CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:1917–1935
Published online 14 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1134

Using a virtual machine to protect sensitive Grid resources

Xin Zhao[*,†], Kevin Borders and Atul Prakash

University of Michigan, 2260 Hayward, Ann Arbor, MI 48109-2121, U.S.A.

SUMMARY

Most Grid systems rely on their operating systems (OSs) to protect their sensitive files and networks. Unfortunately, modern OSs are very complex and it is difficult to completely avoid intrusions. Once intruders compromise the OS and gain system privilege, they can easily disable or bypass the OS security protections. This paper proposes a secure virtual Grid system, SVGrid, to protect sensitive system resources. SVGrid works by isolating Grid applications in Grid virtual machines. The Grid virtual machines' filesystem and network services are moved into a dedicated monitor virtual machine. All file and network accesses are forced to go through this monitor virtual machine, where SVGrid checks request parameters and only accepts the requests that comply with security rules. Because SVGrid enforces security policy in the isolated monitor virtual machine, it can continue to protect sensitive files and networks even if a Grid virtual machine is compromised. We tested SVGrid against attacks on Grid virtual machines. SVGrid was able to prevent all of them from accessing files and networks maliciously. We also evaluated the performance of SVGrid and found that performance cost was reasonable considering the security benefits of SVGrid. Furthermore, the experimental results show that the virtual remote procedure call mechanism proposed in this paper significantly improves system performance. Copyright © 2006 John Wiley & Sons, Ltd.

Received 7 June 2006; Accepted 3 August 2006

KEY WORDS: Grid computing; virtual machine; RPC

1. INTRODUCTION

Grids [1,2] are designed to enable resource sharing across multiple organizations and administrative domains. However, in reality, many organizations are reluctant to share their resources with external Grid users, because they are not confident that their resources are well protected from malicious users or applications. Popular Grid systems such as Globus [3] have incorporated various mechanisms for user

[*]Correspondence to: Xin Zhao, University of Michigan, 2260 Hayward, Ann Arbor, MI 48109-2121, U.S.A.
[†]E-mail: zhaoxin@eecs.umich.edu

Contract/grant sponsor: National Science Foundation ITR Award; contract/grant number: ATM 0325332

authentication and secure communication. These mechanisms are effective in stopping intruders who have no legitimate accounts. However, a legitimate but malicious Grid user can still log into the Grid system and mount attacks from inside. One can employ other security mechanisms to thwart attacks from legitimate Grid users. Possible mechanisms vary from simple discretionary access control (DAC) mechanisms to complicated mandatory access control (MAC) mechanisms. Most of these security mechanisms reside in the same space as their underlying operating system (OS). They have to assume the integrity of the underlying OSs. Unfortunately, OSs themselves are not always secure in reality. Modern OSs are very complex. They usually involve millions of lines of code and lots of components. A small implementation error in an OS kernel or privileged programs could be exploited to compromise the whole system. For example, a vulnerable SSH-1 daemon could be exploited by malicious users to gain administrative privilege [4]. Despite the best efforts of system researchers and administrators, it is difficult, if not impossible, to completely prevent system intrusion. Once hackers gain the system privilege, they can easily bypass or disable any security mechanisms deployed in the victim OS. Therefore, it is desirable to have a security mechanism that can continue protecting sensitive resources even in a compromised system.

This paper proposes a secure virtual grid system, called SVGrid, that can protect sensitive files and networks from a compromised OS. By leveraging virtual machine (VM) technology, SVGrid is able to run multiple VMs in a single physical host. Each Grid application is isolated in a single VM, called a 'Grid VM'. SVGrid moves the network and filesystem services of all Grid VMs into a dedicated, privileged VM. This VM is called the 'Monitor VM'. When a Grid application needs to access files or networks, its requests must be sent to the Monitor VM, where SVGrid checks the requests' arguments. SVGrid only serves the requests that satisfy security policies.

In the Monitor VM, SVGrid runs a VM-based filesystem, SVFS (Secure Virtual File System), to provide a transparent filesystem service to Grid VMs. All file access requests from Grid VMs will be checked by SVFS independently. SVFS also introduces the virtual remote procedure call (VRPC) mechanism to support fast data exchange across VM boundaries. VRPC significantly improves filesystem performance in an SVGrid system. For network protection, SVGrid deploys the iptables [5] firewall in the Monitor VM. For each Grid application, SVGrid loads appropriate network access rules into iptables and checks all network communications at runtime.

SVGrid brings the following benefits to a Grid computing system.

- *Secure isolation.* Grid applications are isolated from the real machine and can only access the resources allocated to their Grid VMs. Nothing a malicious Grid application does on its Grid VM can change or damage the other parts of the real computer.
- *Complete mediation.* All accesses to files and networks are subject to security checking.
- *Attack resistance.* Security policies are independently enforced in the isolated Monitor VM. A compromised Grid VM cannot disable the SVGrid protection. Therefore, while not guaranteeing the security of Grid VMs, SVGrid can still protect files and networks even if a Grid VM is compromised.
- *Ease of use.* Grid applications can transparently access networks and files, while isolating those resources in the Monitor VM. No modifications on Grid applications are required, making SVGrid easy to use.

We implemented a prototype of SVGrid for Linux-based Grid systems. The prototype was tested against eight real-world malicious applications. Experimental results showed that SVGrid was able to prevent all unauthorized accesses to sensitive files and networks. We also evaluated performance impacts of SVGrid. By comparing the application performance on SVGrid to that on a native Linux system, we found that performance cost was reasonable, considering the security benefits of SVGrid. The performance evaluation also showed that VRPC significantly improves the filesystem performance. Currently, SVGrid only focuses on protection of Linux-based Grid systems. However, there is no technical obstacle to applying the same idea to other OSs.

The remainder of this paper is organized as follows. Section 2 discusses possible solutions that a Grid system can use to protect its resources. Section 3 first gives the threat model, and then introduces the VM technology. Section 4 presents the SVGrid architecture and its major components. This section also analyzes the security of SVGrid. Section 5 presents the evaluation of the effectiveness and performance of SVGrid. Finally, Section 6 presents conclusions.

2. RELATED SECURITY SOLUTIONS

As a first barrier, existing Grid security infrastructures such as GSI [3] enable decentralized security management together with a single sign-on for users of the Grid. These infrastructures can effectively authenticate Grid users and secure network communication. Many intruders can be stopped by this kind of barrier. However, these infrastructures do not prevent Grid resource abuses once a Grid user logs in.

To protect Grid resources from malicious Grid users, one can grant Grid users the least number of privileges they need. The OS access control mechanism can then enforce security policies.

