A Virtualization Architecture for In-depth Kernel Isolation[*]

Jiang Wang, Sameer Niphadkar, Angelos Stavrou, and Anup K. Ghosh
Center for Secure Information Systems,
George Mason University Fairfax, VA 22030

Abstract

Recent advances in virtualization technologies have sparked a renewed interest in the use of kernel and process virtualization as a security mechanism to enforce resource isolation and management. Unfortunately, virtualization solutions incur performance overhead. The magnitude of this overhead is directly proportional to the extend of virtualization they offer: full virtualization incurs an additional indirection layer to interface with the ever increasing hardware devices.

In this paper, we propose a hypervisor-assisted, microkernel architecture which aims to provide in-depth resource isolation without the performance penalty of full virtualization. To that end, we extend the hypervisor capabilities with a lightweight VMM which enforces "identity context" to all assigned devices for each of the hosted kernels. Furthermore, we separate the control from the data plane for all hardware devices using data memory mapping and modifications of the native device drivers to divert control flow via the hypervisor. Our approach is layered, accommodating a wide-range of devices from legacy to experimental devices able to provide native, in-silicon context separation.

1 Introduction

In recent years, there has been a renewed interest in virtualization technologies. This lead to a wealth of systems that offer different layers of virtualization ranging from lightweight process virtualization [2, 10, 17, 11] to full kernel virtualization [3]. However, the primary design tenet of all virtualization techniques is to enforce logical separation between shared resources by implementing kernel data-structures called "containers", "namespaces" or more general reference monitors. Lightweight virtualization systems including OpenVZ [2], VServer [17], ZAP [10], and Solaris zones [11] have been successful in implementing in-kernel process isolation into containers to manage and share available resources. Similarly, para-virtualization and hardware-based virtualization systems such as Xen [5] and VMware [3] depend on a privileged abstraction layer which acts as a mediator between the individually instantiated kernels and hardware devices. This layer creates the required indirection that enables the isolation of hardware resources from un-trusted kernels.

However, from security standpoint, para-virtualization offers better isolation, preventing kernel rootkits from escaping from the process containers. Unfortunately, although lightweight compared to full virtualization, current para-virtualization schemes incur more overhead than the in-kernel, container isolation mechanisms [17, 10, 2]. Most of this overhead can be attributed to the virtualization of I/O operations and the need for shared device context switching. Recent advances in hardware attempt to address this by enhancing the way I/O virtualization is implemented. However, recent hypervisor designs do not reflect these improvements. Moreover, the complexity of VMMs has increased remarkably owing to the management of resources and compatibility with various kernels and applications.

In summary, there are two major families of virtualized systems: one that is based on a single kernel model with shared resources and separate processes in the user space, and another that utilizes a hypervisor to isolate different kernels at the cost of I/O overhead and dependence on a complex VMM. We posit that both aforementioned virtualization models have strengths that we can harness. By combining strong isolation techniques of the para-virtualization architecture with the improved performance and low resource consumption of the native device drivers from process containers, we get the best of both worlds without any of the weaknesses. Our aim is affordable security and dependability through strong isolation.

Unlike container-based systems, we maintain state separation both in the kernel and user-space by deep hypervisor-assisted, kernel-level isolation which includes the device driver state. In addition, to reduce the performance overhead, we propose a different I/O model that differentiates between the control and data access requests for each device. To that end, we identify the necessary modifications for native device drivers to be hypervisor-

[*]This work was funded in part by DARPA under Army contract W31P4Q-07-C-0244 and also by AFOSR under MURI Grant # FA9550-07-1-0527.

friendly, supporting many state contexts. Lacking native device support, the hypervisor keeps a table of the contexts for the device and maps it to different guest OSes. All the device driver code always executes under the identity of the guest kernel that issued the request. Of course, any access for both control or data requires the permission of a controller module in the hypervisor. The same controller is employed when there is a conflict for device access. Unlike para-virtualization, we do not rely on the validity of the native driver code nor of a privileged domain acting as a manager. Since there is no master device driver or a back-end driver which takes full control of the device and shared by the guest OSes, there is complete isolation between different guest OSes preventing any data "bleeding" from hardware devices. For instance, even if an application or a device driver is compromised within a guest OS, then this compromise will not enable the attacker to gain access to the rest of the system. Notice that not all guest OSes require the same set of device drivers.

Another important characteristic of our system that enhances the dependability, and some might argue security through availability, is the reliable allocation and scheduling of available host resources. Being able to enforce resource consumption policies prevents resource starvation attacks. Indeed, we employ a resource module on the hypervisor exploiting the fact that it runs at a lower level controlling all hardware resources.

