atsec information security corporation
9130 Jollyville Road, Suite 260
Austin, TX 78759
Tel: 512-349-7525
Fax: 512-349-7933
www.atsec.com

KVM Security Comparison

Version:	1.01	**Status:**	Released
Owner:	Stephan Mueller	**Classification:**	public
		Last Mod. Date:	2009-11-02

Table of contents

1 Management Summary

Virtual machines allow an effective and efficient use of hardware by reducing the number of idle hardware, migration of environments to different systems and in general a better flexibility in utilizing hardware. Users and administrators can choose among many different virtual machine monitor (VMM) implementations. All of these VMMs support the basic concept of virtualizing a physical computer to allow concurrent execution of multiple operating systems. By using hardware support for virtualization, almost all VMMs allow different operating systems to be supported in their guest virtual machines, including Red Hat Enterprise Linux and other Linux distributions, Microsoft Windows, different BSD variants, and others.
To support users and administrators in selecting which VMM implementation is most suitable for their needs, the analysis compares the security-relevant functionality of Red Hat's KVM with other VMM implementations based on attack vectors and usage scenarios. The analysis explains how the different VMM implementations mitigate potential attacks and support different usage scenarios.

This document is a management summary providing the conclusions of the complete assessment. The assessment report provides details and explanations for the determinations given in with this summary.

Analyzed Virtual Machine Monitors

Many virtual machine monitors (VMMs) with different characteristics are available. The following set of VMMs is subject to the analysis and comparison of security aspects:

- KVM provided with RHEL 5.4 using the RHEV Manager
- Xen version 3.4
- VMWare ESX Server version 4

General Security Issues

To identify the security-relevant properties of a VMM that can be used for comparing different implementations, the nature of the threats needs to be examined more closely. When a threat becomes real, an attacker uses services of the VMM to execute the threat. This implies that the attacker uses an interface provided by the VMM to the external world. The attack potential rises with the complexity and size of the attack surface of the services offered by the VMM to the external entities. This issue relates to the size of the software stack forming the VMM.

Security concerns	KVM	Xen	VMWare ESX
Size of software stack	Medium	Medium to High	N/A* Based on available information: Medium to High
Number of interfaces	Medium	High	N/A* Based on available information: High
Assurance of Development environment	Linux kernel: High QEMU: Medium	Medium	N/A* Based on information from CC evaluation: medium

Table 1: Comparison results considering security concerns

* Due to the proprietary nature of VMWare ESX Server, reliable and complete information are not considered to be available to conclude the assessment of the respective of security concerns. However, hints and partial information are presented and discussed below.

Please note that the metric defined in the assessment report for the given table does not have hard to measure properties, the results given in the table and explained in the assessment are not the sole truth.

However, the assessment provides a general idea of how the different VMM implementations relate to each other.

Security Comparison Based on Scenarios

To add another aspect to the comparison of the VMMs with respect to security behavior and mechanisms, a number of attack and usage scenarios are defined, which are used as a basis for comparison. The comparison of security characteristics of the different VMM implementations is based on the attack vectors and usage scenarios introduced in the sections below. Each attack vector and usage scenario is analyzed to identify the mechanisms provided by each VMM to either counter and mitigate the threat, or to support the usage scenario.

The following table summarizes the assurance the examined VMM implementations provide for covering the security aspects in case of a security-relevant flaw. The given assertions are relative to each other and do not provide any hint of absolute assurance for the respective VMM (as such, a value of "low" might still mean that even if a security-relevant flaw is found, it might be very hard to actually exploit it). The assessment assumes the most secure configuration possible to mitigate the outlined threats – the sections below outline these configurations.

Scenarios	KVM	Xen	VMWare ESX Server*
Assurance of protection against VM accessing unassigned resources mediated by para-virtualized drivers	Medium[1]	Low	Low
Assurance of protection against VM accessing unassigned resources mediated by full virtualization support software	Medium[2]	Stubdom: Medium Default: Low	N/A
Assurance of protection against subversion of trusted VMM software – subversion of Hypervisor	High[3]	High	Medium
Assurance of protection against Subversion of trusted VMM software – subversion of other virtual machines	Medium[4]	Stubdom: High Default: Medium	N/A
Assurance of protection against Subversion of trusted VMM software – subversion of boot process	High[5]	Stubdom: High Default: Medium	N/A
Assurance of protection against one VM causing a DoS of other VMs	High	Medium	Medium
Support for sandboxing usage	High	Medium	Low
VMs belong to different security domains	Low[6]	Medium	Low

Table 2: Assessment of coverage of security aspects based on scenarios in relation to each VMM

*The assessment of the VMWare ESX Server is based on knowledge obtained from public information. If VMWare ESX Server also includes additional mechanisms relevant to the scenarios described, the assessment might be incomplete.

Guest VM access to unassigned resources

For this attack scenario, an attacker is considered to perform access attempts to resources which are not assigned to his virtual machine. The attacker originates from within a virtual machine and can be characterized to possess the following capabilities:

- An attacker has full access to user and kernel space of one regular general purpose virtual machine.

[1] Assessment is "Medium to High" if sVirt is considered due to the fact that the SELinux separation enforcement also covers para-virtualized devices provided by the QEMU logic to the guest system. However, in newer implementations of KVM, para-virtualized devices may be provided by the Linux host system, limiting the effect of SELinux in this area.
[2] Assessment is "High" if sVirt is considered.
[3] Assessment is "High" if sVirt is considered.
[4] Assessment is "High" if sVirt is considered.
[5] Assessment is "High" if sVirt is considered.
[6] Assessment is "High" if sVirt is considered.

- An attacker has full access to all interfaces offered by the VMM to a virtual machine.
- An attacker tries to access resources that are unassigned to its virtual machine. These resources include physical, virtualized, and emulated resources.

Guest VM subversion of trusted VMM software

In order to ensure the proper separation of resources, the entire functionality of the hypervisor must be trusted. Any software component part of the hypervisor has full hardware privileges and can access any resource of the system, including physical devices, data stored on devices, etc.

If any of the mentioned trusted software components can be altered by the guest software running within a virtual machine, the software is considered to be subverted, and trust in this software component, as well as trust in the separation capability of the VMM, is undermined.

In addition to the modification of trusted software, an attacker might want to add a completely new software component. If the guest software has the ability to place another layer of software between the hypervisor and the hardware, the results of the operation of the hypervisor cannot be fully trusted, as the additional software layer can emulate a different environment and behavior of the underlying hardware. Effectively, this additional software layer adds another untrusted VMM layer.

Guest VM causes Denial-of-Service for other VMs

In addition to the proper separation of virtual machine resources, the VMM implementation also has to ensure that resources shared between the virtual machines are shared such that one virtual machine cannot dominate the use of the resource.

Usage of VMM for sandboxing

The first usage scenario covers the application of VMMs for sandboxing. Sandboxing is considered when the host or one virtual machine is used for regular day-to-day work. It is irrelevant whether a regular server logic is implemented in the one virtual machine or whether the VMM implementation is used on an end-user system. In general, only one operating system is hosted on the entire physical machine.

Guest VMs belong to different enterprise security domains

Another usage scenario applied to this analysis is the handling and protection of groups of virtual machines based on their assignments to security domains. A security domain in enterprise networks is a group of services, information and/or resources that are considered to require an equivalent level of trust when accessed.

This user scenario analyzes how the different VMM implementations can handle groups of virtual machines that belong to different security domains. This implies that resources belonging to the different groups of virtual machines can be categorized and that additional separation of the resources of the virtual machines is performed based on these categorizations.

2 Introduction

Users and administrators can choose among many different virtual machine monitor (VMM) implementations. All of these VMMs support the basic concept of virtualizing a physical computer to allow concurrent execution of multiple operating systems. By using hardware support for virtualization, almost all VMMs allow different operating systems to be supported in their guest virtual machines, including Red Hat Enterprise Linux and other Linux distributions, Microsoft Windows, different BSD variants, and others.

To support users and administrators in selecting which VMM implementation is most suitable for their needs, this analysis compares the security-relevant functionality of KVM with other VMM implementations based on attack vectors and usage scenarios. The analysis explains how the different VMM implementations mitigate potential attacks and support different usage scenarios.

Comparison of the security characteristics of different VMMs with those provided by KVM allows the reader to identify how the different VMM implementations address security-related concerns.

The comparison of security functions begins by introducing the analyzed VMMs, including the technical aspects that support analysis of the security functionality. As part of the security analysis, the interfaces provided by the VMM to either the guest virtual machines or the administrator are enumerated.

Following the introduction of the VMM implementations, the attack vectors and usage scenarios considered as a basis for this comparison are outlined. A comparison of security mechanisms based on scenarios enables the reader to compare the intended IT environment for his implementation with the environment considered for this comparative analysis.

General security concerns that must be addressed by VMM implementations are outlined in chapter 3. Chapter 4 gives a presentation of the architecture of the virtual machines and relates the implementations to the security concerns identified earlier. The attack and usage scenarios that form the basis for the core analysis as well as the analysis are given in chapter 5. The usage scenarios and attack vectors are analyzed for each VMM. The assessment explains in detail how each VMM covers the analyzed security concerns. Table 4 condenses the analysis into a broad overview. Chapter 6 gives general considerations that have an overall impact on the assessment of security mechanisms.

The goal of this analysis is to provide an assessment of current state-of-the art VMM implementations and their general functionality. The following topics would also be relevant to the discussion but are explicitly excluded to keep the size of the analysis reasonable:

- The analysis focuses on the separation mechanism for the virtual machines. As such, the hypervisor and supporting software for mediating access to resources are subject to analysis. Other security-related mechanisms, such as administration of the virtual machines and VMM, network security that might be provided by the VMM, or migration aspects in moving a virtual machine from one host to another are relevant, but are not considered here.
- The VMMs under discussion implement different virtualization schemes, including the use of processor support that allows the use of another processor state to manage certain resources vital to the VMM. A different virtualization schema also might implement complete software emulation of the processor instructions by the VMM without the use of special processor support, or it might work on an entirely different model. This analysis covers only VMM schemes with processor support enabling the VMM to offer only full virtualization to its virtual machines. Para-virtualized drivers may be provided to support a more efficient access to resources.
- VMMs implement different services for different CPU architectures. This assessment is limited to the x86 architecture, which includes 32-bit and 64-bit Intel-compatible CPUs.
- For the scope of the analysis the implementation of the memory management (either using hardware support of the latest x86 CPUs or via shadow page tables) is not considered to be relevant as the implementation is assumed to be very similar among the different VMM implementations.
- As of writing of this assessment, I/O virtualization is not yet supported by the x86 CPUs. Therefore, the assessment considers VMM implementations which handle I/O virtualization in software.

3 General Security Issues

Before starting the assessment of security aspects of virtual machine monitors, a review of the security problems and security concerns facing VMMs is useful.

A broad range of security objectives must be handled by the VMM, such as:

- Virtualizing or emulating hardware to allow general-purpose operating systems to concurrently execute on shared hardware without interference
- Preventing communication between virtual machines.
- Preventing a virtual machine from accessing resources that are not configured and assigned to that virtual machine.
- Preventing takeover of the Hypervisor, as well as the management facilities of the VMM, by hostile virtual machines.

This list of security objectives could be extended with many more entries. However, such a list of security objectives is not a useful basis to perform a security-related assessment of different VMM implementations, since all implementations share these objectives and fulfill them.

Mitigation can be achieved with different implementations, architectures, and designs. Considering that virtualization has been discussed for more than 40 years (it started when the IBM System/360 mainframe was being developed and released in the 1960s), the approach as to how to solve the fundamental security issues has been well analyzed. Therefore, the focus of this security-related assessment will not solely rely on examination of the design or architecture of the different virtual machine implementations. In addition, the following method is used.

To identify the security-relevant properties of a VMM that can be used for comparing different implementations, the nature of the threats needs to be examined more closely. When a threat becomes real, an attacker uses services of the VMM to execute the threat. This implies that the attacker uses an interface provided by the VMM to the external world. The attack potential rises with the complexity and size of the attack surface of the services offered by the VMM to the external entities. This issue relates to the size of the software stack forming the VMM.

3.1 Size of VMM software stack

Security vulnerabilities depend on the size of the code executing with privileges that is exposed to external entities, including virtual machines and network communication mechanisms. The larger the software component, the more likely the implementation will have coding errors including security-relevant flaws. This is only natural, as software development is a more or less manual process; with increasing size and complexity, humans add more and more errors into the code. These security flaws are the focal point of attackers. In essence, the likelihood of flaws that can be exploited is proportional to the size of the software stack.

In order to assess the size of the VMM software stack with respect to security issues, the software components operating with privileges must be identified. Privileged software components are considered to be those that:

- manage the assignment of and access to resources to be shared between the virtual machines,
- manage the virtual machine settings, or
- manage VMM functionality (such as updates).

Based on these characteristics, privileged software components can be identified for each VMM implementation. After identification of the privileged software components, the size (number of lines of code), as well as the complexity of the code, can be determined. The size and complexity of the code also reflect the number of services provided by the VMM.

Software components that do not have the technical ability (i.e., privilege) to interfere with the above listed operations are considered to be unprivileged. These software components can be left out of scope for the assessment. Unless otherwise noted, references to the VMM software stack throughout the remainder of this analysis consider the privileged components only.

The size of the software stack will be explained in the introduction to the architecture of the VMM implementations considered in this analysis.

The structure of the development regime also has a major impact on the number of flaws found in an implementation. This aspect is considered in section 3.3.

To allow a high-level comparison of the size and code complexity of different implementations, the following metric is used:

- Low level of size and complexity: A VMM implementation with a low level of size and complexity usually implements only separation of virtual machines, without virtualizing or emulating devices or other resources. The hypervisor is usually only a very thin layer without any additional supporting software.

- Medium level of size and complexity: A VMM implementation with a medium level of size and complexity usually provides separation of virtual machines and some level of virtualizing or emulation. The hypervisor may be larger, but the size of the supporting software that needs to be trustworthy is limited.
- High level of size and complexity: A VMM implementation with a high level of size and complexity implements separation of virtual machines and provides emulation and/or virtualization with a large hypervisor, which in turn is supported by a sizable set of supporting software that needs to be trustworthy.