One can also deploy sandboxing systems such as *chroot* [6], *Janus* [7] and *GridBox* [8] to prevent untrusted applications from seeing the whole system resources.

Mandatory access control systems such as SELinux [9] and GridPortal [10] also help protect sensitive resources by associating immutable attributes with subjects and objects. MAC systems perform authorization checks based on those attributes.

However, the above approaches suffer from the same problem: intruders can turn off the protection if they manage to exploit an OS vulnerability [11], which is hard to completely avoid. In addition, experienced intruders can also read and write sensitive resources by directly accessing devices, rendering system call monitors ineffective.

One can isolate Grid applications from sensitive resources by running Grid applications in multiple distributed computers. Appropriate software such as PVM [12] can permit a heterogeneous collection of computers hooked together by a network to be used as a single large parallel computer. In a PVM system, sensitive files can be stored in a central server. Access to sensitive files is subject to security checking at the central server. By physically partitioning computers, the security moderator on the central server cannot be disabled even if other computers are compromised. However, this scheme falls short in two aspects. First, a distributed filesystem is much slower than local filesystems, which can significantly reduce the whole system performance. Second, using multiple physical machines increases both hardware and management costs.

VM systems such as VMWare [13] and Xen [14] can isolate Grid applications into guest VMs that run a single physical machine. Nothing a malicious application does on its VM can change or damage the real computer. However, if directly adopted, these VM systems do not provide efficient protection on sensitive files and networks. For example, these VM systems usually maintain a filesystem for each guest VM, but they do not provide good protection on the filesystems. An evil application can still taint its filesystem and disrupt subsequent applications on the same VM. For example, an evil application can modify Globus binaries to refuse serving all other applications on the same VM. One can get around this problem by creating a separate filesystem for each Grid application, but the overhead would be too high to be practical. Moreover, these VM systems do not dynamically adjust security policy according to different Grid user/applications, rendering the security policy enforcement less flexible.

Like VM systems, virtual server systems such as Virtuozzo [15] can run multiple isolated virtual private servers (VPSs) on a single physical server to share hardware, licenses and management effort with maximum efficiency. Virtual server systems achieve isolation at the OS level by modifying system calls. Different servers are allowed to share a single filesystem in a controlled manner. However, this advantage quickly becomes a disadvantage. Because the system calls are often changed with OS kernel upgrade, virtual server systems have to be modified for each kernel version. This causes hard upgrade. Moreover, OS-level isolation involves a lot of OS components, making the implementation very complex and error prone.

In addition, firewall software or hardware such as iptables [5] can be deployed. However, if the firewall is deployed on a Grid computer, it can be disabled by a successful intruder; if the firewall is deployed on a stand-alone device such as a router, it is less effective in protecting computers that are directly interconnected and behind the firewall. For example, a compromised machine can flood Internet worms within a subnet without going through the firewall-protected router.

3. SVGRID PRELIMINARY

In this section, we first describe the threat model of this paper. Then, we briefly present the Xen VM system on which SVGrid is built.

3.1. Threat model

SVGrid assumes that attackers do not have physical access to the machine. All intrusions are assumed to come across a network.

We assume that a guest OS can be compromised by an attacker despite preventative security measures. We define 'compromised' to mean that the attacker has administrative access to the guest OS. Once the OS is compromised, intruders can abuse various Grid resources. However, this paper only focuses on protections of network and filesystems.

Once a guest VM is compromised, we assume that a hacker will attempt to manipulate the network to mount active attacks [16–18] or passively sniff network communication [19]. The intruder may also access sensitive files to steal confidential information (e.g. passwords), taint system logs to cover up intrusion trails, or create back doors and Trojan horses to hide their presence and retain access to the machine [20,21]. On the other hand, Grid resource owners want to detect and prevent the above malicious activities.

SVGrid treats all unauthorized network-related activities as invalid. This includes waiting for connection on unauthorized ports, and sending packets to unauthorized hosts. The following types of files and directories are considered to be sensitive:

- files or directories that contain critical system configurations; for example, file `/etc/rc.d/rc.sysinit` and directory `/etc/rcX.d`;
- system executables and shared libraries;
- system logs that an intruder may want to modify to cover up the intrusion trail;
- Grid supporting systems such as Globus [3] and MPICH [22];
- other files and directories specified as sensitive by an administrator.

SVGrid requires administrators to specify the access permissions on these sensitive files. File accesses that violate the security policies are regarded as malicious.

SVGrid does not protect a guest VM's boot sectors and partition tables. They usually reside on fixed locations of a guest VM's virtual disk or even out of the virtual disks. Therefore, the protection of this kind of data is relatively easy and will not be discussed in this paper.

3.2. Overview of the Xen VM system

SVGrid is built on the Xen VM infrastructure. VMs have existed for 30 years, and used to be deployed on mainframes. Recently, VMs have been growing in popularity, and are now widely used on personal computers and servers. Examples of modern VM systems include VMWare [13], UML [23], FAUmachine [24], Denali [25], Disco [26] and Xen [14]. Some of these systems have led to successful businesses and are currently used by many large corporations [27].

Xen is a VM monitor for x86 that supports the execution of multiple guest OSs with resource isolation and performance close to that of a bare machine [14]. As shown in Figure 1, Xen is a thin software layer that runs directly on a physical machine's hardware. It provides each VM with a set of virtual interfaces that resemble direct interfaces to the underlying hardware. Applications on a VM can run without modification as if they were on a dedicated physical machine. Furthermore, Xen provides secure partitioning between VMs (known as `domains` in Xen terminology). Guest VMs run as if they are physically separated machines. They can only see and access their own virtual resources, preventing a malicious VM from directly manipulating other VM's resources. There are two kinds of VMs in a Xen server: Domain0 and DomainU. Domain0 is a required part of any Xen-based server. It is a privileged VM that can communicate with Xen via specified interfaces and control the running of other unprivileged VMs, DomainU.

Xen uses the paravirtualization technique to virtualize accesses to/from network and disk block devices. Accesses to these devices from a guest VM are fulfilled by two cooperative drivers: frontend driver and backend driver. The frontend driver runs in DomainU and has no direct access to physical devices. The frontend driver is responsible for transferring device I/O requests to the backend driver. The backend driver runs in the privileged VM, Domain0. It has direct access to physical devices and knows the information of devices that are allocated to every guest VM. The backend driver is responsible for processing virtual I/O requests received from the frontend driver and sending those requests to the physical devices to get served. We use a network as an example to illustrate how network access is virtualized. Suppose that a guest application needs to send a packet to a network. It first sends it to the guest OS as it does in a native Linux system. The frontend driver in the guest OS forwards the network I/O data to the backend driver via a VM-based communication channel. The backend driver translates the virtual I/O parameters into physical ones. Then, it issues the translated I/O command to the real device for processing. The processing results are finally returned to the frontend driver.