Moving forward, most of the functionality that is currently implemented inside the hypervisor, other than the resource allocation and scheduling, can be transitioned to the hardware. Of course, it is necessary for hardware manufacturers to be encouraged to support the I/O hardware enhancements seen on the modern CPUs. Self-virtualized or direct pass-through I/O and IOMMU [1] support on AMD SVM [4] and Intel VT-d [7] are two technologies that can help devices and device drivers become hypervisor-friendly leaving only the identity management to the hypervisors. We believe that our work provides a compelling security and performance argument towards that direction.

2 System Architecture

The primary goal of our design is to provide full isolation between the instances of the guest OSes. We do so by removing any master or globally shared code including device drivers and by preventing any resource starvation attacks. In addition, we would like to avoid the performance penalty of full virtualization for I/O. Our approach is inspired by recent hardware support for context separation using self-virtualized or direct pass-through I/O and IOMMU[1] support on AMD SVM[4] and Intel VT-d[7]. However, even if this hardware-assisted I/O architectures appear to be superior in terms of performance and context separation, to take advantage of their potential, we need the support of device manufacturers. In our system, we do not assume the existence of such support. Instead, we can also accommodate legacy devices with unmodified drivers. We do, however, analyze the benefits of such support for hypervisors used for both security isolation and virtualization.

Overall, our approach is comprised of the following components (see Figure 1):

1. A lightweight Hypervisor employed for VM scheduling and hypercall handling. In our prototype, we modified a stable version of the Xen hypervisor[5] which is widely popular and tested.

2. A Controller which runs under the hypervisor and manages resource allocation and limits. This includes device data memory mapping, capture and handling of control requests, and scheduling.

3. A novel native device driver design that builds on the existing driver functionality but supports multiple identities that will be managed by the hypervisor. This will enable the seamless sharing of hardware devices with the hypervisor as a mediator.

In addition to the above components, we utilize I/O hardware assists that already support virtualization. We are also using a minimized kernel like the one used in Xen or Xen aware HVMs (Hardware Virtual Machines) but without the overhead of other management structures like XenStore, Dom0 controller or I/O ring structure. Instead, we use a predefined memory allocation and mapping policy, device ID table setup, a write protection policy customized for every device. Contrary to Xen, we don't depend on a privileged domain that shares the state for all other guests. This prohibits direct or indirect attacks to driver or domain code vulnerabilities. Attackers can only target the integrity of the small and static hypervisor code. Initially, we build one guest OS and then share its memory pages for code among all the other guests. Each guest OS has its own memory containing data. Of course, we only support a specific kernel (*in our prototype implementation, Linux*) and thus, we can share its standard code among all the guests. The resulting architecture offers lower performance overhead than the current Xen model and better isolation than OpenVZ. Any guest can share its read-only (RO) code pages by granting them to the controller which validates and advertises the code via the grant reference. Any other guest can map and appropriately use the pages. Resource allocation details are presented in details in the following sections.

Figure 2 depicts the typical architecture of a current virtual machine monitor. In comparison, our architecture is shown in figure 1.

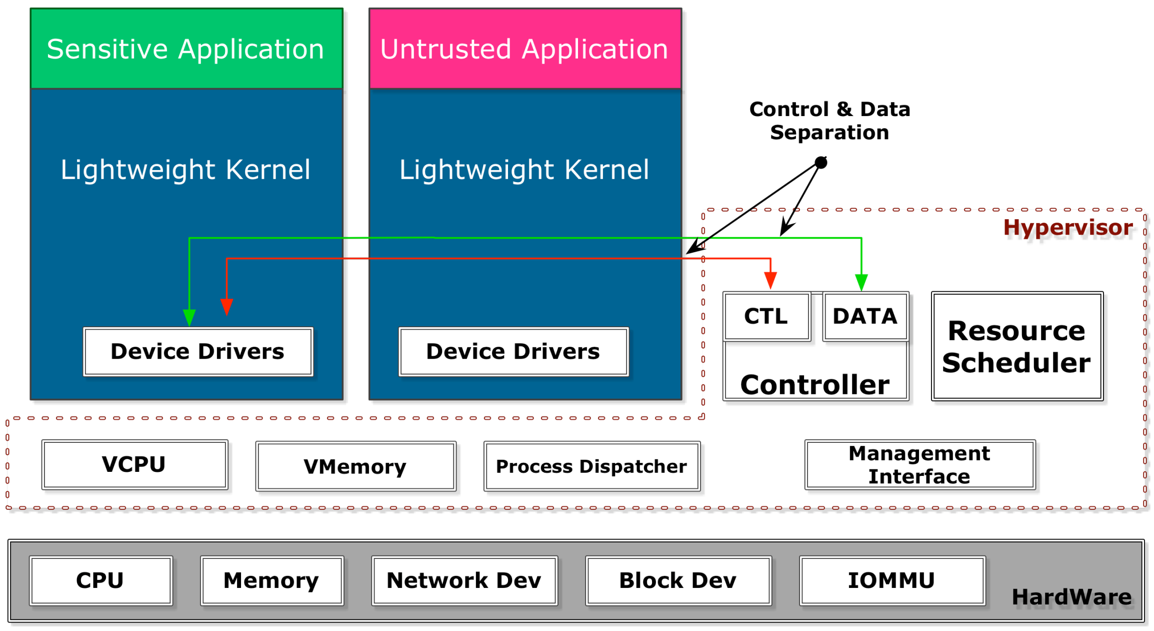

Figure 1: Our proposed Architecture.