3.2 External interfaces

The size of the VMM software stack is related to the number of security-relevant flaws (which also has an impact on the severity of these flaws), but a flaw is only relevant if it can be exploited. This implies also that if there is no interface to the functionality containing the issue, an attacker is blocked from abusing the issue.

Exploitation can only occur by using the interfaces the software provides to other entities, such as virtual machines or network connections. If a software stack exports many interfaces to other entities, those entities have more avenues to identify and exploit flaws.

The following types of interfaces are usually exported by a VMM:

- API calls, such as Hypervisor calls.
- Interrupts generated by hardware and processed by the VMM – note that some information to be processed with an interrupt may be set by a guest operating system.
- Processor instructions including parameters that need to be processed by the VMM.
- The mediation by the VMM of any traffic and information between resources (such as network interfaces or disks) and the guest system exposes interfaces offered by the VMM to external entities.
- Traps that are reflected to the VMM by the processor.

There is also a potential for exploiting side effects. An attacker might use an interface that has no relationship to the functionality that implements the security-relevant flaw. However, actions mediated through the interface might have an impact on the functionality containing the flaw. In this way, the issue might still be exploited. Such indirect attempts to breach security are not considered here, as they are notoriously hard to assess. Also, such issues form only a tiny fraction of the overall number of security concerns.

In essence, the likelihood of accessing flaws is proportional to the size and number of interfaces provided by privileged code to other entities.

To allow a high-level comparison of the size and number of interfaces in different implementations, the following metric is used:

- Low number and size of interfaces: A VMM implementation that is considered to have a low number and size of interfaces usually only handles a few exceptions raised by the hardware due to guest code actions. Hardly any additional interfaces are provided to external entities.
- Medium number and size of interfaces: A VMM implementation with a medium number and size of interfaces usually handles a number of exceptions raised by the hardware due to guest code actions, and provides a limited set of additional interfaces to virtualizing or emulation support.
- High number and size of interfaces: A VMM implementation that is considered to have a high number and size of interfaces handles a large number of exceptions raised by the hardware due to guest code actions, provides a large set of additional interfaces to virtualizing or emulation support, and might even grant access to its management components to external entities.

3.3 Development environment

Of course, there are also other aspects that have an impact on the number of flaws in a software stack, such as the development process. Appropriate processes for code development as well as code review, including

third party reviews, support the assurance of the proper implementation of a VMM. The application of automated code analysis tools supports the hunt for flaws during the development phase.

Also, the duration between the discovery of a flaw and the release of a fix has an impact on the security-relevance of a flaw. The faster a fix is provided, the smaller the time window for an attacker to mount a successful attack of the flaw.

Another aspect that might have an impact on the number of flaws is code maturity; that is, how long has the development organization been developing the code and how often has the code been subject to code review and analysis by external experts. As development of the analyzed VMM implementations started at about the same time, this issue is not further considered.

These development aspects will also be considered in the security analysis in the subsequent chapters.

To allow a high-level comparison of the assurance of code quality, the following metric is used:

- Low assurance: A VMM implementation that is considered to have low code quality assurance is developed in an organization that performs little or no peer review of the developed code. In addition, there is no established policy to resolve security flaws in a timely manner.
- Medium assurance: A VMM implementation with medium assurance is usually developed with a fair level of code review within the development organization, which also has an established policy for resolving security-relevant flaws.
- High assurance: A VMM implementation that is considered to have high code quality assurance must have been developed within an environment requiring substantial peer review including third party reviews, using code analysis tools, and with an established policy in place for resolving security-relevant flaws.

4 Analyzed Virtual Machine Monitors

Many virtual machine monitors (VMMs) with different characteristics are available. The following set of VMMs is subject to the analysis and comparison of security aspects:

- KVM provided with RHEL 5.4 using the RHEV Manager
- Xen version 3.4
- VMWare ESX Server version 4

The VMM implementation of Xen has been chosen as it is another Open Source implementation which is often used. In addition, proper design information is available and it implements a VMM architecture which is found by other VMM implementations, such as Microsoft's HyperV. VMWare ESX Server has also been chosen as it is one of the most widely used VMMs.

The sections that follow introduce the VMMs and explain the technical details relevant to the security comparison.

The results of comparison of the three subject VMMs considering the security concerns described in chapter 3 are summarized in the following table. (These results are derived from the analysis detailed in the sections that follow.)

Security concerns	KVM	Xen	VMWare ESX
Size of software stack	Medium	Medium to High	N/A* Based on available information: Medium to High
Number of interfaces	Medium	High	N/A* Based on available information: High
Assurance of Development environment	Linux kernel: High QEMU: Medium	Medium	N/A* Based on information from CC evaluation: medium

Table 3: Comparison results considering security concerns

* Due to the proprietary nature of VMWare ESX Server, reliable and complete information are not considered to be available to conclude the assessment of the respective of security concerns. However, hints and partial information are presented and discussed below.

Please note that the metric defined for the given table does not have hard to measure properties, the results given in the table and explained below are not the sole truth. However, the assessment provides a general idea of how the different VMM implementations relate to each other.

4.1 KVM

4.1.1 Architecture

KVM is implemented as part of the Linux kernel supported by user space code. It consists of two essential components that implement VMM functionality: the KVM Linux kernel module and QEMU for hardware emulation. The use of QEMU implies that KVM provides full virtualization to its guests and can, therefore, execute unaltered guest operating systems.

The KVM Linux kernel module implements memory management and virtual machine maintenance functionality. This kernel extension makes the entire Linux kernel the hypervisor. Virtual machines are treated by the Linux kernel as normal applications. The kernel schedules them like applications, and they can be handled like applications. As such, the process implementing a virtual machine can be seen in process listings and it can be sent regular signals, like SIGTERM.

Figure 1 depicts that from the Linux kernel perspective, the virtual machine is just another process. However, the virtual machine process has a special layout. As depicted in figure 1, the process image is split into two parts. The first part hosts a regular application logic executing in user mode (the white part of the application box in the figure) – this is used to maintain the QEMU I/O virtualization and some other small KVM-related software. The second part contains the image of the guest code, usually an operating system (the gray part of the application box), which executes in guest mode. This implies that the entire memory used for the guest operating system is allocated by the QEMU application. The kernel keeps track of which parts of the application belong to the guest operating system and which parts to the regular application.

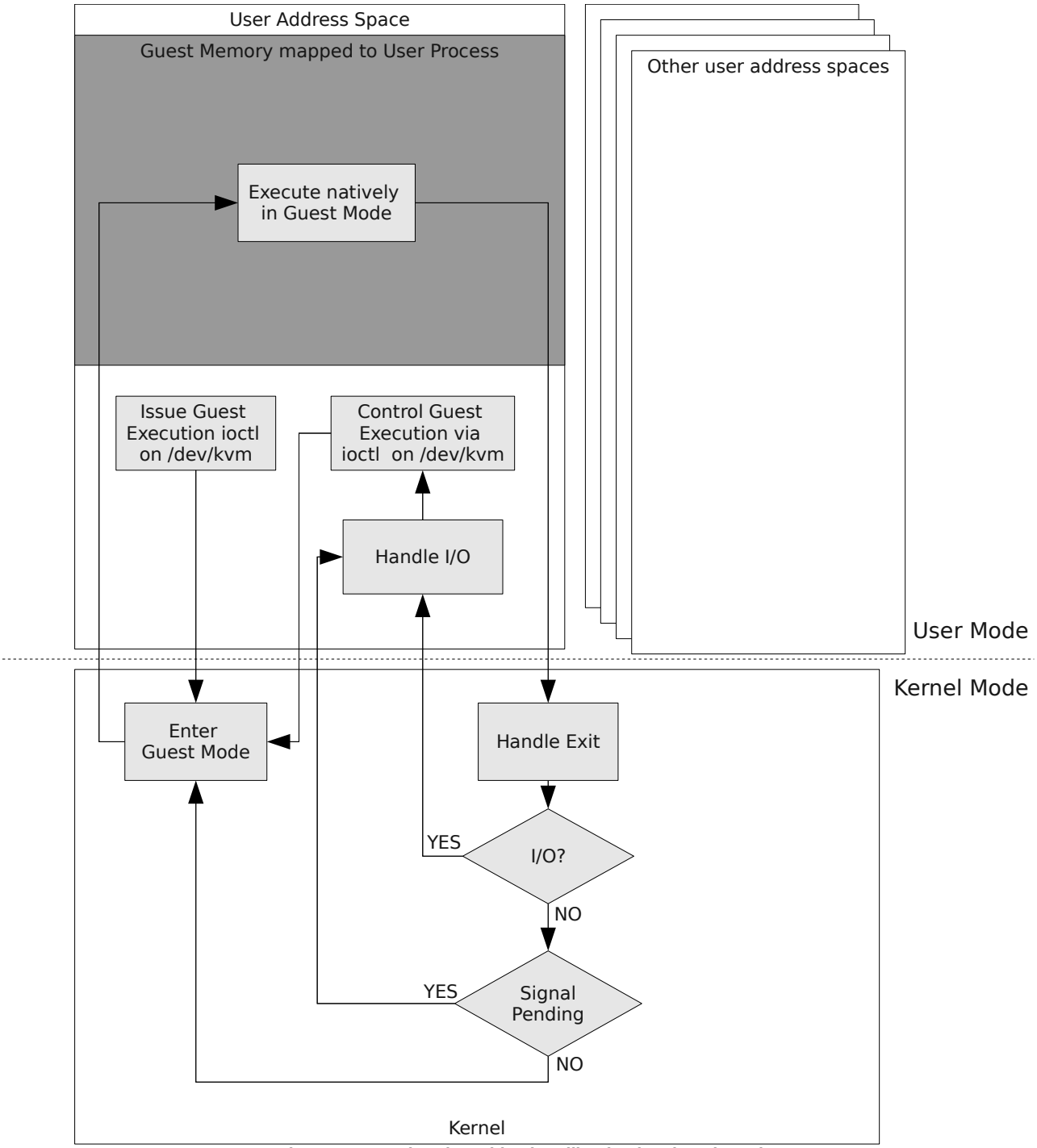

Figure 1: KVM virtual machine handling by the Linux kernel

When the kernel releases control of the CPU to the virtual machine process, it sets the processor state of the CPU to the user state when calling the regular application logic in user mode. However, when returning control of the CPU to the guest code, the CPU can be set either to supervisor state or user state, depending on the state of the CPU when the Linux kernel initially obtained control.

Figure 1 also illustrates the overall logic flow that connects the Linux kernel with the regular application logic and the guest operating system. In essence, this logic flow is an endless loop, which is performed as follows:

1. The regular application logic executing in user mode sets up the virtual machine configuration by instructing the kernel to allocate memory for the application, CPU and other resources. The kernel

sets up these resources and assigns them to the calling process. After setup is complete, the kernel is instructed to execute the guest. This phase starts the loop and is not executed again during the loop.

2. The kernel now causes the hardware to enter guest mode. If the processor exits guest mode due to an event such as an external interrupt or a shadow page table fault, the kernel performs the necessary handling and resumes guest execution. If the exit is due to an I/O instruction or a signal queued to the process, then the kernel exits to the regular application logic in user mode.
3. The processor executes guest code until it encounters an instruction that needs assistance, a fault, or an external interrupt. The processor then returns control to the VMM kernel.
4. If the kernel detects an exit of the guest code due to an I/O instruction or a signal, or until an external event such as arrival of a network packet or a timeout occurs, the kernel invokes the user mode component of the virtual machine process. The processed I/O instructions cover programmed I/O (PIO) whose implementation is not as complex as the second set of processed I/O instructions, the memory mapped I/O (MMIO). QEMU and a small extension for making QEMU KVM-aware is used to implement the I/O handling as it implements a number of emulated devices and mediates access to real resources when a device is accessed via the I/O instruction. QEMU has the capability to alter the virtual processor state to reflect the emulated I/O instruction result to the calling guest code. Once QEMU completes the I/O operation, it signals the kernel that the guest code can resume execution which is implemented with step 2 above.

In this architecture, regular applications (i.e., applications executing only in user state of the CPU and having full access to all services of the Linux kernel and, therefore, other parts of the operating system) coexist with applications that host virtual machines.

Using the RHEV Manager for administering the virtual machines of KVM using a modified libvirt management daemon (which is not identical to the current upstream libvirt-based management provided with the native RHEL 5.4 or Fedora 11 or 12), the process hosting the virtual machine does not run with root privileges. The processes implementing the virtual machines execute with the user ID of “vdsm” which is a normal, unprivileged user ID.

Unlike the RHEV Manager which is subject to the assessment of this analysis, RHEL6 will be based on the libvirt management framework with different security-related properties and mechanisms compared to the above described RHEV Manager. The following bullets provide a preview information to give the reader an understanding of the current development directions in the security area for KVM. Libvirt provided with RHEL6 restricts the capabilities of virtual machines on several layers:

- Every virtual machine process executes with the normal, unprivileged user ID of “qemu” and the group ID of “qemu”. This implies that these processes do not possess root privilege.
- To restrict the capabilities of a virtual machine process including which resources it can access, and which operations it can perform, sVirt[7] will be provided to the user of KVM. sVirt is based on the SELinux functionality provided by the Linux kernel. It aims to isolate guests using label-based mandatory access control (MAC) security policy using SELinux and introduces a pluggable security framework to the management component of the virtual machines and an SELinux implementation. The sVirt framework allows guests and their resources to be uniquely labeled. Each virtual machine and their resources is associated with a unique SELinux category (i.e implementing a multi-category system). In addition, access to resources not assigned to any virtual machine is prevented. Once labeled, rules can be applied to reject access between different guests. The strong security policy enforcement provided by SELinux implies that KVM provides an additional layer of protection from malicious attempts to exploit security flaws.
- Every virtual machine process will be placed in a dedicated cgroup. Cgroup is a mechanism of the Linux kernel to mark processes and assign certain properties to these processes – every process spawned by an already marked process will again bear the same identifier[8]. Using the device whitelist controller with the cgroup mechanism, ACLs on devices are implemented[9]. Libvirt uses cgroups with the device whitelist controller to restrict access of each virtual machine process to only

[7] http://fedoraproject.org/wiki/Features/SVirt_Mandatory_Access_Control
[8] See the file Documentation/cgroups/cgroups.txt in the Linux kernel source code tree for further information.
[9] See the file Documentation/cgroups/devices.txt in the Linux kernel source code tree for further information.

the devices assigned to this virtual machine, even though ordinary UNIX permission bits would have granted access to these devices. Please note that this mechanism only applies when the disk resource granted to a virtual machine is based on iSCSI, LVM or SANs. It does not apply to backends like regular files, NFS or others.