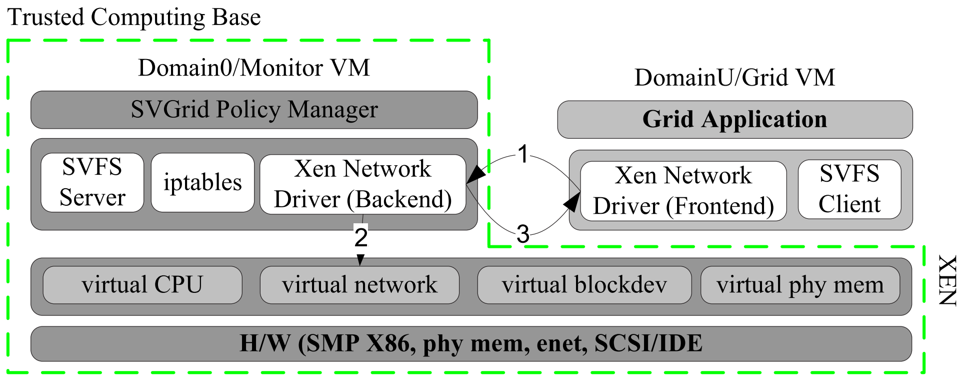

Figure 1. Xen-based SVGrid architecture.

Note that the frontend and backend drivers identify each other by the VM-based communication channel established between them. Xen is responsible for protecting this communication channel from being accessed by any third parties.

4. DESIGN OF SVGRID

In this section, we first present the SVGrid architecture. Then, we look at security policies supported in SVGrid. Next, we describe SVFS-based file protection and iptables-based network protection. Finally, we analyze the security of SVGrid.

4.1. System model

As Figure 1 shows, an SVGrid system consists of two categories of VMs: *Grid VMs* and *Monitor VM*. A Grid VM is dedicated to running one application for one Grid user at a time. An SVGrid system runs a task gateway. When Grid users need to submit a computing task, they must connect the task gateway to authenticate themselves. Currently, SVGrid uses the traditional Unix style authentication based on user name and password verification. Once a user is determined to be valid, SVGrid will create a new Grid VM and direct this user to the Grid VM to run his/her Grid application. The Grid VM, user and application thus form a one-to-one-to-one mapping.

The file and network services of Grid VMs are moved to Monitor VM. When one of the Grid applications needs to access files or networks, it must go through the Monitor VM, which happens transparently. In the Monitor VM, SVGrid identifies the source application by the Grid VM from which the requests are received. Then, SVGrid enforces appropriate security policies and only accepts requests that satisfy these policies.

```
# Sample Customized Policies.  Shows the options
user {abc}              # the policy is customized for grid user "abc"

#Network Description
IP { a.b.c.d/netmask } # abc is allowed to access host/subnet a.b.c.d/netmask
bind {
  80                              # app can bind to port 80,
  443                             # and port 443
}

#File Access Description
open_rw {                         # app can read/write these files
  /path/to/file1
 /path/to/dir/* }
open_a  { }                       # app can append to these files
unlink  { }                       # app can unlink these files
```

Figure 2. A sample SVGrid policy.

SVGrid protects files and networks with three major components: *SVGrid policy manager*, *SVFS filesystem* and *iptables-based network controller*. SVGrid policy manager is responsible for reconciling and loading security policies for Grid applications. SVGrid security policy is discussed in detail in Section 4.2.

SVFS is a secure virtual filesystem designed to provide a transparent filesystem service to Grid VMs. SVFS works like a network-based filesystem such as NFS [28]. A Grid VM can mount SVFS directories from the Monitor VM into its own namespace. The mounted directories and files can be accessed by a Grid application transparently, as if they reside in that Grid application's local VM. However, all file accesses must go through the Monitor VM, where SVFS can enforce appropriate policies and stop malicious file accesses. Section 4.3 describes the SVFS filesystem in detail.

Network access control is achieved with the combination of iptables firewall and Xen paravirtualized network driver. As described in Section 3.2, Xen uses frontend and backend drivers to virtualize accesses to network devices. All network packets generated by a guest VM are transparently forwarded to the backend driver in Domain0, which is the Monitor VM. SVGrid deploys iptables [5] network firewall in the Monitor VM to enforce the network security policies. Section 4.5 describes SVGrid network access control in more detail.

4.2. SVGrid security policy

An SVGrid policy is usually reconciled from three sources: *system default policy*, *application policy* and *administrator policy*. All three of these kinds of policy share a similar policy format. A sample policy shown in Figure 2 helps illustrate the policies. A policy usually consists of a network policy and a filesystem policy. The network policy specifies which hosts or ports can be accessed by Grid applications. The filesystem policy defines the access permission of filesystems.

The *system default policy* is defined by Grid computer administrators and works as a default security policy for all Grid VMs. In a system default policy, administrators can define the *base software environment*. A base software environment is the software collection that is necessary for running Grid applications. It usually includes OS files, system utilities, shared libraries and Grid supporting systems such as Globus [3] or MPICH [22]. Files in a base software environment are set to be read-only by default. This decision comes from two reasons. First, as all Grid VMs share the base software environment, it is important to protect the integrity of this environment. Second, the base software environment is installed by the Grid resource owner. Most Grid applications only use them and seldom need to change these files. SVGrid also maintains a home directory for each Grid user. Grid users can freely modify any files in their home directories. SVGrid denies any accesses to directories or files that are not explicitly specified in the system default policy. By default, a Grid application is only allowed to connect back to the host from which the Grid task is submitted.

System default policy could be too restrictive in many circumstances. Some Grid applications need to access machines other than their source hosts. For example, an application may need to download data from other machines before starting to run. Grid applications may also need to create some files outside their home directories. For example, some Grid applications need to create a lock file in a `/tmp` directory or a `/var` directory to achieve mutual exclusion. These operations are normal and should be allowed. The *application policy* is designed to provide this flexibility. An application policy is defined by a Grid user and submitted along with a specific application. Its rules only apply to this application. In the application policy, a Grid user can request some exceptions from the system default policy during the running of the specified application. SVGrid usually accepts an application policy, as long as this policy does not conflict with rules explicitly defined in the system default policy. However, the extra permissions requested in the user-specified policy will be logged and strictly enforced. While SVGrid does not know the behavior details of a Grid application, it records the hosts and files that an application potentially tries to access. If some malicious activities are detected on these objects, SVGrid can easily trace them back to the hostile Grid user. This accountability helps keep Grid users on track, since their behavior will be logged and abuse of resources will incur severe penalties.