2.1 Memory Allocation & Mapping

Currently, the para-virtualized kernel boots in a trusted domain (Dom0). The guest OS, called DomU by the domain builder, is a lower-privilege process running inside the trusted domain. The trusted domain builds initial page tables and loads the kernel image at the appropriate virtual address. Guest OSes are responsible for allocating and initializing page tables for their processes (restricted to read only access). A guest OS allocates and initializes a page and registers it with Xen to serve as the new page table. Direct page writes are intercepted, validated and applied by the Xen hypervisor. The page has a reference count number to count the number of references to the page by the VMs. The page frame can be reused only when its un-pinned and its reference count is zero. Each domain has a maximum and current memory allocation – max allocation is set at domain creation time and cannot be modified. Xen provides a domain with a list of machine frames during bootstrapping, and it is the domain's responsibility to create the pseudo-physical address space from this.

We use the same concept for memory allocation. However, we place the domain builder in the hypervisor as part of the resource controller. For every domain, we use the writable page table model implemented in Xen where the page tables are marked read-only (RO). All the process-level executable code, libraries, and kernel code is shared amoung the domains using physical to machine mapping of the corresponding memory pages and hash verification. Once the controller validates the hash of a page, it is advertised to all the other guests to use. The guests on their part are instantiated with pre-hashed values of executable modules and only need to map to the appropriate memory locations allocated by the controller. The hypervisor has the machine to physical mapping of the entire memory address space and updates a requesting domain, only if the request is permitted by the controller. Similarly, we restrict the hypercalls only for updates, writes and control operations. A guest keeps track of the allocated address space. Any kernel or user-level code is restricted to 3 operations:

1. (R)ead/(W)rite/e(X)ecute memory operations.
2. DMA requests from I/O devices.
3. Interrupts to and from hardware and hypervisor: Physical and Virtual IRQs.

Each of the operations is implemented as summarized below:

R/W/X: We use the hardware support for extented page tables and writable page tables for Read and Execute operations. The controller is responsible for allocation and mapping of memory. It defines the maximum reserve per domain. Any attempt to exceed this reserve would fail. It prepares a startinfo page similar to Xen which might be used during the initial bootstrapping and then the guest takes over with a sharedinfo page. Each guest is permitted to read and execute in its own address space but a write operation has to be validated by the controller. No guest

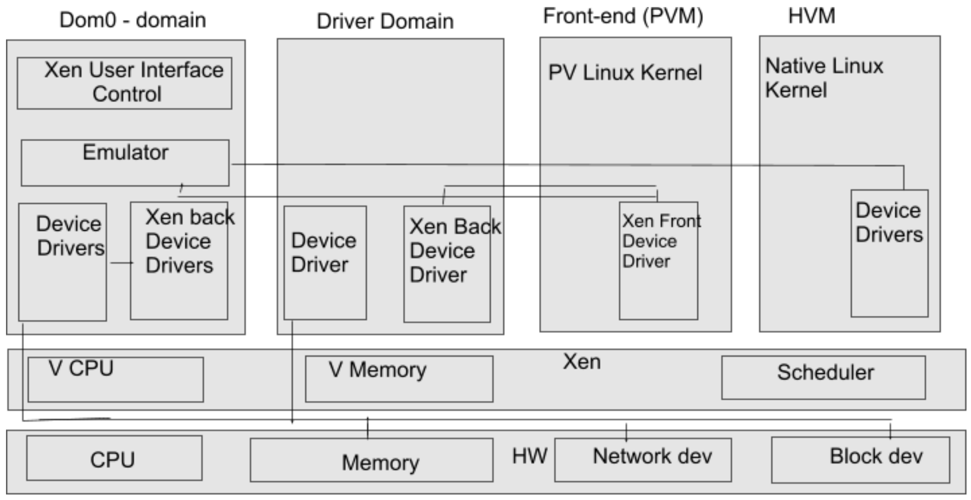

Figure 2: Current para-virtualization (Xen) Architecture.

is permitted to write to any other address space except itself. Any attempt to read or execute pages in a different address space has to be validated by the controller which uses grant reference for locating shared regions.