Management of the virtual machines is implemented with a the RHEV Manager, a daemon executing with the user ID of “vdsm”. It uses a restricted set of sudo commands to implement the execution of privileged commands, such as creating a logical volume used to provide the disk space for a virtual machine. This daemon maintains the configuration for all defined virtual machines. In addition, this daemon spawns the virtual machine processes discussed above. These virtual machine processes are controlled and can be managed, as well as terminated, by this daemon. The management daemon provides a network interface to allow administrator-facing applications to interact with the daemon.

To summarize, the KVM software stack consists of the following components, which have privileges to or manage the VMM functionality:

- The entire Linux kernel, which provides the extension to manage virtual machines, as well as the virtual machine processes.
- The QEMU and its KVM-wrapper executing as part of the virtual machine application. As this application runs with normal user privileges, the logic in the application has only restricted capabilities to interfere with the operations of the Linux host operating system.
- The RHEV Manager libvirt daemon controlling the virtual machines. As this daemon executes with a normal, unprivileged user ID, it has restricted capabilities to interfere with the operations of the Linux host system. However, it executes with the same user ID as the virtual machines and can therefore directly interfere with the execution of all virtual machines.

4.1.2 Exported interfaces

This section analyzes the interfaces offered by the privileged components of KVM.

The most important set of interfaces are those offered to the guest code, as this code must be considered untrustworthy by KVM. As the guest code executes within an application, the most interesting question is whether this guest code is allowed to use the standard interfaces exported by the Linux kernel, namely system calls and exceptions. The simple answer is that the Linux kernel does not allow any system call to be used by the guest code. In addition, exceptions are handled as discussed in section 4.1.1; that is, I/O exceptions are relayed to the QEMU code of the virtual machine application and other exceptions are handled by the Linux kernel. In addition to the standard Linux kernel interfaces, a search for interfaces commonly exported by a hypervisor must be conducted. Common interfaces of hypervisors are as follows:

- Hypercalls: Hypercalls have a very similar logic to system calls. Only two hypercalls are exported by KVM: one does not perform any operation, while the second hypercall allows the guest to perform some MMU interaction with the host.
- Interfaces to para-virtualized devices/resources: Support for para-virtualized devices and resources is under development, with a limited set of para-virtualized devices and resources available.
- I/O instructions: I/O instructions are intercepted by the Linux kernel and forwarded to QEMU for handling. Therefore, these instructions are considered to be the external interface of QEMU although technically they are intercepted by the Linux kernel.
- IOCTLs accessible to the user space (such as QEMU) to configure virtual machines and interact with KVM.

In addition to the Linux kernel – the hypervisor of KVM – interfaces, the interfaces implemented by the full device virtualization support offered with QEMU present another set of relevant interfaces, as QEMU is considered to be a privileged software component in the KVM virtualization software stack, as discussed in section 4.1.1. The interface offered by QEMU to the guest code is established with I/O instructions. The number of I/O interfaces depends on the number of emulated devices. A list of emulated devices can be

obtained from the man page qemu(1)[10]. As QEMU might invoke real resources based on the received I/O instructions, QEMU also must process the responses from the invoked resource. As the responses can hardly be influenced by the calling guest code, this QEMU interface is not further considered.

Please note that with para-virtualized device support, the size of the code needed for providing access to the resource/device is much less than for fully virtualized devices/resources. In addition, for KVM, the para-virtualized backend is implemented as part of QEMU and not in the host Linux kernel as one would expect considering other hypervisor implementations. This implies that for the para-virtualized resources/devices, the executed privileged code is less than the full simulation code. However, as both code paths are provided to the guest code, the number of interfaces and the code size increases with the development of new para-virtualized devices/resources. KVM for RHEL5.4 provides only one para-virtualized device backend: a block device driver.

The next software component to be analyzed for interfaces is the libvirt management daemon. It provides a network interface as well as a UNIX domain socket; both interfaces can be enabled or disabled with the configuration of the management daemon. The network connections may be protected using TLS/SSL, SASL and Kerberos. The UNIX domain socket can be protected with the proper permission settings on the UNIX domain socket device file. In addition, read-only and read-write permissions can be configured when using UNIX domain sockets. As this interface should be tightly controlled either by using the provided access control mechanisms or by making the daemon accessible from a dedicated administrative LAN only, this assessment assumes that this interface is well protected so that it cannot be accessed by any untrusted entities, either using the guest code or using the network connections to the guest code that are mediated by the host operating system. This implies that the external entities are considered to have no access to this management daemon, which results in the conclusion that the security aspects and potential security flaws of this management daemon are irrelevant for this discussion.

4.1.3 Assessment of security concerns

With regard to the security concerns enumerated in chapter 3, the KVM implementation can be characterized as follows.

Code handling, virtual machine instantiation, and exceptions are implemented in one Linux kernel module supported by an additional kernel module implementing the CPU-specific logic. The size of the code is not overly large (about 650kB of C code). However, for the proper operation of the entire VMM logic, the remainder of the Linux kernel is also important, as it schedules the virtual machines, performs memory management, implements the device drivers for the physical devices, and provides other essential services. In addition, the Linux kernel cannot be separated from the KVM kernel modules at runtime, as both parts run in the same address space with the same hardware privileges of executing in the hypervisor mode of the processor[11]. As such, both the Linux kernel and the KVM kernel modules rely on each other for ensuring that the security objectives for a VMM are achieved. This results in the finding that the entire Linux kernel must be considered part of the software stack. As QEMU with the KVM wrapper logic is also part of the software stack but execute with a normal unprivileged user ID, potential security deficiencies can only have a very limited effect on the Linux host system. Nevertheless, as all QEMU instances for different virtual systems as well as the libvirt management daemon execute with the same user ID, potential security issues may have an impact on the proper isolation of the virtual machines from each other. In essence, the size of the software stack is considered to be medium level compared to other implementations.

The number of interfaces offered to external entities covers a very limited set implemented by the KVM kernel modules, as para-virtualized device driver is not yet fully implemented. On the other hand, the interfaces provided by the full virtualization logic provided with QEMU are large in size, as multiple different emulated devices are offered. This results in the determination that the number of interfaces exported to external entities is considered to be in the medium range compared to other implementations.

Considering the development regime applied to the development of KVM, the two components relevant for our assessment are considered separately:

---

[10] Please note that for emulating a VGA device and for supporting the boot sequence of the guest, an additional BIOS is emulated by code that is technically maintained outside of QEMU, but is logically considered to belong to QEMU. Thus, every time QEMU is referred to, this BIOS is also implied in this reference.

[11] The hypervisor mode is also called the root mode on Intel-based CPUs

- KVM Linux kernel module and KVM wrapper user code development: The kernel module development is considered to receive a fairly high amount of review from peer developers inside Red Hat, as well as from the Linux kernel development community, as the kernel modules are part of the upstream Linux kernel. In addition, regular analysis of the Linux kernel with Coverity[12], as well as performance of third-party reviews, supports the code quality and reduces the number of flaws. The size of the KVM wrapper for QEMU in user mode is considered negligible, as it is small and does not implement much logic.
- QEMU development: The QEMU development is a standard open source project, but does not receive as much attention as the Linux kernel. Development rests with a small development team supported by occasional review by Linux distribution vendors.

The Red Hat approach to monitoring sources for bug reports and maintaining a dedicated security team supported by other security teams from other Linux vendors to handle bug fixes is considered to be fully supportive to ensuring that once a security-relevant flaw has been found, a fix will be released in an appropriate time frame. Comparing this approach to practices of other vendors and implementations, there is a high level of code assurance for the Linux kernel and the KVM kernel modules. Considering QEMU development, it does not receive much attention from other parties and few code inspection tools are utilized, but Red Hat provides the same efficient security flaw fix process as for the Linux kernel, and this results in the conclusion that overall, there is a medium assurance of code quality for the QEMU component.

4.2 Xen

4.2.1 Architecture

Xen is a VMM implementation based on a small separation kernel, the hypervisor, which performs the separation of different virtual machines. The goal of the separation kernel is the isolation of the virtual machines in terms of preventing access to resources belonging to one virtual machine — mainly memory, CPUs, and virtual machine scheduling — by another virtual machine. The hypervisor operates in a privileged CPU mode, which allows the hypervisor to protect its operation from interference by any guest operating system.

In Xen terminology, every virtual machine is called a domain, or dom in shorthand.

As the hypervisor separates only a subset of available resources, it is supported by an additional software component that mediates access to additional resources by the domains. This additional software component is a special Linux operating system that resides in a privileged domain called Dom0. Using the Dom0 logic, access to resources like network devices or block devices (i.e., hard disks) is established. The hypervisor grants Dom0 special rights to access physical I/O resources. Dom0 is used exclusively for mediating access to resources and the administration of Xen, as it also provides management tools and interfaces. Dom0 is the very first virtual machine that is automatically instantiated after the Xen hypervisor is loaded.

Any other virtual machine is provided for general purpose use and does not possess any special privileges with respect to Xen. These general-purpose virtual machines are called DomU. Xen defines two types of virtual machines:

- PV guests: PV guests (para-virtualized guests) contain support for inter-operation with Xen. Inter-operation covers aspects where the para-virtualized guests use special interfaces provided by either the hypervisor or Dom0, such as virtual memory management or device drivers. In addition, the guest operating system must be aware of the fact that it has not full control over the physical machine. PV guests are not further considered in this assessment.
- HVM guests: HVM guests (hardware virtual machine guests) are unmodified guest operating systems that require full virtualization of the hardware. Similar to KVM, Xen provides full virtualization via QEMU. In addition, HVM guests may use para-virtualized device drivers to use the Xen-offered resources more effectively.

Figure 2 depicts the architecture of Xen.

[12] See http://scan.coverity.com/rungAll.html

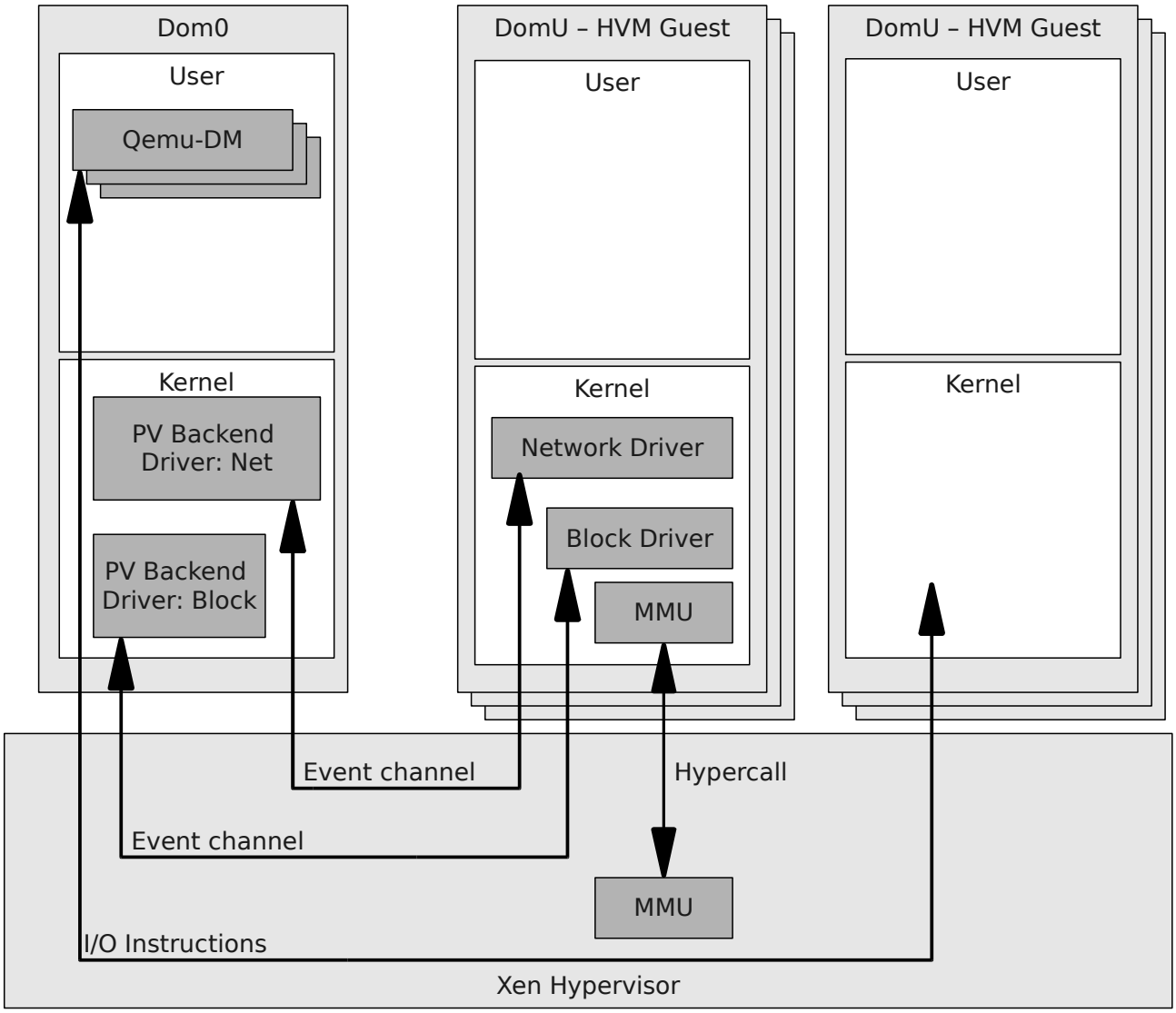

Figure 2: Xen domains and communication relationships

Figure 2 shows the Dom0 and the DomU virtual machines. In addition, this figure documents the communication relationships between the domains, Dom0, and the hypervisor.