Application policies will be accepted only if they do not conflict with rules explicitly defined in the system default policy. However, in some circumstances, Grid applications need to override some of these rules. For example, while default rules specify directory `/usr/lib` to be read-only, some Grid applications need to install their shared libraries into that directory. This kind of operation is often too sensitive to be allowed. However, for trusted users or applications, a Grid system administrator can define an *administrator policy* to allow this kind of activity. An administrator policy is also Grid application specific. It has higher preference than the other two kinds of policies. If rules in an administrator policy conflict with any other policies, SVGrid will accept the administrator policy. Similar to the application policy, the administrator policy will also be logged for the purpose of misfeasance trace back.

4.2.1. Limitation of the SVGrid security policy

An application must explicitly specify the target host's name or IP address, if it wants to access hosts other than its source machine. Sometimes, this requirement is hard to meet. For example, some Grid tasks use the MPICH scheduler to divide themselves into multiple subtasks and dispatch these subtasks to idle hosts for running. Before the scheduler dispatches subtasks, Grid users do not know

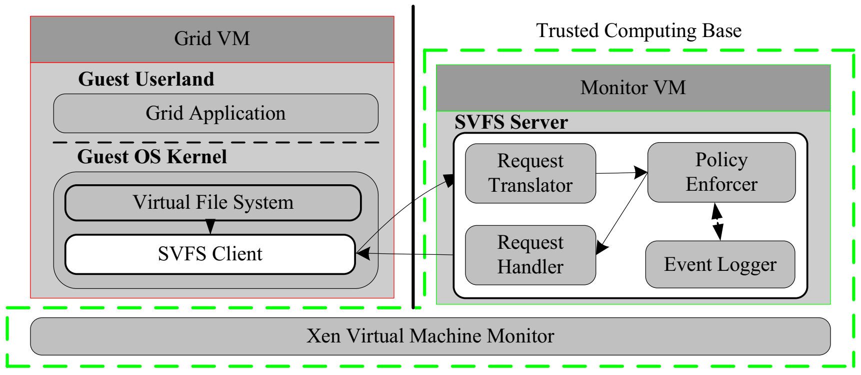

Figure 3. SVFS filesystem architecture.

the target hosts on which the subtasks will run. They, therefore, cannot report target host names as required. Currently, SVGrid does not address this problem. One possible solution is to extend the schedulers to report the target hosts' IP addresses to the SVGrid policy engine before running Grid tasks. This solution is realistic, because only a few schedulers such as the MPICH scheduler are used by most Grid applications. Extension on these schedulers will satisfy most circumstances. An alternative solution is to allow Grid applications to communicate with any network hosts they want but log their network activities. This method does not stop network attacks, but allows administrators to trace the attacker. One challenge is that logging network activities could significantly decrease the system performance.

4.3. SVFS filesystem

SVFS has been developed to provide enhanced data security while keeping its convenience on existing systems. We minimize the system modification to avoid disrupting the user working pattern. A SVFS filesystem consists of client side and server side, as shown in Figure 3. The client side runs in a Grid VM and forwards file access requests to the server side, which runs in a Monitor VM and serves the received requests. Our prototype of SVFS is able to use NFSv2 protocol [28] to achieve transparent file access across the VM boundary. The same architecture also works for other distributed filesystem protocols such as CIFS [29] and Samba [30]. NFSv2 was chosen over other protocols because it is simple and good for a fast prototype implementation.

4.3.1. SVFS client

An SVFS client runs at the kernel level of each Grid VM. It registers the SVFS filesystem and is hooked to the virtual filesystem (VFS) layer [31]. Upon receiving a request to files on the SVFS filesystem, the Linux VFS layer determines that the requested file object belongs to SVFS and calls the corresponding SVFS function. The SVFS client then packs the file request parameters and puts the packed request into the request queue by issuing a VRPC to the SVFS server (the VRPC mechanism is described in detail in Section 4.4). After the VRPC returns, SVFS client unpacks the result and sends it back to the Linux VFS layer, giving guest applications an illusion that the request is processed locally.

4.3.2. SVFS server

SVFS server runs in the Monitor VM and uses the Ext3 filesystem as its local filesystem to store files for guest VMs. SVFS consists of four major components: *request translator*, *policy enforcer*, *request handler* and *event logger*.

- *Request translator*. Request translator determines where a request comes from. The source VMs are identified by the communication channel over which the requests are received. Once the source VM is identified, request translator calls the appropriate decoder to parse the request parameters. Finally, the preprocessed request is forwarded to the policy enforcer, the next module in the SVFS server.
- *Policy enforcer*. Policy enforcer checks the parameters of incoming requests against security policies. If the requested operation (i.e. write to /etc/shadow) is not allowed for the requesting VM, the request will be denied immediately. Only valid requests will be forwarded to the request handler to get served.
- *Request handler*. SVFS uses the Ext3 filesystem to store files for guest VMs. Request handler serves requests by calling Ext3 filesystem functions. After the Ext3 filesystem functions process the request, the request handler packs the result and returns it to the client VM.
- *Event logger*. Requests that violate the security policy are logged by the event logger. Optionally, the event logger can be configured to log all requests, along with their status.

4.4. VRPC

Because all SVFS files reside in the Monitor VM, they can be accessed from other guest VMs only through inter-VM communication. It is important to have a mechanism that can efficiently exchange data between SVFS client and server. RPC is traditionally a convenient way to exchange data over networks. We initially used RPC for communication between the SVFS client and the SVFS server. However, the traditional RPC mechanism hurts SVFS performance because it involves expensive operations, such as XDR encoding/decoding, packet packing/unpacking and connection establishing. These operations are unnecessary for the SVFS system as the client and server sides reside on the same physical host.

Bershad *et al.* [32] proposed the use of a thread tunneling technique to improve the performance of RPC communication within a single machine. However, while a Xen server runs on a single physical host, VMs running on it are isolated from one another, as if they were physically separated machines. Hence, RPC communication between SVFS client and server essentially occurs across two machines. This fundamental difference makes thread tunneling infeasible on a VM system. Moreover, thread tunneling is effective only if RPC arguments are small. However, SVFS often needs to transmit the bulk of file data.