DMA requests: IOMMU is used to setup the DMA request translation from the guest to the device, direct or persistent mapping[18] policy may be used. The write protection and isolation is achieved in the hardware itself. We employ IOTLB per guest to achieve faster translations. For instance, in a VT-d operation when a device DMA request is intercepted by VT-d engine, a VT-d page table corresponding to that device is walked for valid mapping. VT-d page fault is just for log purpose since the PCI bus doesn't support I/O restart yet.

Physical and Virtual IRQs: We can have the same mechanism of physical and virtual IRQs as it is implemented under Xen. We use the HVM based fast system call mechanism for ring 3 to ring 0 translation. We are also using the VMCALL and VMMCALL for faster hypercall management. We borrow the interrupt remapping mechanism seen on modern VT[7] processors only for hypervisor to guest communication. All the interrupts are delivered when a particular guest is scheduled to run similar to the present mechanism.

Our approach eliminates most of the complexity involved especially in hardware virtualized (HVM) and the para-virtualized (PV) systems that exist in the Xen para-virtualization model. Almost all the security policies can be defined in and enforced by the controller. The Control Manager is an extension of the controller outside the hypervisor used by guests to control and setup resource allocation. Newer devices with virtualization capabilities, such as multiple queue based NICs and self-virtualized devices are supported by our design. The most important aspect of our design is the control and data separation managed by the controller . We understand that most of the legacy devices may not be supporting data isolation - so it is the responsibility of the controller to categorize the data and separate them from the control operations.

2.2 I/O Setup

The device I/O model in current virtualization technologies relies on native device drivers to control the physical devices through a trusted domain. Guest OSes, however, get an emulated view of the virtual devices through an indirection layer designed to multiplex a shared physical device among many virtual machines. If the native device driver, which controls the physical device, happens to contain vulnerable code, then the integrity of the virtual machine is compromised. Even worse, if the native device driver resides in a privileged domain, such as Dom 0 in Xen, then the compromised device driver can control all the virtual machines and the privileged domain. The driver domain model in Xen attempts to limit this threat but without providing any guarantees: a malicious device driver can still affect all the guest OSes that rely on that driver. In addition, there is a performance hit due to the increased indirection through intermediate buffers.

Our aim is to address the above problems by not relying on a single native device driver to control the physical device. We rely on hardware support provided by Intel VT-d and AMD IOMMU technologies and allow the guest OSes to access the physical device directly. At the same time, we add a controller which maintains the context of the device when necessary and maps the context to different guest OSes. In this model, since the most complicated and vulnerable device driver runs in the guest OS that made the request, a compromised device driver has a limited attack surface. In addition, the controller does not contain a full-fledged device driver but only maintains the minimum control of the device. Therefore, the code for controlling the device should be much simpler and easy to verify.

Current hardware supported pass-through I/O is mainly used for exclusively assigning a physical device to a specific guest OS. In our model, we need to share one physical device among many guest OSes with or without device support. We present this idea to the device manufactures so that we can have controller friendly functions.

In general,devices may provide three levels of contexts support for I/O virtualization:

- Full-context support: The device supports multiple contexts for both control and data operations. For example, network cards provide transmit queues with each queue having its own MAC address. In addition, all the control operations should be directed per queue by the device itself. If the number of guest OS's is less than the number of queues, the resource controller is only required to perform the context management by assigning one queue per guest OS. If the number of guest OS's are more than the number of queues, the resource controller has to multiplex the queues for those guest OSes.

- Data-only context support: The device supports only the multiple data contexts but does not support multiple control contexts. For instance, there may be multiple queues per device, but the control operations may not be specifically identified per device queue. It is the responsibility of the resource controller to identify and direct the per queue control operations.

- Legacy devices, no context support: The device does not support multiple data nor multiple control contexts. It is seen in today's mainstream and even virtualized devices.

In all the above three scenarios, the resource controller maintains the mapping (control and data) between the guest OSes and the context on the devices, which is similar to virtual pass-through I/O model suggested by Lei[19]. When the device driver tries to access the device, either through a legacy I/O port or a memory mapped I/O, it will be intercepted by the hypervisor (with support of Intel VT-d or AMD IOMMU). The controller then handles these requests, which is described in the following sections.

2.2.1 Handling Data Access

First, for data access, our controller will have different behavior depending on the hardware:

1. If the device supports full context and the number of contexts are more than the number of actual guest OSes, then the data buffers on the device will be directly mapped to the guest OSes. And the data access will not be intercepted by the hypervisor.