As already indicated, HVM guests with para-virtualized drivers cooperatively interact with Xen to ensure efficient resource access and as such. This implies that these guests access their resources much faster than HVM guests with native device drivers. The cooperation between para-virtualized device drivers and Xen is twofold:

- Concerning memory management and registering of callback functions that allow the hypervisor to inform the guest kernel about events, the guest uses hypercalls, which are conceptually equivalent to system calls exported by the hypervisor. The hypervisor is responsible to react to the requests of the guest.
- For access to other resources (figure 2 depicts access to a network resource and a block storage driver), the vpara-virtualized device driver implements a frontend that effectively makes the device known to the guest operating system and translates access requests from the OS into requests that can be sent to the backend driver counterpart. That backend driver resides in the Linux kernel of Dom0 and interfaces with the physical device driver of the Linux kernel to perform the requested action. As only one instance of each backend driver is established in Dom0, the backend driver must ensure that it keeps track of where the request originates and whether the requester is allowed to

perform the request (i.e., whether the requester is granted access to the requested resource), and must track that the response is sent back to the requester. The hypervisor provides a bi-directional communication channel between the DomU and the Dom0, called an event channel implemented supported with shared memory on top of the Xenbus inter-virtual machine communication channel. The event channel is implemented with ring buffers maintained in memory that is shared between the tow communicating domains by the hypervisor. Event channels operate as asynchronous delivery systems that are also used to deliver interrupts.

HVM guests with native drivers are implemented without the knowledge of Xen. As such, Xen must emulate devices for all device drivers available in the guest operating system. When the guest operating system performs I/O operations on a device, Xen intercepts the request and forwards it to Dom0. In Dom0, one process per HVM guest hosts QEMU. QEMU performs the emulation of the device and translates requests into access requests to real devices offered by the Linux operating system in Dom0. To boot an HVM guest, Xen places a virtual BIOS into every DomU, which ensures that the guest operating system finds all resources and instructions it expects from a boot procedure on native hardware.

The current implementation of Xen allows placement of QEMU support into a DomU, where QEMU executes on a dedicated mini-operating system that implements the para-virtualized frontend drivers for the exported devices. These para-virtualized drivers in turn access the respective para-virtualized backend drivers of Dom0. In this configuration, the special DomU, also called Stubdom, operates like a para-virtualized guest that can be the recipient of I/O instructions. Figure 3 shows this configuration option. One Stubdom can service only one general-purpose guest domain, but allows flexible configurations. With the configuration of QEMU executing in Dom0 for HVM guests, QEMU executes as root daemon. To prevent abuses from potential QEMU exploits, the use of Stubdoms effectively separates all QEMU instances from one another, reducing the size of the trusted software stack of Xen – please note that the privilege QEMU is considered to possess results from the fact that QEMU executes as root in Dom0. But Dom0 is also privileged, which means that if QEMU were to have a security flaw, that flaw could be exploited to use those privileges. Such an exploitation cannot occur if QEMU is executed in a dedicated DomU.

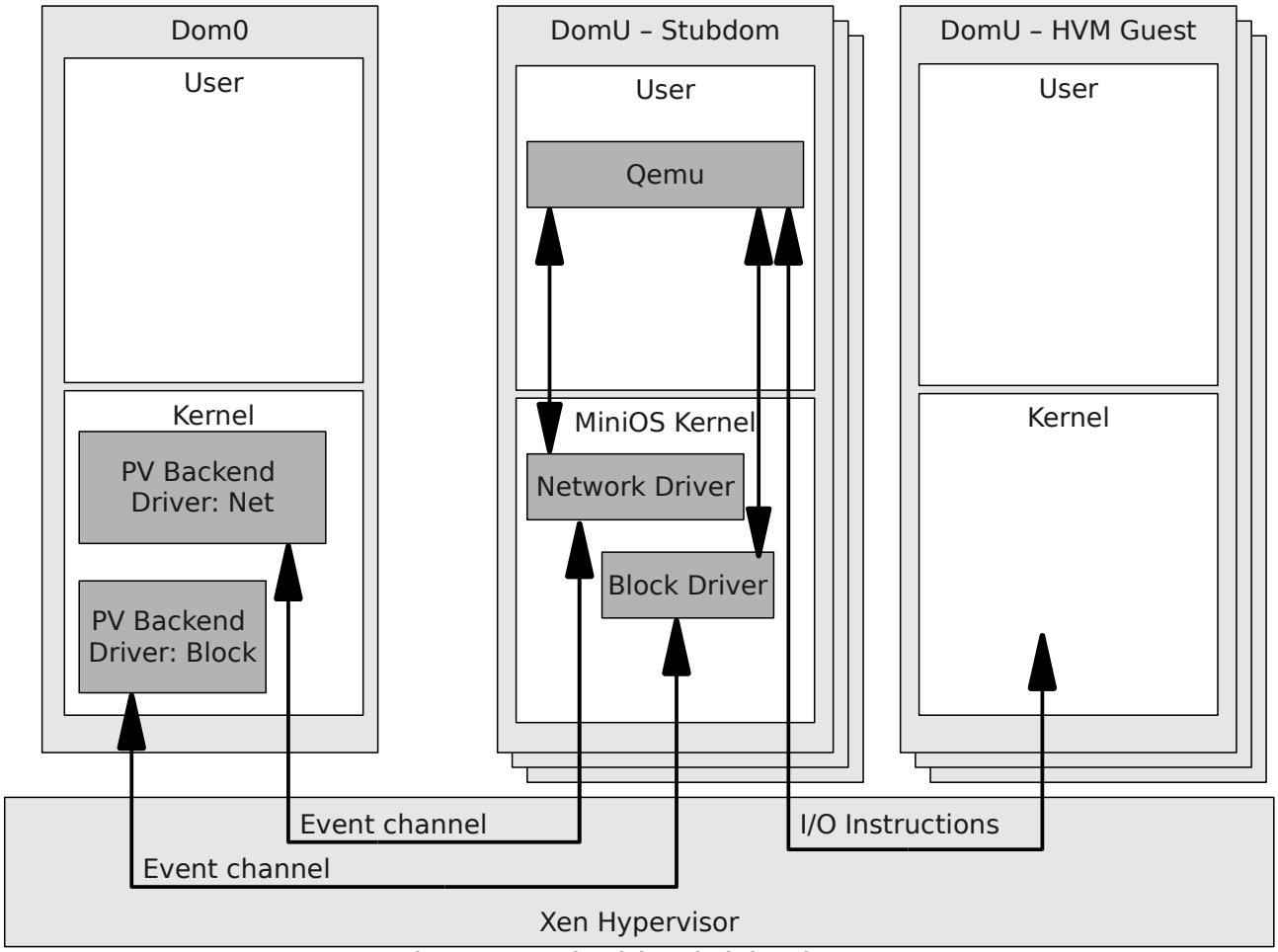

Figure 3: Xen and Stubdom - isolation of QEMU

The Xen hypervisor implements an access control schema based on the Flask logic implemented in SELinux as well as sHype developed by IBM Watson Research Lab. The implementation is based on the ACM security call backs. Both access control mechanisms allow the specification of rules governing multi-level access control, multi-category access control, role-based access control, as well as type enforcement can be implemented. Similar to SELinux, a policy to define the rule set enforced by Flask is required. This rule set is loaded by Dom0 into the hypervisor. Currently, only a rudimentary policy is available; the policy might be extended in the future, but is not further considered for this assessment. Also for sHype, a rule set needs to be defined and implemented as no default rule set could have been identified.

Dom0 hosts a registry daemon, xenstored, which maintains the configuration and state information for each domain. This registry daemon is accessible from every domain via the Xenbus maintainted by the hypervisor. Any guest can read any part of the store, but only if it has permission to do so. Dom0 may read or write anywhere in the store, regardless of permissions, and permissions are set up by the tools in Dom0, or by xenstored when it first starts up.

To configure Xen, a networked administration daemon executes on Dom0. Command-line tools that use the network interface to configure the domains are provided. The daemons execute as root and, therefore, must be trusted to perform the requested operations.

To summarize, the Xen software stack consists of the following components that have privileges to interfere with or manage the VMM functionality:

- The Xen hypervisor provides the separation kernel on which rests proper memory management and inter-domain communication. As such, it is the main trusted component.

- The entire Linux kernel executing in Dom0, which ultimately mediates access to the physical resources. In case of the use of para-virtualized backend drivers, these drivers share memory with the rest of the kernel and utilize the Linux kernel physical device drivers. In addition, if QEMU support in Dom0 is configured, the Linux kernel is needed to allow the user space QEMU process to access the requested hardware. Therefore, the entire kernel is relevant to maintain the secure operation of the VMM.
- If HVM is configured and QEMU support in Dom0 is utilized, then QEMU also needs to be trusted, as it executes as root. As such, it has the potential to interfere with the operation or configuration of Dom0 and implicitly with any other domain. If a security flaw were to be embedded in QEMU, it could be used to circumvent protection mechanisms configured in Dom0.
- The xenstore daemon provides the configuration information for every domain and is consulted by the different components of Xen.
- The management daemons executing in Dom0 controlling the virtual machines. As the daemon runs with root privileges, the logic in the application might interfere with the entire operation provided with Dom0.

4.2.2 Exported interfaces

In addition to examining the architecture of Xen, identification of the interfaces offered by the privileged components of Xen is vital.

The central component is the Xen hypervisor. The hypervisor provides the following types of interfaces:

- Hypercalls: Hypercalls are equivalent to system calls exported by the hypervisor to guest operating system kernels. With these hypercalls, guest operating systems can directly request services from the hypervisor. Such services include memory management cooperation with the hypervisor and registration of event handlers for the event channel. A number of available hypercalls are restricted to Dom0, where the hypervisor verifies that the calling domain is Dom0. Such privileged hypercalls include the domain configuration at runtime (domctl), system parameter settings (sysctl), access control settings (acm_op), and others. Although only 39 hypercalls are implemented, many of these hypercalls contain de-multiplexing code, where a hypercall is used to trigger many different mechanisms selected with a command flag supplied as part of the hypercall. As such, architecture-wise, the number of actual hypercalls (i.e., individual mechanisms that can be invoked) is significantly larger than the number of hypercall interface functions exported to the domains. For the assessment of security impact, hypercalls exported only to Dom0 are considered to be inaccessible to untrusted entities, as Dom0 must be considered to be appropriately protected. Therefore, only a subset of all implemented hypercalls is considered relevant for further assessment of security issues.
- Event channels and shared pages: Event channels supported by shared pages enabling the bulk data transfer establish communication between two domains based on shared memory. The Xen hypervisor establishes the channel, but does not act upon the information flowing through that channel. Therefore, this channel is considered to be the external interface of the para-virtualized backend drivers discussed below.
- Xenbus: The Xenbus is used for configuration negotiation and other meta information transport. It is used to notify other domains about events. The data transport between domains is left to the event channels with the shared pages.
- I/O instructions: I/O instructions are intercepted by the Xen hypervisor and forwarded to QEMU for handling. Therefore, these instructions are considered to be the external interface of QEMU, although technically they are intercepted by the hypervisor.

The para-virtualized backend drivers export their interfaces to untrusted entities via event channels. As the para-virtualized frontend drivers executing in the guest operating system kernel cannot be trusted to operate correctly or to be used at all when accessing the event channels, the current assessment relies on the technically-enforced interface that separates external entities from the trusted code. Therefore, the protocol between the frontend drivers and the backend drivers flowing through the event channel is considered to be

the external interface. The following backend drivers are currently available and implement external para-virtualized interfaces to domains:

- Block-device backend driver: Allows the kernel to export its block devices to other guests via event channels.
- Block-device tap backend driver: An alternative to the block back driver that allows VM block requests to be redirected to user space through a device interface. The tap allows user space development of high-performance block backends, where disk images may be implemented as files, in memory, or on other hosts across the network. This driver can safely coexist with the existing block-device backend driver.
- Network-device backend driver: Allows the kernel to export its network devices to other guests via event channels. As a sub-category, a special pipelined transmitter is available, as well. In addition, a special network-device backend driver accelerator for Solarflare network interface cards is available. Also, a network-loopback driver backend is configurable.
- PCI-device backend driver: Allows the kernel to export arbitrary PCI devices to other guests.
- TPM-device backend driver: This driver provides access to the virtualized TPM support provided by the underlying hardware.
- SCSI backend driver: Allows the kernel to export its SCSI devices to other guests via a high-performance shared-memory interface.
- USB backend driver: Allows the kernel to export its USB devices to other guests.

In addition to the Xen hypervisor interfaces, the interfaces implemented by the full device virtualization support offered with QEMU present another set of relevant interfaces, as QEMU is considered to be a privileged software component in the Xen virtualization software stack, as discussed in section 4.2.1. The interface offered by QEMU to the guest code is established with I/O instructions, but QEMU also may have direct access to the calling domain's memory, in case memory operations need to be performed. The number of I/O interfaces depends on the number of emulated devices. A list of emulated devices can be obtained from the man page qemu(1). As QEMU might invoke real resources based on the received I/O instructions, QEMU also must process the responses from the invoked resource. As the responses can hardly be influenced by the calling guest code, this QEMU interface is not further considered.

The xenstore daemon hosting the configuration of the different virtual machines can be accessed from DomU via the XenBus protocol mediated through the Xen hypervisor. XenBus provides a bus abstraction to support the establishment of event channels for para-virtualized drivers. In practice, the bus is used for configuration negotiation, leaving most data transfer to be done via an inter-domain channel composed of a shared page and an event channel.

The set of privileged software covers the administrative aspects. Network interfaces are provided by the management daemon to allow remote management of Xen. However, as administrators have inherent privileges to modify the configuration of domains and security properties of Xen, this assessment assumes that the external interfaces provided with the management facilities are technically restricted. In addition, any shell-level access to Dom0 is considered to be restricted. Only trusted administrators are assumed to have the potential to access these interfaces. This may be achieved by use of a dedicated network interface, where the administrative facilities listen exclusively. This network interface is restricted to an administrative LAN. Therefore, the interface and the administrative mechanisms in general and shell-level access to Dom0 are considered to be out of reach for untrusted external entities and, therefore, out of scope for this assessment.

4.2.3 Assessment of security concerns

With regard to the security concerns enumerated in chapter 3, the Xen implementation can be characterized as follows.