We therefore designed the Xen-based VRPC, a modified RPC mechanism, to support fast data exchange between the SVFS client and server running on the same physical machine. VRPC shares the same protocol with traditional RPC, but differs in that VRPC does not rely on a network to transmit data and is more efficient. The VRPC work flow is the same as that of standard RPC, and can be illustrated as follows.

1. The calling thread of execution in the client process marshals a procedure reference and a set of arguments into a buffer, transfers this buffer to the server process and blocks awaiting a reply.
2. A thread within the server process reads the procedure reference and the arguments out of the buffer, and constructs a call frame to invoke the specified procedure with the supplied arguments. On return from the invocation, the server thread marshals any results or exceptions into a reply buffer and transfers it back to the client.
3. The client thread is unblocked, unmarshals the results and returns to its caller or raises an exception as appropriate.

VRPC improves the data exchange performance by leveraging Xen's memory re-mapping support. In a Xen server, a VM can allocate a memory region and report the starting address and size of this region to another VM, which can then map that memory region into its own memory space and access data stored in that region directly. VRPC leverages this memory re-mapping support to setup a shared memory space between Monitor VM and Grid VM. These two VMs can then exchange data using this memory space, which avoids expensive network transmission and reduces the number of memory copies needed for data exchange to zero. Because the data stored in the shared memory can be seen directly by two VMs, no extra memory copy is needed to send data across VM boundaries. In contrast, network-based RPC needs at least two memory copies: the sender copying data to the network stack from the data buffer and the receiver copying data from the network stack to the data buffer. In addition, the inter-domain communication happens on the same physical host, as all VMs have the same endian order. XDR encoding and decoding is unnecessary for VRPC and is thus removed, which further reduces the VRPC overheads.

Different SVFS requests involve different amounts of data exchange. For example, the function *lookup()* searches a certain directory for a specified file. The input parameters are file name and parent directory inode. The output values are file attributes such as file size, modification time and access rights. This function only involves a small amount of data exchange. On the other hand, the functions *read()* or *write()* can involve data of arbitrary length. VRPC implements a hybrid memory re-mapping mechanism to handle different types of data request. For functions that need to exchange data of variable length or of more than 4 kB of data transfer, VRPC allows SVFS client to dynamically allocate memory and map it into Monitor VM for data sharing. In SVFS, dynamic memory sharing is used in the following functions: `read()`, `write()`, `readlink()` and `readdir()`. All of these functions involve variable-length data exchange. Other filesystem requests and results can be packed into relatively small buffers (less than 4 kB bytes). At the initialization stage, the SVFS client allocates a memory region that is sufficient for such requests. The memory region is mapped into the SVFS server's memory space and shared by the SVFS client and server. This static memory sharing needs only one Xen call to re-map memory at the initialization stage, and is thus very cheap. The dynamic memory sharing mechanism is more expensive, because it needs a Xen call for each memory mapping. However, it allows an arbitrary length of data to be transferred in a VRPC call.

A *communication ring*, consisting of statically allocated shared memory, is used to exchange request parameters and responses between an SVFS client and the SVFS server. The communication ring is divided into multiple 4 kB slots, with each request and response using one slot. The SVFS client puts file requests and fetches responses. The SVFS server fetches the requests and put the responses back to the ring.

When a new request or response is put into the shared ring, its producer has to notify the other party that new data are available. VRPC uses the event channel mechanism provided by Xen to achieve notification. Event channels are an asynchronous inter-VM notification mechanism. Before a Grid VM starts to run, Monitor VM instantiates an event channel between this Grid VM and itself. Both VMs can request that a virtual IRQ (interrupt request line) be attached to notifications on a given channel. The result of this is that one VM can send a notification on an event channel, resulting in an interrupt in the other VM. By installing an IRQ handler, the SVFS server and client can detect new requests or responses that are put by the other party. They can then check the shared communication ring to get new requests or responses. The event channel can only transmit a 1-bit message, so it is not appropriate to transfer bulk data, but it is reasonable for event notification.

An example helps illustrate the working of a VRPC. Suppose that a Grid VM wants to read 8 kB of data from a sensitive file f stored in Monitor VM. The SVFS client first allocates a buffer of size 8 kB. The SVFS client then puts a request in the shared ring that includes the read function code, the file handle, offset, the starting physical address x of the buffer for receiving the results and the buffer size 8 kB. It then notifies the SVFS server of this new request via the event notification mechanism. Upon receiving this request, the SVFS server maps the memory buffer of length 8 kB that starts at physical address x into its own memory space and executes the read operation, resulting in the file data being placed in the mapped memory buffer. After the operation is completed, Monitor VM notifies the Grid VM. The SVFS client receives an interrupt, and returns the data that are in the mapped memory region.

A malicious Grid VM may try to request excessive memory sharing, hoping to exhaust all of the memory of the Monitor VM as an attempt of a denial-of-service (DoS) attack. However, the SVFS server never allocates memory in Monitor VM for sharing with SVFS clients. Instead, it only maps the memory allocated by an SVFS client in a Grid VM. Therefore, this kind of DoS attack is impossible. Because the starting address and the size of the memory region are reported by the SVFS client from the Grid VM, a hacker may also try to report a false address or memory size for sharing, hoping to disrupt the SVFS server or inject malicious data into Monitor VM. However, this attack is unlikely to succeed. While the SVFS server has no way of validating the reported starting memory address and size, a false starting address only causes memory mapping failure (when the memory space does not belong to the guest VM) or the SVFS server to access data at the wrong place in the guest VM, which will only disrupt the malicious guest VM itself. Moreover, this method is unlikely to be used to attack a guest OS either. The SVFS server remaps the memory region from the foreign domain only if it receives a request from an SVFS client. The SVFS client runs in the guest OS kernel. If intruders can impersonate the SVFS client to send false memory information, they must already have the system privilege on the guest VM. Reporting false information will not further benefit the intruder.

4.5. Network access control

The network access control in SVGrid is achieved through the combination of the Xen paravirtualized network driver and the iptables network firewall [5]. As described in Section 3.2, accesses to network devices from a Grid VM are virtualized using a pair of virtual device drivers. Monitor VM, which has access to physical network devices, runs the *backend* driver. Each Grid VM runs a *frontend* driver.

If a Grid application needs to send packets over the network, it still creates a socket and sends data through this socket, just as it does in a native Linux system. The difference is that operations happen underneath. The packets to be sent are intercepted by the frontend driver, which in turn forwards these packets over a VM-based communication channel to the backend driver. Upon receiving the packets from the frontend driver, the backend driver checks these packets and sends them over the network.