2. If the device supports data-only context, it is similar to full context support.

3. If the device has no context support, then we cannot directly map the buffer on the device to the guest OS. Instead, we may map the buffer on the device to a memory controlled by the controller. The controller also allocates buffers for each guest OS. When the guest OS tries to write the data to the device, it will be intercepted by the hypervisor and written to the temporary buffer maintained by the controller. When the device is idle, the controller can move the data to the device. In this case, the controller needs to have some functions as a device driver so that it can actually move the data. There is another option for some specific devices as well. When the device driver tries to access the legacy device and the device is not available, the controller can inject a virtual IRQ or a virtual event to the guest OS. Since the device driver would be running in a guest OS, it can provide a function to wait for the device when it gets the "device busy" event.

The resource controller will maintain a table of contexts for the device, which is similar to virtual pass-through I/O model [19]. Since the controller will know the context of the device, there are mainly two scenarios when passing data: device is available or device is used by another guest OS. If the device is available, the driver would behave like normal without any issues. If the device is busy, the device driver can provide a `wait_for_device()` function and wait for the device. When the device is available, the driver tries again. As an alternative, the controller can buffer the data or the control operation to a memory location tagged per guest OS and move the data itself when the device is idle.

2.2.2 Handling Control Access

If there is a control access for the device, the controller will behave differently depending on the hardware support. For hardware devices with full context support, the controller can directly pass the control access to the device. Only the management of mappings between the guest OSes and the virtual functions on the devices is handled by the resource controller. In case of legacy devices with no or data-only context support, the controller is required to perform a more complex task when dealing with control commands. It is presumed that the controller has some knowledge of the device and cooperates with the device driver. Therefore, the controller first maintains the device state and then has some logic to determine whether to pass the control command to the device or not. For example, if one device driver in a guest OS attempts to disable the device while other guest OSes are still using that device, the controller should not pass-through the control to the real device. Instead, the controller should build a virtual state table for each guest OS and put the virtual state for the device into "disabled" for that guest OS.

Furthermore, if a sequence of instructions compose one command, then the controller should be able to capture that sequence for a single virtual state and execute the state on behalf of the guest OS. We claim that the functionality required to control the device is a relatively small portion of the total functionality of a typical data-intensive device driver and incur infrequently compared to the data transfer functions. Many other functions, such as interfacing with operating system, are not needed for our controller. To facilitate this separation, the controller can provide a general API for the device drivers so that it can be hardware architecture agnostic.

2.2.3 Example of Logic Flow

Here, we recite the steps for a guest OS through the device driver to access the physical device. Once again, there are three cases corresponding to three levels of device support.

First, let's consider a device with full context support. The steps are shown in the figure 3. If the device has already provided full context support, then the controller just needs to map between the guest OSes and the respective buffer on the device. After that, all the operations can be passed down to the device directly.

Arrow labeled 1 in figure 3 denotes that the data is transmitted directly from the driver in guest OS 1 to the data buffer 1 in the device. Arrow labeled 2 means that data is received directly by the driver in guest OS 2 from the data buffer 2 in the device. The rest of the arrows indicate the control operations, passed directly from per guest OS to the respective location on the device.

For a device with data-only context support, the steps are shown in Figure 4. Each guest OS can be assigned to a specific data buffer on the device. Therefore, the data access will bypass the controller. For the control access, it will be intercepted by the controller and finally transferred to the respective handler. The numerals in the figure can be explained as follows: (1) Data is transmitted directly from the driver in guest OS 1 to the data buffer 1 on the device. (2) Control operations are intercepted by the hypervisor and looked up by the controller.(3) Lookup table in the controller matches the guest ID with the appropriate handler. (4) Received data is directly passed from the device buffer 2 to the driver in guest OS 2.

Finally, if the device has no context support, the typical sequence when the I/O is instantiated by a guest OS is shown in the Figure 5.

This is the worst case scenario in terms of performance and it involves a no-context support device with a legacy driver without support for control or data isolation. The logical flow is as follows: (1) Operation of device driver makes a page fault intercepted by the controller in the hypervisor. (2) Based on the type of operation - control or data, the matched ID for buffering is obtained. (3) Either actual data operation takes place if the device is free or a VIRQ is generated back to the device driver if busy. For control operation, the function call is manipulated to reflect a per VM operation. (4) Request is received.(5) Based on the received data - lookup takes place in the data table. (6) Data packets are buffered if the VM is not available or sent up if available.

Usually, the device channel in Xen uses inter-domain shared memory using grant tables and event channels to emulate the functionality of a device. The combination of the shared memory containing a ring buffer for requests and responses and the event channel provides the facilities for the domains to talk to each other. There are at least two different types of event "channels": interrupt notification (upper call or uni-directional) and notification of queued descriptors (bi-directional). To implement our prototype model, we modified the current pass-through mechanism provided by Xen. In the current pass-through model of Xen, one PCI device can be exclusively assigned to a Dom U if the hardware supports I/O virtualization. After that, even the privileged domain cannot access that device. To achieve sharing among multiple guest OSes, we employ the pass-through model and add the per device map in the controller which can export it to a requesting VM. Then the device can be assigned to multiple VMs and the controller can then context switch the device among them.