The software stack that operates in privileged mode with respect to security-relevant mechanisms contains the Xen hypervisor as the core of the virtualization mechanism. The hypervisor is considered to be of mid-range size, as its code size is about 13MB. It cannot be regarded as a true minimalistic separation kernel,

which would only require a few thousand lines of code – please note that a sizable part of the software provides support for para-virtualized guests where no hardware-based virtualization support is available which is not considered here and therefore this functionality is not considered to be accessible. On the other hand, it is not a massive kernel, like the Linux kernel for KVM. In addition, the entire Linux kernel in Dom0 is considered to be trusted, as it always mediates access to the physical devices, regardless of whether the access request is made through QEMU or through the para-virtualized backend drivers. The entire kernel is considered to be security sensitive, as the entire kernel operates the same address space with the same privileges to access resources. Therefore, any component within the Linux kernel must be trusted. In addition, the xenstore daemon is security sensitive, as it is accessible from every domain, and stores and manages the configurations of every domain. This daemon is consulted by the hypervisor to set up the proper parameters for each domain. Depending on the configuration, QEMU is also considered to be part of the trusted software stack if HVM guests are configured without a Stubdom. When considering the entire stack with QEMU, it is much larger than the software stack needed for KVM and is, therefore, considered to be medium-to-high level compared to other implementations.

On the other hand, if the HVM domains are configured to execute QEMU in a Stubdom or there are no HVM domains, QEMU can be considered as not relevant for maintaining security, as it cannot interfere with the security implications of Xen. When excluding QEMU from the software stack, Xen can be considered to be of similar size to KVM, as QEMU needed by KVM is a bit larger than the Xen hypervisor source code which is not present in KVM. Therefore, without QEMU, the size is considered to be medium level.

The number of interfaces offered to external entities covers a set of hypercalls implemented by the Xen hypervisor. In addition, para-virtualized backend drivers in the Linux kernel of Dom0 provide another sizable set of interfaces, as there are several of those para-virtualized drivers available. On the other hand, the interfaces provided by the full virtualization logic provided with QEMU are large in size, as multiple different emulated devices are offered. Finally, the xenstored interfaces also must be considered. This results in the conclusion that the number of interfaces exported to external entities is considered to be in the high range compared to other implementations. Therefore, the set of interfaces available to one particular domain is considered to be at a high level.

With regard to the development process applied to development of the different components of the Xen software stack, the components relevant for our assessment are considered separately:

- The Xen hypervisor is developed as an independent open source project. A large number of commercial vendors support the development. Also, the IBM Watson Research Lab reviewed the hypervisor code and contributed the sHype security enhancement. However, it is unknown whether code analysis tools are employed on a regular basis. As such, the hypervisor is considered to be well reviewed by many parties, providing a medium-to-high assurance of code quality.
- The Xen management tools including the xenstore daemon, are also part of the development project, together with the Xen hypervisor. However, as they are not regarded as the most critical and complex logic and also reside in Dom0, a medium assurance of code quality can be assumed, considering that the developer base is much smaller, reviews from third parties are limited and it is also unknown whether code analysis tools are employed.
- The Xen Dom0 Linux kernel patches unfortunately are not part of the upstream Linux kernel and as such, are not covered by the same code standards considered for the kernel. Independent of the Linux kernel community discussions about these patches, they have some drawbacks for administrators: they are accessible for older kernels only (2.6.18 is the latest kernel). As such, security fixes for the Linux kernel must be backported to this old kernel. Some distributors no longer support such backporting. Even though these out-of-tree patches can be considered to have good code quality, the application of them to an old Linux kernel does not support the maintenance of a secure system. Please note that this issue applies to the Dom0 Linux kernel only, but as discussed above, it is considered part of the trusted code base. Therefore, based on this patch issue only, a low-to-medium assurance of code quality must be assigned. Note that there are renewed efforts underway to push the Dom0 patches into the upstream Linux kernel.
- Dom0 Linux kernel development is considered to receive a fairly high amount of review from peer developers inside Red Hat, as well as from the Linux kernel development community. In addition, the Linux kernel is regularly analyzed with Coverity, and third-party reviews are performed. Both of these practices support the code quality and reduce the number of coding errors including security flaws.

Apart from the issue around the Dom0 patches discussed above, the Linux kernel code quality must be considered high.

- QEMU development is a standard open source project, but does not receive as much attention as the Linux kernel. Development rests with a small development team supported by occasional review by Linux distribution vendors.

Although the support by Red Hat for Xen may be reduced, the current RHEL distribution still includes Xen. This implies that Red Hat provides bug fixes for Xen at the same level as for other parts of the distribution. The Red Hat approach to monitoring sources for bug reports and maintaining a dedicated security team to handle bug fixes is considered to be fully supportive to ensuring that once a security-relevant flaw has been found, a fix will be released in an appropriate time frame. When compared to the practices of other vendors and implementations, this approach provides additional assurance of code quality and security flaw remediation.

Based on the enumeration of the different software components, an overall quality of medium can be awarded. If resolution of the issue around the Dom0 patches results in integration of these patches into the upstream kernel, a medium-to-high level of assurance is considered.

4.3 VMWare ESX Server

The analysis of VMWare ESX is limited because it is proprietary software and, therefore, limited design information is available. This assessment covers as much information as possible. In case of insufficient available information, the assessment will mark this fact appropriately.

Please note that the assessment focuses on the VMWare ESX implementation, which differs from the ESXi version by adding a Linux-based. ESX provides a Linux-based service console for accessing and managing the virtual machines. This service console is greatly minimized to provide only small shell access and network access for management components that are executed on remote administration servers on ESXi. The hypervisor and the concept of managing virtual machines, however, are identical between the ESX and ESXi products.

4.3.1 Architecture

Initially, all virtualization products originating from VMWare implemented a virtualization technique called binary translation. Binary translation means that the hypervisor intercepts the entire instruction stream of all guest operating systems and translates them to real processor instructions. As part of the translation, the hypervisor verifies that the instruction parameters do not violate the bounds defined for the virtual machine in which the guest executes. For separating memory, the hypervisor uses the segmentation engine of the processor. As such, the hypervisor executes in the ring 0 privilege level of the x86 CPU and moves the guest operating systems into the ring 1 privilege level. The architecture of the hypervisor implementing the binary translation is considered very complex. As VMWare ESX server implements support for the Intel VT-x, as well as the AMD-V extension, the current assessment disregards the binary translation support. Although still implemented, the binary translation is only used on CPUs where the mentioned extensions are available. This also implies that the administrator does not configure binary translation as mandatory.

VMWare ESX consists of a hypervisor separation kernel called vmkernel, whose primary function is to ensure the separation of the virtual machines from each other. To ensure the separation of the virtual machines, the hypervisor mediates access to a set of resources that are exclusively assigned to virtual machines. The following components are contained in the hypervisor[13]:

- Resource scheduling including virtual memory management ensures that the memory allocated to virtual machines is not overlapping. In addition, this component ensures that virtual machines are scheduled on the available physical CPUs.
- The user world API implements a subset of a POSIX-compliant interface for supporting services. The supporting services are implemented outside of the hypervisor and execute on top of the hypervisor, similarly to the virtual machines.

[13] See http://www.vmware.com/files/pdf/ESXServer3i_architecture.pdf

- Storage support and device drivers for different storage hardware solutions are implemented in the hypervisor. As such, the storage requirements of the virtual machines are covered.
- Networking support including the respective physical device drivers for network hardware is also implemented in the hypervisor. As part of the networking support, the hypervisor provides virtual Ethernet adapters to virtual machines and implements an Ethernet switch for these virtual adapters.

Figure 4 shows the structure of the hypervisor and its supporting software.

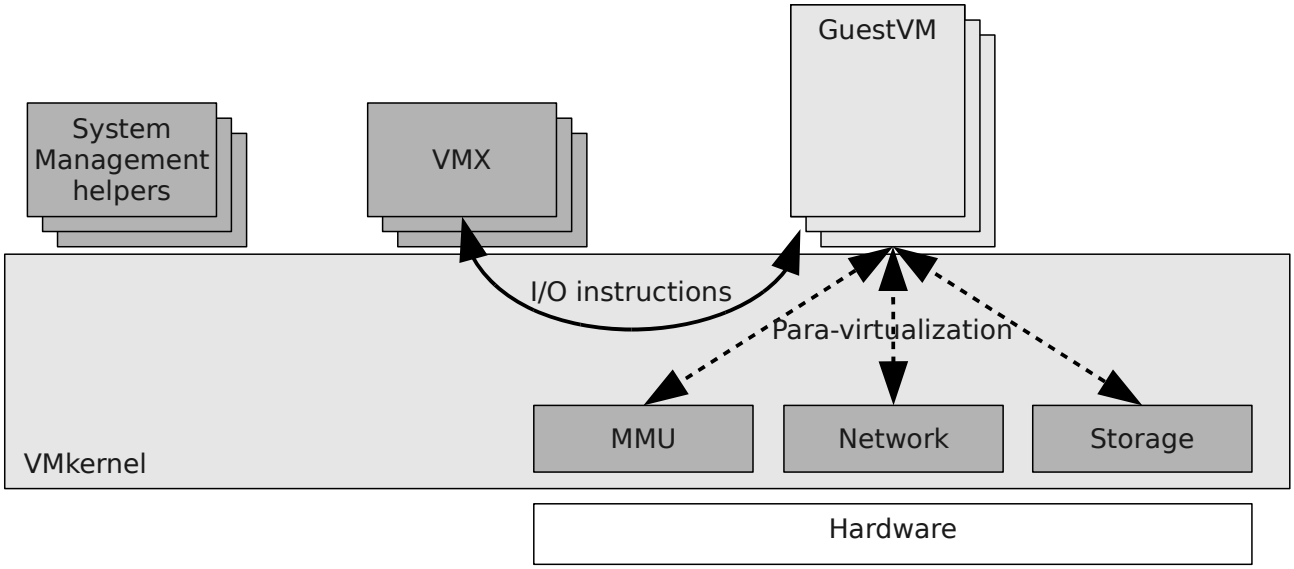

Figure 4: VMWare ESX components and communications

In addition to the hypervisor, VMWare ESX provides additional services that execute outside of the hypervisor along with the virtual machines. These additional services are considered as “user world”. The user world provides services like remote management access via the Common Information Model (CIM) broker, a small shell-like management console (the DCUI), SNMP support, Syslog support, VMX helpers, and others. One VMX helper is initiated per virtual machine and provides the emulated hardware environment to allow the execution of full virtualized guests. It is unclear as yet how strongly the VMX instances are separated from each other. To support the booting of guest operating systems, a virtual BIOS is placed in the virtual machine memory to allow operating systems to find the same kind of environment as on a native hardware.

Considering the description of the VMI hypercall interface specification[14] published by VMWare, the hypervisor provides a number of hypercalls for para-virtualized guests. As para-virtualized guests are not considered, it is unclear how many of these interfaces are still available to hardware-based virtual machines and using para-virtualized drivers. Para-virtualized device driver support is provided by the hypervisor for memory management, I/O devices, and other calls. As no further information about the internal design is available, a reliable picture of how para-virtualized device drivers are handled cannot be given. However, it may be assumed that the para-virtualized driver support is provided by the hypervisor kernel, which also implements the physical device drivers.

To summarize, the VMWare ESX software stack consists of the following components that have privileges to interfere with or manage the VMM functionality:

- The vmkernel hypervisor is the core of the virtualization mechanism, as it maintains the virtual machines and enforces the separation of them by managing the resources assigned to the virtual machines.
- The VMX helper implements full virtualization support by emulating the devices accessible by the respective virtual machines. As these VMX helpers have access to the user world API, it is assumed that they also potentially have access to administrative interfaces provided by the hypervisor. As

[14] See http://www.vmware.com/pdf/vmi_specs.pdf

such, the VMX helper must be considered trustworthy to ensure proper virtual machine separation. These helpers use Linux drivers provided with VMKLinux to implement the connection to devices.

- The system management helpers are considered to be part of the trusted software stack, as they are provided for performing administrative tasks. As such, these software components have the potential to modify the virtual machine configuration and, therefore, the separation functionality of VMWare ESX. The service console is a limited distribution of RHEL5 and "provides an execution environment to monitor and administer the entire ESX host"[15]. As the service console administers the entire ESX host and therefore has privileged access to virtual machines and their configurations, at least the privileged components of the service console must be considered trustworthy, which include the Linux kernel and all applications that execute with root privilege. Due to the proprietary nature of VMWare, the interfaces to the vmkernel, VMX instances or other parts of the ESX product are unknown. Therefore, it is unclear whether even non-root application may have the potential to interact with other, privileged parts of the ESX software stack. If this would be the case, even these non-root applications would need to be added to the trusted software stack.

4.3.2 Exported interfaces

A number of different types of interfaces are exported to external entities from components of the trusted software stack. The first interface is the set of hypercalls provided by the hypervisor. It is assumed that the hypercalls follow the VMI specification defined by VMWare. However, it is unclear whether the documented VMI interfaces are all technically-accessible hypercalls or whether only a subset of the VMI calls are implemented in case of hardware-based virtualization and para-virtualized device drivers. A hint may be the data structure Vmnix_VTable defined in the Linux kernel source code for the service console in include/linux/esxsc.h which lists callback functions for various mechanisms which seem to be implemented by the vmkernel. But again, it is unclear whether these interfaces can be triggered from virtual machines.

In addition, for full virtualized guests, the hypervisor must trap the I/O-related processor instructions and forward them to the respective VMX instance for further operation. This VMX instance mediates access to the resources configured for the calling virtual machine.

The system management helpers provide network access to manage the virtual machines. The service console, a CIM broker, and other tools are provided to allow remote administration interfaces to manage the virtual machine settings. For this assessment, however, the management interfaces are considered to be restricted to authorized administrators, where other external entities have no technical ability to interact with these interfaces. As such, this interface is considered to be irrelevant for the discussion. Furthermore, as the system management helpers are inaccessible by untrusted external entities, potential security-relevant flaws in their functionality cannot be abused. Therefore, the system management helpers are out of scope for the reminder of this assessment.

However, the following interfaces are unclear from the documentation provided around VMWare ESX. As these issues are unclear, they are not further considered:

- It is unclear whether the user world API is technically accessible from the virtual machines. If so, this API is also considered to be an external interface.
- The interfaces between the virtual machines and the para-virtualized devices are not fully defined. Therefore, the lines between the virtual machines and the device drivers in the hypervisor are marked with a dotted line in figure 4, as this architecture cannot currently be verified.

4.3.3 Assessment of security concerns

With regard to the security concerns enumerated in chapter 3, the VMWare ESX implementation can be characterized as follows.