For a Grid VM, the backend driver may be viewed as a virtual Ethernet switch element with each Grid VM having one or more virtual network interfaces connected to it. Because all network communication of Grid VMs must go through the backend driver, we deploy the iptables firewall in Monitor VM to enforce network security policies. The network security checking includes two parts.

1. *Source authentication*. Many network attackers attempt to hide themselves by spoofing source IP or MAC addresses in attacking packets. The backend driver checks that source MAC and IP addresses of packets from Grid VMs are authentic. Upon receiving a packet, the backend driver first identifies the source VM by the communication channel over which the packet is received. It then extracts the source MAC and IP addresses from this packet. Because Monitor VM knows the authentic MAC and IP addresses of all Grid VMs, it can easily determine whether the source IP or MAC addresses claimed in the packet are spoofed or not. In an SVGrid system, source authentication is enabled by turning on the ‘anti-spoof’ switch in the startup scripts of Grid VMs.
2. *Enforcement of access control policy*. SVGrid uses iptables network firewall to control the hosts and ports that a Grid VM can access. With source authentication, IP/MAC spoofing can be detected. Therefore, network security policies can be safely enforced on the basis of source IP/interface. Before a Grid VM is activated to run, the SVGrid policy manager translates this VM’s network policy into IP/interface-oriented rules. These rules are then loaded into the iptables policy engine and enforced throughout the lifetime of this Grid VM.

4.6. Security analysis of SVGrid

The security of SVGrid relies on several preconditions. First, the trusted computing base (TCB) of SVGrid must be difficult for an attacker to compromise. Our argument for the security of SVGrid rests on this assumption. A compromise of SVGrid’s TCB is a catastrophic failure in an SVFS system.

Second, Monitor VM must be securely isolated from Grid VMs. SVGrid can continue to function in a compromised environment, *only if* a compromised Grid VM cannot directly access Monitor VM. This requires that Monitor VM be securely isolated from other VMs regardless of whether other VMs are compromised or not.

Last, communication between Monitor VM and Grid VMs must be secure. Grid VMs access network and filesystem services via inter-VM communication with Monitor VM. If the communication channel is not secure, a malicious VM can impersonate Monitor VM or another Grid VM to inject false data.

Note that SVGrid does not rely on the security of Grid VMs. A malicious VM can send any invalid file or network requests, but all requests will be checked independently by SVGrid.

4.6.1. TCB assurance

The TCB of SVGrid is the combination of Virtual Machine Monitor (VMM) and Monitor VM. We believe that this TCB is hard to compromise for two reasons. First, compared to a full operating system that typically has several million lines of code and runs lots of services, VMM and Monitor VM are much simpler and more likely to be implemented correctly. Specifically, VMM is primarily responsible for virtualizing the hardware of a single physical machine and partitioning it into logically separate VMs. A VMM is usually built in less than 100K lines of code. For example, Disco [26] and Denali [25] have around 30K lines of code [33], and Xen has around 60K lines of code. Although Monitor VM also runs an OS, it is dedicated to running SVGrid services only. All irrelevant components are removed.

Second, the interfaces to VMM and the Monitor VM are much simpler, more constrained and well specified than a standard OS, which reduces the risk of being attacked. Specifically, Xen VM monitor can be accessed only through 28 predefined hypervisor calls. Moreover, Monitor VM (Domain0) needs to expose few communication channels to each guest VM for I/O virtualization and SVFS filesystem service. In contrast, modern OSs expose many more interfaces to the outside. For example, a standard Linux OS can be accessed via many system calls (Linux kernel 2.6.11 exposes 289 system calls), special devices such as `/dev/kmem`, kernel modules, plenty of privileged programs (sendmail and sshd, for example) and a variety of network services provided by the OS or other software vendors. One may argue that the privileged Monitor VM also exposes network interfaces to perform administrative functions. However, several easy solutions can reduce this risk. The system can be configured to only allow administration on the console, removing the need for exposing the network for administration. If remote administration is still desired, the administrative daemon should be the only daemon exposed by the Monitor VM. This daemon should be protected by the network firewall and should only expose limited network ports. While there is a risk that the administrative daemon is compromised, because developers only need to focus on one daemon, they can carefully validate this daemon, reducing the risk to a minimum. If all of the above precautions are taken, then it would be exceedingly difficult for a remote attacker to compromise an administrative daemon.

4.6.2. Isolation between the Monitor VM and Grid VMs

In an SVGrid system, Xen VM monitor securely partitions VMs from one another. Therefore, Monitor VM cannot be accessed by another VM, which prevents SVGrid protection mechanisms from being disabled. In addition, the isolation also prevents a malicious Grid VM from accessing physical devices directly, which forces Grid VMs to send all file and network requests to Monitor VM to access devices. Monitor VM can therefore easily achieve complete mediation on all file and network accesses.

4.6.3. Secure communication between Monitor VM and Grid VMs

Secure communication between Monitor VM and Grid VMs contains a twofold meaning. First, the sources of communication data are authentic. Second, communication data are not modified by any third party during transmission. SVGrid uses VM-based communication channels. These are essentially a memory region allocated in a Grid VM and mapped into Monitor VM's memory space. VMM protects this memory so as to be accessible only to the owner VM and Monitor VM. No third party can access this communication channel. Therefore, both the owner VM and Monitor VM can identify each other by this communication channel. No source address spoofing is possible. For the same reason, the data transmitted over the VM-based communication channel cannot be accessed or modified by any third party during transmission.

5. EVALUATION

5.1. Security evaluation

In this section, we present the security evaluation results of the SVGrid prototype. The evaluation consisted of testing known malicious applications to see whether they were blocked by SVGrid. We installed eight malicious applications. Six of them tried to modify sensitive files. One of them sent out malicious packets to other online hosts. Another listened on unauthorized ports to allow remote users to establish connection back to the testing machine. All of the malicious applications were obtained from public sources[‡]. For this experiment, malicious applications were run that already had administrative privileges on the guest OS.

We derived file access policies from default Tripwire policies that are included in the standard Tripwire [34] package. In general, we set system directories and files within them to be read-only for normal VMs. These included `/etc`, `/sbin`, `/lib` `/usr/lib`, `/usr/bin` and `/usr/sbin`. Directory `/dev` is managed by udev [35], which allows device files to be generated in memory instead of disk. SVGrid prevents new files from being created on disk in the `/dev` directory. For a few variable files in the `/etc` directory (`/etc/dhclient-eth0.conf`, for example), we moved them to a different directory `/var/etc` and left symbolic links in `/etc`. As such, we can lock down the `/etc` directory while still allowing variable files to be modified at runtime.