2.2.4 Resource Management

The Virtual Machine Monitor is a thin software layer designed to multiplex hardware resources efficiently among virtual machines. The VMM, in our case the resource controller, manages system hardware directly, providing high I/O performance and complete control over resource management. This control interface, together with profiling statistics on the current state of the system, is exported to a suite of kernel level management software running in per guest OS. This complement of administrative tools allows convenient management of the entire system. The controller can create and destroy domains, monitor privileges, set network filters and routing rules, monitor per-domain network activity at packet and flow granularity, and allocate and safeguard all host-based resources. This

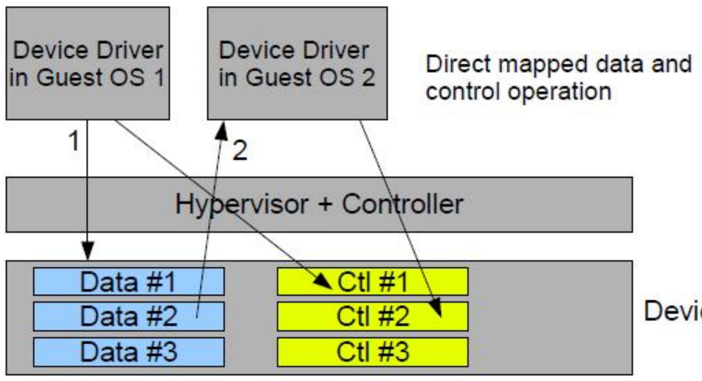
Device Driver in Guest OS 1
Device Driver in Guest OS 2
Direct mapped data and control operation
1
2
Hypervisor + Controller
Data #1
Data #2
Data #3
Ctl #1
Ctl #2
Ctl #3
Device

Figure 3: Necessary communication steps for a device with full context support.

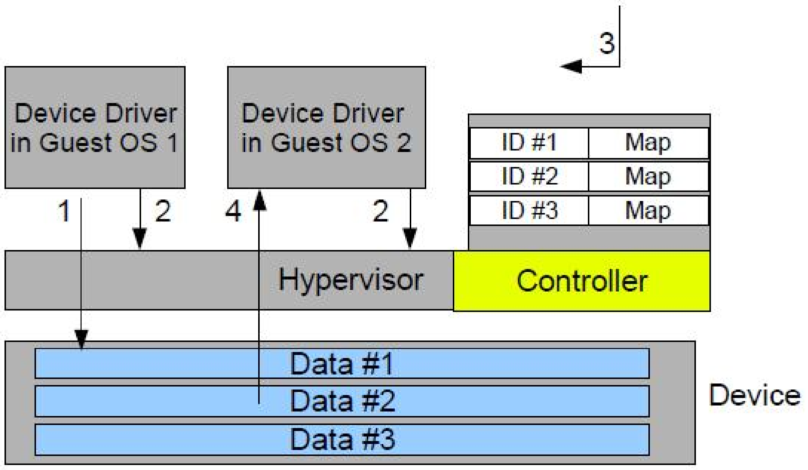

Figure 4: Necessary communication steps for a device with data-only context support.

resource management includes storage, processing, I/O, memory, and network described in this order in the next paragraphs.

All the VMs access persistent storage directly and are validated by the Controller. Each VM has a device ID table to map the device to its respective memory location allowing the VMs to manage the storage. This keeps the mechanisms very simple and avoids more intricate solutions such as VBDs used in the PV Xen model. A typical guest OS disk scheduling algorithm will re-order requests prior to en-queuing them in an attempt to reduce response time, and to apply differentiated service. A translation table is maintained by IOMMU for each guest OS; the entries within this table are installed and managed by the Controller. On receiving a disk request, the translation takes place if the request is validated by IOMMU mapping. Permission checks also take place at this time. Zero-copy data transfer takes place using DMA between the disk and pinned memory pages in the requesting domain. Since each VM has a pre-defined disk space and can behave independently like a full OS. Each VM allocated disk space can have its own quota tools. It is possible to increase the VM disk space but may be difficult to do it dynamically. However since we are relying on the per VM specific applications, we may not actually need to increase the VM disk space any more than its initial allocation.