The software stack that must be trusted to maintain the proper operation of security-relevant mechanisms includes the hypervisor, as well as the VMX helper. The size of both cannot be estimated, as no access to the source code is available. Considering the vmkernel file that implements the hypervisor, a significant part of the file is not considered to be active, as it implements the binary translation code, which is considered to be

[15] See http://www.vmware.com/pdf/vsphere4/r40/vsp_40_esx_server_config.pdf, chapter 11.

inactive in the configuration scenarios outlined here. However, considering the available information, the size of the software stack is considered to be low-to-medium, because there is no full-fledged general purpose operating system kernel part of the trusted software stack (which lowers the size significantly), and VMWare ESX adds the full virtualization support with VMX (which increases the size).

The set of interfaces exported by the software stack that is considered to be trusted covers the hypercalls exported by the hypervisor. In addition, the I/O interfaces provided with the full virtualization by the VMX helper is another set of interfaces. As already outlined in section 4.3.2, additional interfaces might be available to external entities. Therefore, a reliable assessment of the number of available interfaces is not possible and is, therefore, not performed here. However, considering the available information, the number of the interfaces is considered to be medium-to-high, at a similar level as Xen with QEMU, but without the xenstore daemon.

With regard to the development environment, VMWare's strategies about how to handle security-relevant flaws and monitor different sources for flaw reports is not fully public knowledge. Therefore, the development environment and resulting code quality cannot be reliably assessed. However, considering the Common Criteria evaluation of VMWare ESX server 3.0.2 at EAL4 augmented with ALC_FLR.1 and assuming that the evaluated development procedures apply to the currently-discussed version of VMWare ESX server, the development environment can be characterized as follows:

- Development of code is subject to developer peer review.
- Procedures exist to address security-relevant flaws and publish fixes.

Considering the Common Criteria evaluation, a medium assurance of the code quality can be assumed. Please note that this assessment states the lower boundary of the assurance given by VMWare. Procedures may exist that require a much more stringent code review and security flaw remediation than assessed by the Common Criteria evaluation.

5 Security Comparison Based on Scenarios

In the previous chapter, the analyzed VMM implementations are presented with their architectures and interfaces. Initial estimates of the attack surface are given, supported by the discussion of how and how quickly identified security-relevant flaws will be fixed.

To add another aspect to the comparison of the VMMs with respect to security behavior and mechanisms, this chapter defines a number of attack and usage scenarios, which are used as a basis for comparison. The comparison of security characteristics of the different VMM implementations is based on the attack vectors and usage scenarios introduced in the sections below. Each attack vector and usage scenario is analyzed to identify the mechanisms provided by each VMM to either counter and mitigate the threat, or to support the usage scenario.

The scenarios outlined below are standard issues a VMM implementation must be able to handle in day-to-day operations. This list could be extended by other scenarios, but the current assessment limits its focus to these scenarios.

The goal of the security comparison is to enable readers to understand how different VMM systems support day-to-day operations, so that they can identify whether a particular VMM is suitable for their intended workload and IT environment.

The following table summarizes the assurance the examined VMM implementations provide for covering the security aspects in case of a security-relevant flaw. The given assertions are relative to each other and do not provide any hint of absolute assurance for the respective VMM (as such, a value of “low” might still mean that even if a security-relevant flaw is found, it might be very hard to actually exploit it). The assessment assumes the most secure configuration possible to mitigate the outlined threats – the sections below outline these configurations.

Scenarios	KVM	Xen	VMWare ESX Server*
Assurance of protection against VM accessing unassigned resources mediated by para-virtualized drivers	Medium[16]	Low	Low
Assurance of protection against VM accessing unassigned resources mediated by full virtualization support software	Medium[17]	Stubdom: Medium Default: Low	N/A
Assurance of protection against subversion of trusted VMM software – subversion of Hypervisor	High[18]	High	Medium
Assurance of protection against Subversion of trusted VMM software – subversion of other virtual machines	Medium[19]	Stubdom: High Default: Medium	N/A
Assurance of protection against Subversion of trusted VMM software – subversion of boot process	High[20]	Stubdom: High Default: Medium	N/A
Assurance of protection against one VM causing a DoS of other VMs	High	Medium	Medium
Support for sandboxing usage	High	Medium	Low
VMs belong to different security domains	Low[21]	Medium	Low

Table 4: Assessment of coverage of security aspects based on scenarios in relation to each VMM

*The assessment of the VMWare ESX Server is based on knowledge obtained from public information. If VMWare ESX Server also includes additional mechanisms relevant to the scenarios described, the assessment might be incomplete.

The sections that follow elaborate on the summaries given in the table.

5.1 Guest VM access to unassigned resources

5.1.1 Attack scenario

The main goal of a VMM implementation is to restrict a guest operating system's access to assigned resources. Any resources not assigned to the virtual machine hosting the guest operating system must be forbidden. This statement applies to all kind of resources: physical resources, virtualized resources (i.e., physical resources where only a part or just some types of access are allowed by a virtual machine), or emulated resources (i.e., logical resources that are not backed by physical resources).

From the point of view of the VMM implementation, any code executing within a virtual machine is considered to be untrusted. Any information, access requests and other interactions with the code in the virtual machine must first be sanitized before the VMM can use it to perform operations with the potential to access resources that might not have been assigned to the virtual machine.

That said, threats arising from virtual machines that must be countered by the VMM must be considered to originate in the virtual machine's user state and supervisor state. Therefore, the VMM cannot even trust the kernel of the operating system executing within the virtual machine. Effectively, the VMM must still enforce access restrictions to resources if the entire operating system in a virtual machine has been completely taken over and the attacker has full access to the user space and kernel space of the virtual machine.

For this attack scenario, therefore, the following is considered:

[16] Assessment is “Medium to High” if sVirt is considered due to the fact that the SELinux separation enforcement also covers para-virtualized devices provided by the QEMU logic to the guest system. However, in newer implementations of KVM, para-virtualized devices may be provided by the Linux host system, limiting the effect of SELinux in this area.
[17] Assessment is “High” if sVirt is considered.
[18] Assessment is “High” if sVirt is considered.
[19] Assessment is “High” if sVirt is considered.
[20] Assessment is “High” if sVirt is considered.
[21] Assessment is “High” if sVirt is considered.

- An attacker has full access to user and kernel space of one regular general purpose virtual machine.
- An attacker has full access to all interfaces offered by the VMM to a virtual machine.
- An attacker tries to access resources that are unassigned to its virtual machine. These resources include physical, virtualized, and emulated resources.

5.1.2 General discussion of scenario

For assessing the outlined scenario, all components of the VMM mediating access to resources have to be considered. The following components are identified:

- Para-virtualized device driver support requires that access to resources is usually mediated either by drivers operating as part of the hypervisor (VMWare ESX) or as part of the Linux kernel in the privileged virtual machine (Xen). Only KVM implements these backends external to the hypervisor in an unprivileged environment, the QEMU process. These drivers mediate access requests from virtual machines to the respective resources. The implementations for VMWare and Xen have one aspect in common: there is only one instantiation per driver that mediates access for all virtual machines. For KVM, there is still one driver instance per virtual machine.
- Full virtualization support is mediated by a separate software component executing outside the hypervisor. This support is provided with QEMU (KVM, Xen) or VMX (VMWare ESX). For the full virtualization support software, the VMM implementations have one aspect in common: one instance of the software component is dedicated to handling the requests of one virtual machine. This implies that there are a number of instances of this software component equal to the number of virtual machines executing on the system.

All software components mediating access to resources have the goal to restrict access requests to the resources configured for the calling virtual machine. However, these software components have design-based drawbacks that work against that goal:

- The software components are large and very complex – this applies especially to full virtualization support. This implies that flaws, including security-relevant flaws, are likely to be hidden in the code base.
- The software components execute in a very privileged environment (the Linux kernel in Dom0 for Xen); some execute in the most privileged environment (the hypervisor for VMWare ESX) of the VMM – as such, any security-relevant flaw may provide an attack vector allowing the abuse of this high privilege.

The following assessment analyzes how the different VMM implementations handle these issues. This analysis includes a review of whether supporting separation mechanisms inherent in the design or implementation of each VMM mitigate the risks outlined above.

5.1.3 KVM

KVM currently provides only one para-virtualized devices which is even implemented in QEMU[22]. As such, the QEMU mediates access to a limited number of resources and is, therefore, considered to provide a smaller likelihood of attack vectors to circumvent resource separation than Xen and VMWare due to the limited privileges granted to QEMU by the Linux host system. A potential security issue of QEMU cannot be used to stage unrestricted access attempts to devices known to the Linux host system. However, such potential security issues can still be used to stage access attempts to resources assigned to other virtual machines, because all virtual machines as well as the virtual machine management daemon of libvirt execute with the same user ID.

In addition to the assessment of the para-virtualized device drivers, the full virtualization support implemented with QEMU is important, as QEMU also controls resources that must not be shared with other virtual machines, such as disk devices. Concerning the implementation details relevant for this assessment, please see section 5.2.3, which covers the following topics:

- The virtual machine processes also hosting QEMU execute with normal, unprivileged user ID. The use of such an unprivileged user ID is very effective in ensuring that the virtual machine processes cannot take control of the Linux host system. This user ID cannot be used to circumvent the access control settings of the UNIX permission bits or ACLs, trace other processes with a different user ID or access and modify their memory.
- All virtual machine processes with QEMU support execute with the same user ID[23]. The use of the same user ID for all virtual machines results in the conclusion that all resources of all virtual machines must be accessible by this user ID. Therefore, in case of security flaws in QEMU, the fact that all resources are accessible by all QEMU instances does not support the separation of resources assigned to different virtual machines.

5.1.4 Xen

The Xen hypervisor implements device drivers that are used solely to allow the hypervisor's execution on its hardware. The hypervisor does not implement the para-virtualization interfaces, keeping the hypervisor's device drivers exclusively for its operation.

The para-virtualized backend device drivers are implemented in the Dom0 Linux kernel. This Linux kernel implements a number of these backend drivers and provides, therefore, a sizable number of interfaces to the virtual machines. As the Linux kernel in Dom0 has direct access to most of the hardware resources, such as disk devices or networking, these backend drivers are the line of defense for separating the resources assigned to the different virtual machines. This implies that a security-relevant flaw in a backend driver might

[22] Please note that for new developments of KVM, para-virtualized device backends are considered to be implemented in the host Linux kernel. The consideration of SELinux enforcements inside the kernel when sVirt is active as part of future KVM releases: As specified below, SELinux with an appropriate rule set is effective in supporting the separation of the resources handled by QEMU. The question is whether SELinux is also effective in kernel space. First, this assessment assumes that the SELinux hooks are executed by the information flow between the para-virtualized support drivers of KVM and the physical device drivers implementing access to the real resources (please note that most of the SELinux hooks are in Linux kernel layers close to user space, well above the driver levels). SELinux would then also provide a hypervisor-inherent restriction on the information flow between resources and virtual machines supporting the separation of resources for the virtual machines. If a security-relevant flaw were identified in the physical or para-virtualized drivers in the kernel, however, it is likely that the restrictions of SELinux could be circumvented more easily, as SELinux operates in the same privileged security domain as the drivers. On the other hand, security flaws in user space, even in processes with root privileges, cannot circumvent SELinux, as these flaws do not provide access to the kernel where SELinux operates. In conclusion, one must consider that any SELinux restrictions enforced for para-virtualized device drivers are not as effective as the same restrictions enforced for user space processes.

[23] The sVirt support which is already available but not yet integrated with RHEL5.4 and the discussed KVM version provides additional separation. The same applies to the separation provided by cgroups concerning disk devices. With sVirt and the associated SELinux policy assign unique SELinux categories to each virtual machine. This also implies that access to resources assigned to a QEMU instance is inherently separated by the VMM. If QEMU contains a security-relevant flaw, sVirt provides an additional system-inherent separation mechanism that would prevent a possible abuse of the flaw to access resources of other virtual machines. Therefore, the SELinux restrictions considered to be effective for reducing the impact of the use of the same user ID for all virtual machines to support the separation of resources. Similarly, access restrictions to disk devices provided with cgroups add another layer of separation between virtual machines.

have a higher likelihood of allowing a break in the separation of resources between virtual machines. The sHype or Flask support of the Xen hypervisor are not considered to provide a remedy, as this support controls access to the hypervisor interfaces and inter-domain communication channels. However, neither sHype nor Flask supports the separation enforced by the Linux kernel backend drivers. An additional consideration is important to the backend drivers: most of them are triggerable via the Xenbus inter-domain channel. This assessment did not further elaborate on the Xenbus protocol, but notes that there is one bus that connects all domains with Dom0. Such a design might have the potential to further weaken the separation enforcement of the backend drivers if attacks are developed for spoofing, replaying, sniffing, or others of Xenbus messages.

With respect to the full virtualization support component of QEMU, section 5.2.4 explains implementation details relevant for this assessment. The default configuration for the QEMU instances mediating access to resources is the execution of equally-privileged processes executing on the Linux kernel in Dom0. As such, the same concerns previously described for KVM also apply to Xen:

- The QEMU processes execute with root privilege in Dom0. This implies that they have authority in the Dom0 operating system and can utilize all services a Linux kernel provides to processes with root privilege. If the QEMU implementation contains a security-relevant flaw, the likelihood is high that it can be used to trigger operations executing with root privileges in Dom0. This also implies that such flaws have a higher likelihood to interfere with the operation of other QEMU processes and the resources they control. This ultimately provides an attack vector that might be used to circumvent the separation of resources between virtual machines. To counter this threat, Xen provides a special configuration for QEMU: hosting each QEMU instance within a separate virtual machine, a Stubdom. As such, a breach of one QEMU instance is restricted to its virtual machine. An attacker might have the chance to abuse such an attack vector to access the para-virtualized devices exported by Dom0 to the Stubdom. Such indirect attack of Dom0 is, however, considered to be much more complex than a direct attack of para-virtualized backend drivers.
- Similar to KVM, the QEMU processes execute with the same user ID in Dom0 in the default setup. Thus, all resources the different QEMU processes must have access to also share the same level of access. However, this issue is negligible compared to the issue of root authority of the QEMU processes discussed above.