For the network security policy, we only allow a Grid application to communicate with its source host, because malicious codes usually do not report the real ports or hosts they will access.

Table I summarizes results from our attempts to install eight malicious applications on a guest OS. SVGrid was able to immediately stop six rootkits from installing. The six rootkits were `t0rn`, `Lrk5`, `Flea`, `Adore-0.31`, `login/PAM backdoors` and `Knark-0.59`. These rootkits relied on modification of sensitive system files that were protected by SVFS. Examples of standard system files that they tried to modify include `netstat`, `tcpd`, `ls`, `ps`, `pstree`, `top`, `read`, `write` and `ifconfig`.

`mworm` is a modified Internet worm. It can flood malicious packets to other online machines. `SAdoor` is a standard backdoor program that accepts connections from remote users and allows a remote user to take control of the machine. Because these two applications do not modify sensitive files protected by SVGrid and can run from a user's home directory, SVGrid did not prevent them from installing. However, their malicious network activities were stopped by SVGrid, since these activities failed to comply with network security policies.

5.2. Performance evaluation

In this section, we first evaluate the performance of SVFS by comparing it with a native Ext3 filesystem and a locally run network-based filesystem. The results show that SVFS offers enhanced security features with reasonable performance overhead. The results also show that the VRPC mechanism can significantly improve overall system performance when compared to a network RPC mechanism.

[‡]See http://packetstormsecurity.nl/UNIX/penetration/rootkits/ and http://www.antiserver.it/backdoor-rootkit/.

Table I. Common attack evaluation results.

Application name	Access sensitive files maliciously?	Access network maliciously?	Prevented by SVGrid?
`t0rn`	✓		✓
`Lrk5`	✓		✓
`Flea`	✓		✓
`Adore-0.31`	✓		✓
`login/PAM backdoors`	✓		✓
`Knark-0.59`	✓		✓
`mworm`		✓	✓
`SAdoor`		✓	✓

5.2.1. Experimental setup

The SVFS prototype runs on a Xen 2.0.6 VM system. Monitor VM is deployed in Xen's privileged Domain0, and runs Linux kernel 2.4.29-xen0. Monitor VM uses the Ext3 filesystem to store files for Grid VMs. The Ext3 filesystem is mounted with the 'data = journal' option and uses a 50 MB RAM disk [36] as the journal device. A Grid VM was also used for this experiment, which ran Linux kernel 2.4.29-xenU. All experiments were run on a PC with a Pentium IV 3.0 MHz processor and 512 MB of RAM. All VMs were configured to use 128 MB of RAM. To evaluate the performance impact of SVFS, we used a Linux kernel built to emulate a Grid task since it has the characteristics of most of the computational Grid applications we ever used: it is CPU intensive, but also needs to access some files.

The kernel build benchmark consists of three phases.

- The *unpack phase* decompresses a tar archive of Linux 2.4.27 kernel (the archive is approximately 173 MB). This phase creates many small files and directories.
- The *configure phase* determines the source code dependencies and involves many small file reads.
- The *build phase* compiles and links the kernel source code, then removes temporary files. It is the most CPU intensive phase, and generates many intermediate object files.

Using the Linux kernel build benchmark, we compared three filesystems. Two of them were versions of SVFS: the first version uses VRPCs to communicate between VMs, and the second uses network-based RPCs with read and write transfer sizes setting to 8 kB. These SVFS versions were compared to the native Ext3 filesystem running within a guest OS. For both SVFS versions, all temporary files were placed on the Monitor VM.

To examine how SVGrid impacts on the network performance of a Grid application, we also measured the time used to `scp` files from a Grid VM to the Monitor VM. Tested file lengths include 1, 2, 5, 10, 15, 30, 50 and 100 MB. We compared the network performance of a system with SVGrid network protection and a system without SVGrid protection.

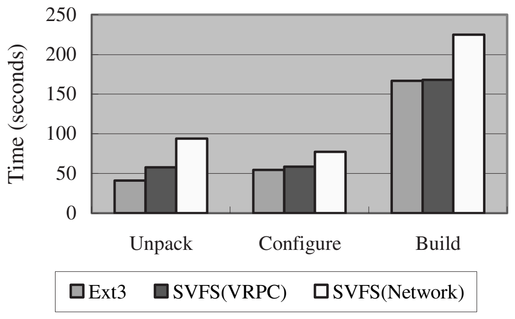

Figure 4. Linux kernel build benchmark.

Table II. Network performance evaluation results.

	File size (MB)							
	1	2	5	10	15	30	50	100
No SVGrid	0.39	0.42	0.70	1.25	1.81	3.40	5.60	11.05
With SVGrid	0.40	0.44	0.72	1.29	1.84	3.45	5.68	11.20
Difference (%)	1.96	3.48	2.67	3.37	1.88	1.45	1.38	1.38

5.2.2. *Performance comparison*

Figure 4 shows the difference in performance between the native Ext3 filesystem, SVFS using VRPCs and SVFS using network-based RPCs. Across all three phases, the Ext3 filesystem was the fastest, and SVFS with VRPCs was 8.3% slower, while SVFS with network RPCs was 51% slower than Ext3. Given these results, it is clear that VRPC is much faster than network-based RPCs. Compared to the local filesystem, SVFS offers a higher degree of security without incurring a significant performance penalty. The *unpack phase* of the benchmark was where VRPC-based SVFS and the native Ext3 filesystem showed the most difference. We believe the reason for this is that SVFS relies on NFS consistency semantics. The NFS consistency semantics require file data to be flushed to disk when closing a file, while Ext3 allows lazy disk updates. The *unpack phase* of the kernel compilation benchmark involves creating, writing and closing of many small files. The current implementation of SVFS is unoptimized. Moving the VRPC implementation from NFS version 2 to version 3 should help improve performance. The use of aggressive meta-data caching and update aggregation mechanisms (used by iSCSI) could further improve performance when creating many small files [37].

The experimental results listed in Table II show that SVGrid network protection only added 1.38% to 3.48% of file transmission time. Therefore, we believe that the network performance impact of SVGrid is trivial.

6. CONCLUSION

This paper has proposed SVGrid, a secure virtual Grid running environment, to protect sensitive files and networks for Grid computers. SVGrid works by containing Grid applications in one or more Grid VMs and deploying a Monitor VM to mediate all file and network requests from Grid VMs. By moving the filesystem and network services of all Grid VMs into the Monitor VM, SVGrid forces all file and network accesses from Grid VMs to go through the checking in Monitor VM before getting served. We have developed the SVFS filesystem to protect sensitive files and provide filesystem services to Grid VMs transparently. The network access control is achieved with the combination of the Xen network driver and the iptables firewall. SVGrid protection is non-bypassable and cannot be disabled.