We are using the Credit based time unit allocation of VCPU, which enforces fair sharing of CPU resources. We of course can also have the standard linux kernel scheduler within each VM to allow per process time quantum allocation. Furthermore, in a typical container based system, each container is assigned an I/O priority, and the I/O scheduler distributes the available I/O bandwidth accord-

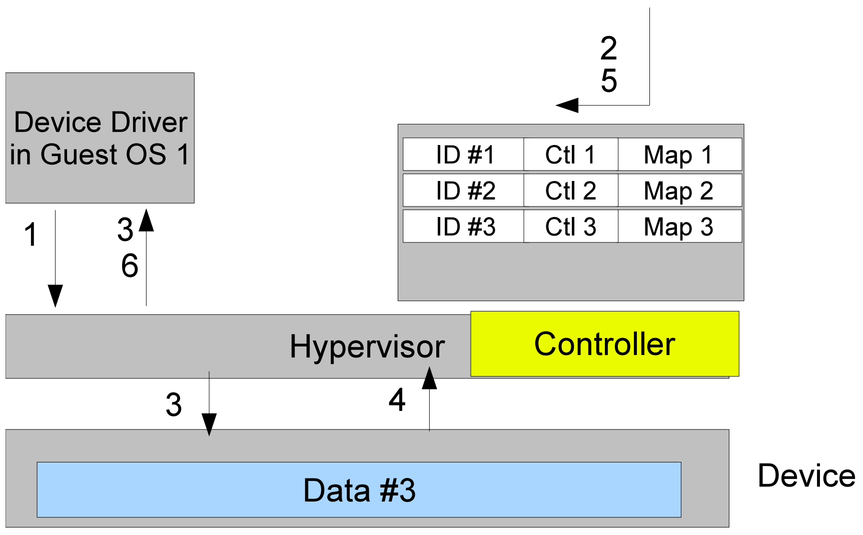

Figure 5: Necessary communication steps for a device with no context support

ing to the priorities assigned. Thus no single container can saturate an I/O channel. However, for our system we rely on the fact that all the VMs would have equal I/O priority. So VMs can be scheduled in a round robin fashion for I/O bandwidth. The Xen model actually employs the same mechanism, the only difference is that complete I/O bandwidth control is under Dom0 in Xen which responds to the requests coming in from the ring buffer. Whereas for our system the controller does the same in sync with the CPU scheduler.

We use pre-mapping with hard limits for each VM, no VM can exceed its allocated address space. Secondly since we use grant references for sharing only the RO code, we keep the sharing limited to code and not data. Now with the idea of multiple data buffers in the controller or hardware and IOMMU support for disk, we try to segregate the data per VM. So in other words, there's hardly any writable data being shared in our system. Hence we do not need the use of Bean counters. However, we can use them if the need so arises.

The resources shared between guest OSes take into consideration common shared objects. Hence, the controller maintains a state space representation for per VM shared resource. Every time a VM acquires a resource the controller follows the control flow of the state space for I/O operation. Once the VM is scheduled out the controller saves the updated state space and loads the new state flow of the running VM.

For network operations there are "hot" pages in memory that are involved in repeated I/O operations. For example, a software cache of physical-to-machine page mappings (P2M) associated with network transmits is augmented to count the number of times each page has been copied. When the count exceeds a specified threshold, the page is transparently remapped into frequently used memory. This scheme seems to be effective with guest operating systems that uses a limited number of pages as network buffers. The decision to remap a page into frequently used memory increases the demand for those pages, which may become a scarce resource. It may be desirable to remap some frequently used pages into normal memory, in order to free up sufficient frequently used pages for remapping I/O pages that are currently "hot".

3 Comparison to Related work

Our architecture borrows most concepts from Xen [5] including the hypervisor module, memory write protection, grant reference for sharing, and interrupt remapping. However, we do not rely on the privileged domain (Dom 0) or the IOEMU (emulator) based design seen in the modern versions of PVM and HVM of Xen. Instead, we use a controller in the hypervisor and customize the direct pass-through model for sharing of devices. Also we don't use any domain initialization and we do not require a domain builder. Contrary to Xen, we implement memory initialization and mapping inside the hypervisor using the resource controller. In addition, we prevent inter-domain communication via XenStore or event channel communication models. We only support network communication between the domains to implement isolation using memory-mapped network interfaces. Finally, we would be relying on hardware assists not just for data isolation - but for controller supported code isolation as well. Our native device driver support is inspired by container-based isolation, such as Solaris Zones [11], FreeBSD Jails [8] , OpenVZ [2], VServer [17] and FVM [20]. In general, process virtualization provides superior performance compared to VMM [17] model.

Traditionally, hypervisors use a native device driver in the privileged domain or the host to access the physical device, and then emulate multiple virtual devices for guest OSes or use split drivers to allow the guest OSes to access the device (such as in Xen). If the device driver in the driver domain [6] is compromised, it will affect all the other related domains. Newer versions of hypervisors can utilize the hardware support to let the device driver in the guest to access the physical device directly. For example, Xen supports pass-through I/O which can exclusively assign a PCI device to a specific domain. However, the device cannot be shared by multiple guest OSes with this method. Another technique uses self-virtualized device and allows sharing one physical device among multiple guest OSes [12]. But it works only with support of the device. Virtual pass-through IO [19] tries to share a legacy device among multiple guest OSes by context switching the device.