5.1.5 VMWare ESX Server

The vmkernel hypervisor implements several device drivers that are accessible from virtual machines via para-virtualized driver interfaces. As the hypervisor and, therefore, these device drivers have direct control over the physical hardware, these drivers together with their para-virtualized driver interfaces must implement the capability to separate the resources assigned to the different virtual machines. Considering the VMI para-virtualization interface description developed by VMWare, a number of para-virtualized drivers together with their physical device drivers are considered to be implemented in the hypervisor. Any security-relevant flaw in any of the para-virtualized device drivers might provide an attack vector to circumvent the separation capability enforced by this driver. Additional architectural support for ensuring the separation is not identified.

The assessment of the full virtualization support component provided with VMX cannot be completed. The following issues for which public documentation was not found would need to be assessed in order to complete the assessment:

- Which operations can be performed by a VMX on the hypervisor?
- Are there additional access control mechanisms enforced by the hypervisor that restrict access requests from one VMX to resources not assigned to that VMX instance?

5.2 Guest VM subversion of trusted VMM software

5.2.1 Attack scenario

In order to ensure the proper separation of resources, the entire functionality of the hypervisor must be trusted. Any software component part of the hypervisor has full hardware privileges and can access any resource of the system, including physical devices, data stored on devices, etc.

In addition to the hypervisor, chapter 4 lists the software that is considered to be trusted and is accessible from virtual machine guest software.

If any of the mentioned trusted software components can be altered by the guest software running within a virtual machine, the software is considered to be subverted, and trust in this software component, as well as trust in the separation capability of the VMM, is undermined.

In addition to the modification of trusted software, an attacker might want to add a completely new software component. If the guest software has the ability to place another layer of software between the hypervisor and the hardware, the results of the operation of the hypervisor cannot be fully trusted, as the additional software layer can emulate a different environment and behavior of the underlying hardware. Effectively, this additional software layer adds another untrusted VMM layer.

These concerns cover rootkits or virus-like exploits of the VMM software, as well as the subversion of the boot process of the VMM.

5.2.2 General discussion of scenario

In order to understand the abilities of the analyzed VMMs to protect against this attack, the goals of viruses and rootkits must be analyzed. Rootkits and viruses try to evade the security policies of the exploited system to grant the owner of the malware access to system resources, user resources, system data, or user data they would not otherwise have access to. For this discussion, it is irrelevant whether rootkits and viruses only exploit the system once and then somehow remove themselves, or whether they are permanent critters that exploit the system over a duration of time.

Viruses and rootkits need privileges to access the security domain of the software they want to subvert, they need privileges to access a security domain that controls other security domains, or they need to have access to the boot procedure. The security domains relevant for VMMs are the resources maintained for and by the hypervisor and those of the separate virtual machines. Therefore, the following general attacks are possible:

- Access of security domains: Malware needs to access the resources of the hypervisor to subvert this hypervisor. Also, malware needs to access the software providing full virtualization support executing in a privileged security domain in order to subvert it.
- Access of security domains controlling other domains: Malware might modify the software providing full virtualization support executing in a privileged environment by first exploiting flaws in the hypervisor, because the hypervisor also controls the resources of the security domain of the privileged virtual machine executing this emulating layer.
- Interfering with boot process: If malware can interfere with the boot process by manipulating the boot loader, the boot loader's configuration, or the first stages of the booted VMM, it can also subvert any operation of the VMM. The reason is that the hardware starts the boot sequence by granting the booted software full privilege. This implies that the hardware initially allows every operation performed by the software. During a normal boot process, the booted VMM must program the hardware to enforce restrictions on subsequent software. When malware is able to access this boot sequence, it has the same hardware privileges as it would have when subverting the hypervisor at VMM runtime.

Based on the description above, it is clear that the hypervisor is the most critical software component of a VMM, as it controls and has access to all resources of the hardware, including its own and those of the virtual machines managed by the hypervisor. This implies that any interfaces exported to virtual machines executing untrusted code and any functionality accessible by those virtual machines are potential attack vectors. The smaller the functionality (and its complexity) and the smaller the number of interfaces, the less attack vectors

exist. The less attack vectors exist, the less likely an implementation can be attacked. Translated to the functionality of common hypervisors, the following statements can be considered: Device drivers and other resource access-mediating logic implemented in the hypervisor that are directly accessible by exporting interfaces usable by para-virtualized device drivers to virtual machines provide direct attack vectors. Attackers can perform any operation on these directly-accessible interfaces (please note also that virtual memory management can be considered as a form of device driver here and, therefore, para-virtual interfaces to the virtual memory management are covered by this statement about direct interfaces). Device drivers and other resource access-mediating logic implemented in the hypervisor that are accessed indirectly via the full virtualization support usually implemented in a privileged virtual machine or a similar concept provide attack vectors where the attacker can only exercise operations limited by the full virtualization software components – one notable exception is KVM which implements the full virtualization support as an unprivileged software component with respect to the host system. To conclude, the following must be considered: the smaller the number of device drivers (methods to access resources mediated by the hypervisor) the hypervisor implements, the less attack vectors are provided. And the less attack vectors are available, the smaller the probability of exploiting the hypervisor.

In addition to the problems to be handled by the hypervisor, the full virtualization support software needs to be examined. As analyzed in chapter 4, all VMM implementations relevant for our discussion implement the full virtualization support software (QEMU for KVM and Xen, VMX for VMWare ESX Server) as part of the privileged software stack. The privilege arises from two aspects:

- The full virtualization support software has privileged interfaces to the hypervisor and, therefore, has the potential to interfere with the security functionality of the hypervisor – this issue was previously described in the discussion about subverting the hypervisor via an indirect attack vector.
- Separately from using the full virtualization support software to access the hypervisor, this support software may also be used to access other instances of the full virtualization support software maintained for other virtual machines. When the different full virtualization support software instances execute with the same privileges for all virtual machines, security flaws in these software components provide attack vectors to subvert the operation of the support software of other virtual machines. This implies that viruses or rootkits might not penetrate the hypervisor, but might gain equal access to other virtual machines' resources by attacking the full virtualization support software, as separation between them is not fully enforced.

5.2.3 KVM

Considering the concerns regarding the hypervisor, the architecture of KVM allows the following conclusions:

- Development of para-virtualization device driver support is ongoing for KVM. Currently only one para-virtualized device backend is directly accessible which is even implemented in QEMU and therefore covered by process isolation mechanisms of the host Linux kernel. As such, the number of para-virtualized devices and the number of interfaces provided by the hypervisor of the Linux kernel is limited when compared to Xen or VMWare ESX. Also, the para-virtualized backend device is not implemented in the privileged hypervisor as it is common practice for other hypervisor implementations, adding additional assurance for maintaining separation of the virtual machines.
- Indirect interfaces are provided by the Linux kernel, which acts as a hypervisor via the QEMU instances. The processes providing the virtual machines that also host the QEMU full virtualization support software all execute unprivileged user IDs on the Linux kernel of the host system. This implies that QEMU only has very limited means to interact with the Linux kernel of the host system, which restricts the capabilities of QEMU to cause unwanted behavior on the host system. Therefore, the hypervisor of the Linux kernel is considered to be hardly penetrable by the QEMU logic.

As the Linux processes hosting the virtual machines, including the QEMU component, all execute with the same user ID[24], the QEMU processes can interfere with each other by the following means:

[24] Future versions of KVM will provide secure configuration options to contain the issues concerning the separation of virtual machines from each other due to the use of the same user ID for each of it: sVirt and cgroups. sVirt employs the SELinux support of the Linux kernel and configures an SELinux label used for all virtual machines. When setting up virtual machines covered by the SELinux label, the above identified attack vectors resting on the issue of the same user IDs of the virtual machines are mitigated. In addition, cgroups further limits access to disk devices for

- The Linux kernel provides facilities to processes executing with the same user ID to access other process' memory with the potential even to alter memory using the ptrace system call.
- The resources assigned to a virtual machine include disk space provided via physical disk or partition device files or via disk files stored on the host system. These disk devices are read and write-able by every virtual machine process, because the virtual machines execute with the same user ID and group ID in the Linux host system. This concern applies to all resources that are accessible as file system objects, such as dedicated devices accessible through device files.

If QEMU contains a security-relevant flaw, it may serve as an attack vector for viruses and rootkits to utilize the discussed interfaces and interfere with the operation and the resources of other virtual machines, but not with the host system.

Finally, the use of an unprivileged user ID does not allow any interference with the boot process of the host system. Even if there would be a QEMU security-relevant flaw, the unprivileged user ID of the virtual machine process with the QEMU instance does not allow any administration on the host boot process.

5.2.4 Xen

The general security concerns applicable to the Xen hypervisor can be characterized as follows:

- The number of hypercalls provided to virtual machines is sizable. A subset of these hypercalls is provided to allow access to the para-virtualized device driver support provided with Xen. However, with the exception of virtual memory management, the hypervisor does not mediate access to devices, but only allows the establishment of inter-virtual machine communication to the para-virtualized backend drivers hosted by the Linux kernel in Dom0.
- The other subset of these hypercalls is used to manage aspects of the virtual machines from Dom0. As the QEMU processes instantiated in Dom0 for each virtual machine have root privilege, they may use every interface accessible from the Linux kernel driving the software in Dom0. However, the QEMU instances have no direct access to the hypercalls. Nevertheless, the Linux kernel provides partially-privileged interfaces in /proc/xen (such as the privcmd file) that are used to manage aspects of the hypervisor, including SELinux policy loading and others. This implies that QEMU has indirect access to the hypervisor via the interfaces exported by the Linux kernel in Dom0.

The Xen QEMU instances execute as root on the Linux kernel in Dom0. As such, the same concerns exist with respect to the separation of the different QEMU instances from each other to ensure that they cannot interfere with each other. The same aspects apply to Xen, as well.

Also due to the fact that the QEMU instances execute with root privileges, potential security problems in QEMU may provide an attack vector to modify the boot process of the Xen hypervisor and the Dom0 environment. Please note that the configuration file of the boot loader as well as every aspect of the boot procedure including the booting of the Xen hypervisor can be modified using the root capability in Dom0 which grants write access to the boot hard disk partition.

Xen allows the use of access control modules, with two such modules currently provided: Flask (equivalent to the SELinux logic found in the Linux kernel) and sHype. However, rule sets need to be defined for these mechanisms to partially remedy the issues outlined above. As no default rule sets are defined, this assessment considers the availability of the framework. Due to the missing rule set, however, the “no enforcement” logic can be applied, which means that the two existing access control modules cannot be considered for this assessment.

An additional remedy that is effective for separating the QEMU instances is the use of Stubdoms. Effectively, each QEMU instance is placed together with a mini kernel into a separate DomU and then translates the I/O requests from an HVM virtual machine into para-virtualized I/O requests handled by the Linux kernel in Dom0. Therefore, the entire issue around QEMU executing as root on the Linux kernel in Dom0 is solved if such a Stubdom is instantiated for every general-purpose DomU. Such a configuration also ensures that the QEMU instances cannot interfere with each other.

virtual machines to only their assigned disks.

5.2.5 VMWare ESX Server

The general security concerns applicable to the vmkernel hypervisor of the VMWare ESX server can be characterized as follows:

- The hypervisor implements a number of hypercalls if it follows the VMI specification defined by VMWare. As such, it offers a number of interfaces to device drivers implemented in the hypervisor, mainly for memory management, storage, and networking. These hypercalls provide a direct interface to the hypervisor device drivers.
- An additional set of interfaces, the “user world” interfaces that implement a subset of POSIX, are offered to support software running outside the hypervisor and do not belong to the virtual machines. One of these support software components is VMX, the full virtualization support software instantiated for each virtual machine. Therefore, VMX allows virtual machines to indirectly use these user world interfaces of the hypervisor, as well. As this set of user world interfaces is also used by administrative services to configure and manage virtual machines, it might be possible that the VMX instances technically also have access to these administrative hypervisor interfaces. It is unclear as yet whether this user world interface is covered by a privilege mechanism that restricts the number of available interfaces to VMX, preventing VMX from accessing administrative interfaces.

Concerning the interference of the VMX instances with each other, no assessment can be made, as the following information is not available:

- Do the VMX instances execute with the same level of privilege?
- What kind of user world interfaces are offered by the hypervisor that could be used by one VMX instance to access resources of another instance?

The assessment of the subversion of the boot procedure is also impeded by the lack of information around the boot process of ESX's vmkernel. The following questions would need to be clarified to conclude the assessment:

- Is the boot partition technically writable by any VMX instance?
- Is there a user world interface that allows triggering/changing the boot loader configuration and the boot loader code?
- Is the disk partition holding the boot loader code technically writable by a VMX instance?

5.3 Guest VM causes Denial-of-Service for other VMs

5.3.1 Attack scenario

In addition to the proper separation of virtual machine resources, the VMM implementation also has to ensure that resources shared between the virtual machines are shared such that one virtual machine cannot fully utilize the resource.

For example, the physical CPUs are assigned to a virtual machine for a specified period of time and then reallocated to other virtual machines by the VMM. The VMM now has to ensure that a virtual machine does not have the potential to utilize more CPUs than configured to avoid resource starvation, also called a denial-of-service, to other virtual machines. Such a denial-of-service attack is considered to take place if the execution of a virtual machine is degraded to such an extent that normal operation of software within that virtual machine is nearly impossible.

Please note that only denial-of-service attempts impacting other virtual machines are considered in this assessment. If a virtual machine can cause a denial-of-service of a part of the VMM without other virtual machines being affected by that denial-of-service, it is considered a non-issue for this assessment. Also, denial-of-service of a resource that is exclusively available to the virtual machine mounting the denial-of-service is also considered to be irrelevant to this discussion.

5.3.2 General discussion of scenario

Denial-of-service attacks can only be performed targeting resources that other virtual machines use (either as shared or exclusive resources). As a number of resources are shared between virtual machines and the VMM (such as CPU time, PCI host bridges, and other physical bus systems when accessing physical devices), the resource utilization of such shared resources by the VMM when it is acting on behalf of a virtual machine request is a very important aspect. If a virtual machine invokes a service from, say, the hypervisor by making a hypercall, the hypervisor CPU execution time for performing the call and returning to the virtual machine must effectively be added to the utilization of the CPU by the calling virtual machine. If such an approach would not be taken, a virtual machine has the potential to fully or partially exhaust the shared resource (the CPU execution time in our case) by repetitively calling the discussed hypervisor service function.

