also higher on the RAM-disk setups. Independently of the type of disks used, the VMT architecture clearly gets an additional penalty from the higher CPU load of the system which results in lower throughput and higher loadfile execution time than the OSMT architecture.

Figure 7(e) shows the variation of system load with the number of customers. In both architectures, the system load is higher when disk storage is used because of all the cycles spent waiting on disk I/O. The system load has a much higher variance in the VMT architecture. We speculate that this is because of variations in the amounts of disk activity required to maintain the state of the virtual images on disk during the different benchmark runs, which resulted in a large variation in the measured load values.

Figure 7(f) shows system load as a function of the throughput. Generally, the lower the load and the higher the throughput, the better the scalability. Clearly, throughput scales better with system load in the RAM-disk setup than in the disk-based setup. Throughput scales best in the OSMT architecture using RAM disks. In the VMT architecture, the wide variation in system load in the disk-based setup results in a relatively steep curve, whereas the curve is flatter for the RAM-disk setup. Overall it can be seen that the OSMT architecture scales better than the VMT architecture.

6 Related Work

Although the designs of some free or open-source object storage cloud solutions [22, 23] are available, to the best of our knowledge no commercial cloud-storage provider has publicly documented its internal architecture. In this work, we analyze for the first time, how the technology applied for obtaining multi-tenancy impacts the security of the customer data.

In this section we discuss alternative techniques that can be leveraged to obtain similar security goals as the architectures analyzed in this work. Because we target this study at production environments, our two architectures are restricted to tools that have achieved a certain maturity and stability. Some of the isolation techniques mentioned in this section are too recent and do not yet satisfy these conditions.

Micro-kernels [24] and virtual-machine monitors are comparable to some extent [25]. In terms of security isolation, Hohmuth et al. [26] argue that the trusted computing base (TCB) is usually smaller with micro-kernels than with hypervisors. In particular, the authors suggest that extending virtualized systems with micro-kernel-like features, such as inter-process communication, can reduce the overall TCB. Although we do not use the TCB terminology to capture better the advantages of a layered security design, we believe their main argument also applies to some extent in the context of this work. For instance, in the VMT architecture, isolating the distributed filesystem[5] from the guests

[5] In the sense of Hohmuth et al. [26], the distributed filesystem is in the TCB.

running the filer services and establishing a stable and secure way of accessing the filesystem (e.g., paravirtualizing the distributed filesystem) would significantly improve the security of this architecture. To the best of our knowledge, the most mature existing technology for KVM that is close to realizing this goal is VirtFS [27].

Better isolation of the filer services and the distributed filesystem can also be achieved by improving the security of the Linux kernel, especially in the OSMT architecture. Using virtual machine introspection (VMI), Christodorescu et al. [28] present an architecture for kernel code integrity and memory protection of critical read-only sections (e.g. jump tables) to prevent most kernel-based rootkits with minimal overhead and without source code analysis. With source code analysis, Petroni and Hicks [29] prevent rootkits by ensuring control-flow integrity of the guest kernel at the hypervisor and therefore also prevent all control-flow redirection based attacks for the Linux kernel, which represents a significant security improvement.

Another approach to enhance the security of the kernel is grsecurity PaX [7], which provides a series of patches that mitigate the exploitation of some Linux kernel vulnerabilities with low performance overhead. In particular, it provides base address randomization for the kernel stack, prevents most user-space pointer dereferences by using segmentation, and prevents various information leaks which can be critical for successful exploitation of vulnerabilities. Other grsecurity patches also feature protection for the exploitation of user-space vulnerabilities.

7 Conclusion

We have presented in this work two alternatives for implementing a multi-tenant filesystem storage cloud, with one architecture isolating different tenants through containers in the OS and the other isolating the tenants through virtual machines in the hypervisor. Neither architecture offers strictly "better" security than the other one; rather, we view both as viable options for implementing multi-tenancy. We have observed that the overhead of the VMT architecture, due to the additional isolation layers, is significantly higher than that of the OSMT architecture as soon as multiple tenants (and not even a large number) access the same infrastructure. We conclude that, under cost constraints for a filesystem storage cloud, the OSMT architecture is a more attractive choice.

References

1. "IBM Scale Out Network Attached Storage." `http://www-03.ibm.com/systems/storage/network/sonas/`.
2. F. Schmuck and R. Haskin, "GPFS: A Shared-disk File System For Large Computing Clusters," in *Proc. File and Storage Technologies*, 2002.
3. A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proc. Linux Symposium*, vol. 1, 2007.
4. H. Cai, B. Reinwald, N. Wang, and C. Guo, "SaaS Multi-Tenancy: Framework, Technology, and Case Study," *International Journal of Cloud Applications and Computing (IJCAC)*, vol. 1, no. 1, 2011.
5. A. Traeger, A. Rai, C. Wright, and E. Zadok, "NFS File Handle Security," tech. rep., Computer Science Department, Stony Brook University, 2004.

6. S. Oehme, J. Deicke, J. Akelbein, R. Sahlberg, A. Tridgell, and R. Haskin, "IBM Scale out File Services: Reinventing network-attached storage," *IBM Journal of Research and Development*, vol. 52, no. 4.5, 2008.
7. "grsecurity." `http://grsecurity.net/`.
8. S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, 2002.
9. P. Kamp and R. Watson, "Jails: Confining the omnipotent root," in *Proc. International System Administration and Network Engineering*, 2000.
10. D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," in *Proc. System administration*, 2004.
11. B. McCarty, *SELinux: NSA's Open Source Security Enhanced Linux*. 2004.
12. S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computing Surveys*, vol. 43, Dec. 2010.
13. G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proc. Storage Security and Survivability*, 2005.
14. S. Schechter, *Computer Security Strength & Risk: A Quantitative Approach*. PhD thesis, Harvard University Cambridge, Massachusetts, 2004.
15. B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, 1999.
16. N. Hardy, "The Confused Deputy," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, 1988.
17. R. Wojtczuk, "Adventures with a certain Xen vulnerability (in the PVFB backend)." Message sent to bugtraq mailing list on October 15th, 2008.
18. T. Ormandy, "An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments," in *Proc. CanSecWest Applied Security Conference*, 2007.
19. K. Kortchinsky, "Cloudburst – Hacking 3D and Breaking out of VMware," 2009.
20. M. Blanc, K. Guerin, J. Lalande, and V. Le Port, "Mandatory Access Control implantation against potential NFS vulnerabilities," *International Symposium on Collaborative Technologies and Systems*, 2009.
21. "libvirt: The virtualization API." `http://libvirt.org/index.html`.
22. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," in *Proc. Cluster Computing and the Grid*, 2009.
23. "OpenStack Swift." `http://swift.openstack.org/`.
24. J. Liedtke, "On micro-kernel construction," in *Proc. SOSP*, 1995.
25. G. Heiser, V. Uhlig, and J. LeVasseur, "Are Virtual Machine Monitors Microkernels Done Right?," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1, 2006.
26. M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors," in *Proc. SIGOPS European workshop*, 2004.
27. V. Jujjuri, E. V. Hensbergen, and A. Liguori, "VirtFS – A virtualization aware File System pass-through," in *Proc. Ottawa Linux Symposium*, 2010.
28. M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni, "Cloud Security Is Not (Just) Virtualization Security: A Short Paper," in *Proc. CCSW*, 2009.
29. N. L. Petroni Jr and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," in *Proc. CCS*, 2007.