To evaluate the effectiveness of SVGrid, we tested it against eight malicious applications. SVFS was successfully able to prevent all of them from modifying sensitive files or maliciously accessing networks. Also, we have evaluated the performance of the SVGrid system and found that SVGrid was able to provide enhanced security with reasonable performance penalty. In particular, we have found that our VRPC mechanism improves the system performance significantly.

REFERENCES

1. Foster IT. The anatomy of the Grid: Enabling scalable virtual organizations. *Proceedings of the 7th International Euro-Par Conference on Parallel Processing (Euro-Par '01)*, London, 2001. Springer: New York, 2001.
2. Foster I, Kesselman C (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Fransisco, CA, 1999.
3. Foster I, Kesselman C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
4. Zalewski M. Remote vulnerability in SSH daemon CRC32 compensation attack detector, February 2001. http://www.bindview.com/Services/RAZOR/Advisories/2001/adv_ssh1crc.cfm [September 2006].
5. NetFilter/IPTables Project Team. What is netfilter/iptables? http://www.netfilter.org [September 2006].
6. McGrath R. Chroot–run command or interactive shell with special root directory. *The Linux Manual Pages*. Free Software Foundation, 2004.
7. Goldberg I, Wagner D, Thomas R, Brewer E. A secure environment for untrusted helper applications. *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, 1996. USENIX Association: Berkeley, CA, 1996; 1–13.
8. Dodonov E, Sousa JQ, Guardia HC. Gridbox: Securing hosts from malicious and greedy applications. *Proceedings of the 2nd Workshop on Middleware for Grid Computing*, New York, NY, 2004. ACM Press: New York, 2004; 17–22.
9. Loscocco P, Smalley S. Integrating flexible support for security policies into the linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association: Berkeley, CA, 2001; 29–42.
10. Butt AR, Adabala S, Kapadia NH, Figueiredo RJ, Fortes JAB. Grid-computing portals and security issues. *Journal of Parallel and Distributed Computing* 2003; **63**(10):1006–1014.
11. Kernel brk() vulnerability. http://seclists.org/lists/bugtraq/2003/Dec/0064.html [September 2006].
12. Sunderam VS. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience* 1990; **2**(4):315–339.
13. Sugerman J, Venkitachalam G, Lim B. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, 2001. USENIX Association: Berkeley, CA, 2001; 1–14.
14. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, New York, NY, 2003. ACM Press: New York, 2003; 164–177.
15. SWsoft Inc. Virtuozzo for Windows and Linux server virtualization. http://www.swsoft.com/en/products/virtuozzo/ [September 2006].
16. Spafford EH. Crisis and aftermath. *Communications of the ACM* 1989; **32**(6):678–687.
17. Schuba CL, Krsul IV, Kuhn MG, spafford EH, Sundaram A, Zamboni D. Analysis of a denial of service attack on tcp. *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP '97)*, Washington, DC, 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 208.

18. de Vivo M, Carrasco E, Isern G, de Vivo GO. A review of port scanning techniques. *SIGCOMM Computer Communication Review* 1999; **29**(2):41–48.
19. Bradley T. Introduction to packet sniffing. http://netsecurity.about.com/cs/hackertools/a/aa121403.htm [September 2006].
20. A new Adore rootkit. http://lwn.net/Articles/75990 [September 2006].
21. Beale J. Detecting server compromises. *Information Security Magazine*. TechTarget, 2003. Available at: http://infosecuritymag.techtarget.com/2003/feb/linuxguru.shtml.
22. Aumage O, Mercier G, Namyst R. MPICH/Madeleine: A true multi-protocol MPI for high performance networks. *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01)*, Washington, DC, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 51.
23. Dike J. A user-mode port of the Linux kernel. *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000, Spenneberg R (ed.). USENIX Association: Berkeley, CA, 2000. Available at: http://www.linuxshowcase.org/2000/2000papers/papers/dike/dike.pdf [September 2006].
24. Höxer H-J, Sieh V, Waitz M. Advanced virtualization techniques for FAUmachine. *Proceedings of the 11th International Linux System Technology Conference*, Erlangen, Germany, 7–10 September 2004, Spenneberg R (ed.); 1–12. Available at: http://www3.informatik.uni-erlangen.de/Publications/Articles/hoexer_linuxkongress04/pdf [September 2006].
25. Whitaker A, Shaw M, Gribble SD. Scale and performance in the Denali isolation kernel. *SIGOPS Operating System Review* 2002; **36**:195–209.
26. Bugnion E, Devine S, Govil K, Rosenblum M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems* 1997; **15**(4):412–447.
27. Aslett M. A virtual success. *Computer Business Review Online*. http://www.cbronline.com/content/COMP/magazine/Articles/Servers_Mainframes/AVirtual-Success.asp.
28. Sandberg R. The Sun network filesystem: Design, implementation, and experience. *Distributed Computing Systems: Concepts and Structures*, Ananda AL, Srinivasan B (ed.). IEEE Computer Society Press: Los Alamitos, CA, 1992; 300–316.
29. Leach P, Perry D. CIFS: A Common Internet File System. *Microsoft Interactive Developer*, November 1996. http://www.microsoft.com/mind/1196/cifs.asp [September 1996].
30. Samba Team. Samba: Opening Windows to a wider world. http://us1.samba.org/samba [September 2006].
31. Bovet DP, Cassetti M. *Understanding the Linux Kernel*, Oram A (ed.). O'Reilly: Sebastopol, CA, 2000.
32. Bershad B, Anderson T, Lazowska E, Levy H. Lightweight remote procedure call. *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, New York, NY, 1989. ACM Press: New York, 1989; 102–113.
33. Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 191–206.
34. Kim GH, Spafford EH. The design and implementation of Tripwire: A file system integrity checker. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM Press: New York, 1994; 18–29.
35. Kroah-Hartman G. udev—a userspace implementation of devfs. *Proceedings of the Linux Symposium*, 23–26 July 2003. Available at: http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Kroah-Hartman-OLS2003.pdf [September 2006].
36. Morton A. Using the Ext3 filesystem in 2.4 kernels. http://www.zip.com.au/~akpm/linux/ext3/ext3-usage.html [September 2006].
37. Radkov P, Yin L, Goyal P, Sarkar P, Shenoy PJ. A performance comparison of NFS and iSCSI for IP-networked storage. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, San Francisco, CA, 2004. USENIX Association: Berkeley, CA 2004; 101–114.