In addition to the CPU, the second major shared resource is memory. As long as portions of memory are statically assigned to virtual machines and these exclusive assignments are not altered for the duration of the execution time of a virtual machine, denial-of-service attacks against memory are less of a problem. However, if active management of memory by the VMM is performed, threats arising from denial-of-service attacks are much higher. General denial-of-service attacks include the following types:

- Attacks against over-commitment of memory imply that the VMM is able to assign more memory to virtual machines than is physically available. Different techniques for over-commitment are available. Denial-of-service attacks can be mounted against two aspects:
  - Using a much slower storage space as an extension of physical memory (swap space on hard disk, for example), forcing the VMM to constantly page memory to and from that extension space, has a high chance to slow the VMM operation down, impacting other virtual machines.
  - Over-committing memory: the VMM must be able to handle situations when it is out of memory. It is especially important that the components of VMM are not affected by the out-of-memory handling logic.

In addition to CPU and memory, other resources also are subject to denial-of-service-considerations. The assessment of the individual VMM implementations will address these aspects if remedy support is implemented.

Please note that the most effective remedy against a denial-of-service attack is to minimize shared resources as much as possible. If shared resources are needed, the level of sharing should be minimal in order to reduce the effect of denial-of-service.

5.3.3 KVM

The hypervisor for KVM is the Linux kernel. As the virtual machines are handled like normal applications by the Linux kernel, several supporting mechanisms against denial-of-service attacks can be utilized:

- The direct CPU utilization depends on how the Linux kernel schedules the virtual machine process compared to other processes. Scheduling can be influenced by setting the nice level for a virtual machine process. This changes how often the Linux kernel scheduler assigns CPU time to the virtual machine. Using the nice level, the effects of a denial-of-service attack by over-utilization of the CPU can be reduced. The Linux kernel even allows the binding of virtual machine processes to specific CPUs, allowing configurations where important virtual machines that might be more trusted have access to different physical CPUs than virtual machines that are not fully trusted. This implies that denial-of-service attacks by untrusted virtual machines against trusted virtual machines cannot be effectively executed with respect to CPU time.[25]
- The Linux host system for KVM usually configures a Swap space to allow the Linux kernel to assign more memory to processes, including virtual machine processes, than is physically available. Processes can cause denial-of-service situations when they can trigger the kernel to start massively swapping memory to/from the Swap space. If such attacks are considered, the Swap space may be either reduced in size or eliminated completely, which essentially reduces the Linux kernel's ability to over-commit memory. Please note that the virtual machines must be configured with the maximum

[25] Using cgroups in RHEL6, virtual machines can also be restricted to CPU limits which supports the prevention of denial of service.

memory available. If the maximum memory settings of virtual machines are less than the physical memory available on the host system, swapping cannot be directly triggered by virtual machines. However, issues in QEMU may trigger additional memory consumption of the QEMU part of the virtual machines, circumventing the maximum memory setting for a virtual machine.

- In case over-commitment of memory is used, what happens when the host system runs out of memory? The Linux kernel implements a mechanism called OOM-Killer which provides an algorithm for the kernel to identify a likely process to kill if the host system is out of memory. As OOM-Killer is tuned to handle general purpose applications, its logic might not be fully suitable if Linux is solely used as a hypervisor. Still, having an OOM-Killer supports the hypervisor in avoiding denial-of-service attacks against memory.
- The Linux kernel allows the configuration of resource quotas or the use of traffic shapers to limit the use of resources including network bandwidths. These mechanism support the host to limit the impact of denial-of-service attacks against covered resources.

Please note that all the relevant settings offered by the VMM must be appropriately tuned to be effective, depending on the actual attack scenario considered for denial-of-service attacks.

5.3.4 Xen

The allocation of CPUs to virtual machines is controlled by the hypervisor based on how many CPUs are configuration for a virtual machine. This implies that the hypervisor must handle denial-of-service attacks against the CPU. Xen does not provide any mechanisms to influence the scheduling of virtual machines on physical CPUs that would support the prevention of denial-of-service attacks.

To limit the effects of denial-of-service attacks against CPU time, the number of virtual CPUs allocated to a virtual machine can be reduced. The hypervisor considers the number of configured virtual CPUs when scheduling virtual machines. The hypervisor even allows the binding of virtual machines to specific CPUs, allowing configurations where important virtual machines that might be more trusted have access to different physical CPUs than virtual machines that are not fully trusted. This implies that denial-of-service attacks of untrusted virtual machines against trusted virtual machines cannot be effectively executed with respect to CPU time.

Memory over-commitment can be configured for virtual machines. Xen implements the over-commitment by a ballooning approach without swapping. As such, the VMM does not maintain a Swap space, but the operating system within virtual machines can dynamically increase or decrease their virtual machine's memory allocation. The hypervisor ensures that every virtual machine is provided with a configurable minimum memory amount. This implies that if an out of memory situation occurs, every virtual machine has still the minimum memory available. This effectively stops the effects of denial-of-service attacks if the administrator sets the minimum memory for virtual machines at a level that ensures the proper operation of the guest software when restricted to this minimum amount of memory.

5.3.5 VMWare ESX Server

The shared resource of CPU time can be protected against denial-of-service attacks by specifying a minimum and maximum percentage value of CPU time available to a certain virtual machine. The ESX server considers these values when scheduling virtual machines on the physical CPUs, causing denial-of-service attacks from one virtual machine against another virtual machine to be ineffective with respect to CPU time.

Similarly to the other VMM implementations, the ESX server allows memory over-commitment. This implies that the memory allowed to be used by all virtual memory is larger than the physically available memory. However, when all virtual machines are started, the initial memory consumption is lower than the physically available memory. The ESX server can modify the virtual machine's memory allocation during runtime. An appropriate mechanism to avoid out-of-memory situations is assumed to be in place. If the administrator wants to be on the safe side, no over-commitment of memory should be configured.

The network bandwidth controlled by the ESX server and granted to virtual machines can also be configured and limited. Therefore, denial-of-service attacks using the network against virtual machines is of limited effect, at least when the attack source is one of the virtual machines operated by the ESX server.

The configuration of limits for shared resources and the assignment of such limits to virtual machines is considered to be average compared to other VMM implementations.

5.4 Usage of VMM for sandboxing

5.4.1 Usage scenario

The previous scenarios all examined possible abuse of either services or the architecture of the VMM implementation. Starting with this scenario, specific usages are analyzed, along with how the VMM implementations cover the respective usage requirements.

The first usage scenario covers the application of VMMs for sandboxing. Sandboxing is considered when the host or one virtual machine is used for regular day-to-day work. It is irrelevant whether a regular server logic is implemented in the one virtual machine or whether the VMM implementation is used on an end-user system. In general, only one operating system is hosted on the entire physical machine.

However, there are times when the user or administrator of the physical machine wants to start a virtual machine as a test environment, for example during development of new software or when testing new configurations or deployment scenarios of software. Also, virtual machines may be used to execute software that cannot run natively on the standard operating system; for example, the standard operating system is Linux, but the user/administrator wants to run an application that can only be executed on Windows.

5.4.2 General discussion of scenario

When using a VMM for sandboxing, the user generally wants to use the hardware with one operating system only and use the VMM environment only sporadically.

As such, this assessment validates whether a VMM implementation can provide one environment that behaves as if were operating on native hardware. This implies that access to all physical resources is possible and that all properties of the hardware are usable from this main environment. In addition, the VMM implementation must allow the starting of virtual machines without interrupting the regular operation of the main environment.

5.4.3 KVM

The KVM virtual machine monitor rests on the operation of a Linux host operating system. As virtual machines execute concurrently with normal processes and the Linux kernel support of normal processes is unaltered, the host system provides a normal Linux environment. This environment allows the execution of normal Linux applications as if KVM support were not available. Virtual machines are instantiated like normal processes and, therefore, starting new virtual machines does not interfere with the general operation of other processes.

The Linux kernel of the host system has full hardware access – there is no additional software layer between the underlying platform and the kernel. As such, the Linux kernel can put all its drivers to use; as a result, the Linux host system behaves like any other Linux system, whether or not virtual machines are running. With this design approach, all desired desktop and laptop functionality can be used, including power management, suspend/resume, hibernation, direct rendering support for graphics hardware (i.e., 3D-support for graphics software), and others.

5.4.4 Xen

The general-purpose operating environment that allows most hardware access is Dom0. If Dom0 is used as the general work environment when using the VMM system for sandboxing, other virtual machines can be spawned without interrupting the operations in Dom0.

Please note that the Linux kernel in Dom0 has direct access to a subset of the underlying hardware only. The hypervisor controls the rest of the hardware. This means that direct access to this hardware is not allowed for the Dom0 Linux kernel and it cannot employ its drivers and support logic when using this hardware.

As the hypervisor is developed solely for the purpose of acting as a hypervisor, support for special hardware mechanisms is limited. This means that Xen does not provide support for mechanisms that are generally assumed to be available for desktop or laptop systems. Such missing support includes power management, suspend/resume, hibernation and other mechanisms that build around the CPU.

This means that Xen is somewhat less suited as a sandbox system if full support of the underlying hardware is important.

5.4.5 VMWare ESX Server

With respect to the support of all aspects of the underlying hardware, the architecture of VMWare ESX server is even more restricted than for Xen. As the ESX server does not maintain a privileged virtual machine environment with access to most of the hardware, all hardware access ultimately is mediated through the vmkernel hypervisor. Considering the number of devices supported by the ESX server hypervisor, only a small subset of hardware available for x86 architectures is supported.

As such, an operating system must execute in a normal virtual machine and is, therefore, limited to the hardware access allowed for such a virtual machine.

5.5 Guest VMs belong to different enterprise security domains

5.5.1 Usage scenario

Another usage scenario applied to this analysis is the handling and protection of groups of virtual machines based on their assignments to security domains. A security domain in enterprise networks is a group of services, information and/or resources that are considered to require an equivalent level of trust when accessed.

For example, the database for a web application is considered to be in a separate security domain than the web application itself. The web application is accessible from the Internet, while the database is well-protected from the Internet, but accessible by the web application. Different levels of trust are applied to the different services and data. Another example would be hosts that provide services of a DMZ vs. other hosts that provide services for the accounting department of an enterprise.

This user scenario analyzes how the different VMM implementations can handle groups of virtual machines that belong to different security domains. This implies that resources belonging to the different groups of virtual machines can be categorized and that additional separation of the resources of the virtual machines is performed based on these categorizations.

5.5.2 General discussion of scenario

When considering the use of VMM for hosting virtual machines belonging to different enterprise security domains, it is important that the VMM ensure several aspects:

- The classification of enterprise security domains implies that general categories of data and functions are to be considered when administering the enterprise network. The different categories of data and functions must be differently protected, different levels of network protection might be considered to be necessary, and different backup strategies might be important. As such, a VMM would support different security domains if it has the capability to enforce an access control policy based on the labeling of data and resources. These labels shall reflect the category of the respective enterprise security domain. In essence, such mandatory access control restrictions support the prevention of covert storage channels between the different enterprise security domains.

- In addition to a mandatory access control mechanism for controlling access to resources, the problem of controlling covered channels arise. Separation of different enterprise security domains also implies that no information shall flow from one domain to the other. Resource sharing systems, including VMMs, always provide covered timing channels with the shared resources. Currently, there is no practical solution to avoid timing channels in such systems. Therefore, the issue of timing channels is disregarded for this analysis.

The hardware hosting the VMM and all resources accessible by the VMM must be protected based on the requirements defined for the software in a virtual machine belonging to the most critical security domain. For example, a security domain for public servers and internal databases exists, where the internal database servers must be hosted in a special computer lab. If a VMM now hosts a virtual machine that belongs to the public server category, as well as another virtual machine belonging to the protected database server category, the hardware hosting the VMM must be treated according to the stricter physical requirements defined for the database server category.

5.5.3 KVM

For KVM, no mandatory access control enforcement could be identified to support the definition of enterprise security domains and assign resources as well as virtual machines to these domains[26].

5.5.4 Xen

The Xen hypervisor allows the use of an access control module that allows the intercept of hypervisors, as well as inter-domain communication. Two different access control modules are provided: sHype and Flask. Both modules must be employed with a policy similar to SELinux that governs access control enforcement. As indicated earlier, no pre-defined rule set could have been identified, but an administrator may specify his own rule set.

However, mandatory separation enforcement with respect to virtual machines and their resources is not considered to be complete, as the resource access mediated in Dom0 (either via the Linux kernel para-virtualized drivers or via the QEMU processes) is not subject to the mandatory access control enforcement provided by sHype or Flask.

Considering the ACM hooks defined by include/xsm/acm/acm_hooks.h, no hint as to the coverage of the Xenbus inter-domain communication channel by ACM and, thus, sHype or Flask, is evident. As such, the Xenbus and all communication flowing through it provides another communication channel that is not subject to a mandatory access control policy.

5.5.5 VMWare ESX Server

For VMWare ESX Server, no mandatory access control enforcement could be identified.

6 General Considerations

The assessment of security aspects could be enhanced by considering additional factors beyond the technical aspects of VMM design and architecture discussed in this paper. For example, additional in-depth analysis might be performed covering implementation around generally complex areas, including:

- DMA handling
- Real mode handling for the boot process (please note that during the boot process, every operating system assumes the processor is in real mode)

Coverage of such details is out of scope of this assessment, as it would have greatly enlarged the discussion.

[26] The Linux kernel offers SELinux functionality, which can be used to enforce a mandatory access control rule set. A rule set is defined for KVM, and the management interface already applies this rule set by assigning an SELinux label to the virtual machine resources and the virtual machine process. The discussion of sVirt and how it supports the mandatory separation of virtual machines and their resources applies to this scenario as well.

In addition, the following non-technical aspects may be considered when determining whether a VMM is suitable for an organization's needs:

- Open source vs. closed source: The reader and user of VMMs should be aware of the two fundamentally different development approaches and determine which approach best supports the needs of the deployment environment.
- Available hardening procedures: Hardening procedures available for a specific VMM might support the secure operation of this VMM.
- Administration interfaces: Well-designed and helpful administration interfaces that do not hide the functionality of VMMs are very important. GUIs tend to obfuscate the complex nature of a product – thus, well-implemented GUIs that allow the administrator to understand the implications of his actions are important.