More recently, TwinDriver model[9] proposed to split the device driver into a guest OS part and a hypervisor part aiming to achieve good performance without enhanced security. Our approach does not have any part of the driver data path inside the hypervisor but rather we target to improve the safety for drivers running inside the guest OS. We remove the driver domain and schedule the device in the hypervisor, so that even if a guest OS becomes compromised, the other guest OSes can still use the device as long as the hypervisor holds its integrity. To that end, we propose a general I/O model which separates the handling for control and data. It can work with devices with full context support as well as devices without any context support. We do require some modifications of device drivers to enforce some control functions inside the hypervisor and make the device driver hypervisor-friendly.

sHype [13] is a hypervisor security architecture developed by IBM Research, in various stages of implementation in several hypervisors. The goal behind sHype is based on isolation, mediated sharing and mandatory policies similar to SELinux [16]. However sHype relies heavily on attestation and authentication for integrity guarantees. It also believes in the Trusted Computing Group and controlled monitoring of all the resources. But since it is primarily focused on security of the hypervisor and not its performance, it still relies on the native hypervisor based VMM model, which adds an overhead. Besides with all these policies, the complexity of the VMM is further increased. In contrast, our system mainly focus on isolation of I/O devices, and it can utilize the hardware with full context support to directly assign a device to multiple guest OSes. The hardware can thus enforce the isolation. Moreover, we use a controller to control the device access when the hardware does not provide enough support.

SecVisor [15] uses a hypervisor to enforce the code isolation between the user-space and kernel-space. However, it does not support multiple guest OSes. Loki [21] system uses tagged memory to enforce application policies on hardware. In contrast, we try to isolate applications by putting them into their own protection domains. Jose Renato Santos et. al. [14] proposed a general model for network devices. Our model is more general, encompassing all I/O devices and allows layered deployment supporting both legacy and hardware-assisted I/O.

4 Conclusion

Current virtualization schemes exhibit a trade-off between the degree of isolation and the system performance: light-weight container systems offer weak isolation whereas the strong isolation of full or para-virtualization systems suffer from poor I/O performance due to the use of virtualized drivers. In this paper, we propose and analyze a new architecture that attempts to use recent advances to provide enhanced kernel isolation without sacrificing performance.

Our system borrows concepts from existing technologies and recent advances in hardware-assisted I/O mapping to isolate full kernels including their device drivers. We desiged a layered architecture that can support legacy devices by implementing the necessary functionality in the hypervisor with a performance penalty. To amend this, we advocate for new devices and device drivers that can support multiple identities leaving only the identity management to the hypervisor. We posit that our work provides a compelling security and performance argument towards that direction.

References

[1] IOMMU Architectural Specification. Advanced Micro Devices, Inc.

[2] Openvz. http://wiki.openvz.org.

[3] Vmware virtualization products. http://www.vmware.com/products/.

[4] Secure Virtual Machine Architecture Reference Manual, Advanced Micro Devices,Inc, 2005.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS ASPLOS 2004 workshop*, 2004.

[7] R. Hiremane. Intel® Virtualization Technology for Directed I/O (Intel® VT-d). *Technology© Intel Magazine*, 4(10), 2007.

[8] P. Kamp and R. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[9] A. Menon, S. Schubert, and W. Zwaenepoel. Twin-drivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 301–312, New York, NY, USA, 2009. ACM.

[10] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation Due to copyright restrictions we are not able to make the PDFs for this conference available for downloading*, pages 361–376, New York, NY, USA, 2002. ACM.

[11] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *18th Large Installation System Administration Conference (LISA 2004*, 2004.

[12] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188, New York, NY, USA, 2007. ACM.

[13] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. *Computer Security Applications Conference, 21st Annual*, pages 10 pp.–, Dec. 2005.

[14] J. R. Santos, Y. Turner, and J. Mudigonda. Taming Heterogeneous NIC Capabilities for I/O Virtualization. In *Workshop on I/O Virtualization (WIOV 2008)*, 2008.

[15] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[16] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 1:43, 2001.

[17] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 275–287, New York, NY, USA, 2007. ACM.

[18] P. Willmann, S. Rixner, and A. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *USENIX Annual Technical Conference*, 2008.

[19] L. Xia, J. Lange, and P. Dinda. Towards Virtual Passthrough I/O on Commodity Devices. In *Workshop on I/O Virtualization (WIOV 2008)*, 2008.

[20] Y. Yu, F. Guo, S. Nanda, L. Lam, and T. Chiueh. A feather-weight virtual machine for windows applications. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments*, volume 14, pages 24–34, 2006.

[21] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI '08: Proceedings of the 8th symposium on Operating systems design and implementation*, 2008.