RC23537 (W0502-132) February 25, 2005
Computer Science

IBM Research Report

Building a General-Purpose Secure Virtual Machine Monitor

Reiner Sailer, Trent Jaeger, John Linwood Griffin, Stefan Berger, Leendert van Doorn, Ronald Perez, Enriquillo Valdez

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Building a General-purpose Secure Virtual Machine Monitor

Reiner Sailer Trent Jaeger John Linwood Griffin
Stefan Berger Leendert van Doorn Ronald Perez Enriquillo Valdez

{sailer,jaegert,jlg,stefanb,leendert,ronpz,rvaldez}@us.ibm.com
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA

ABSTRACT

Recent advances in the hardware available for commodity computer systems are enabling the construction of virtual machine monitors (VMMs) that provide complete isolation between virtual machines (VMs). This paper predicts that the availability of this isolation will increase the demand for VMM-based systems running mutually distrusted coalitions of VMs. Because the VMM systems can provide reliable isolation, some controlled sharing responsibilities of operating systems will be moved to the VMM, where practical; we investigate the efficacy of providing such controls in the VMM in this paper.

This paper describes the design of the sHype security architecture, carefully considering which virtualizable resources are appropriately controlled by the VMM. sHype enables control of these resources using a system-wide mandatory access control (MAC) policy. One sHype design goal is to permit the hypervisor to retain a very stable, near-minimal code base, allowing significant security assurances (e.g., Common Criteria) to be achieved. Notably, this paper argues that it is not necessary to aim for the highest levels of assurance when designing secure VMMs for commodity hardware—when *absolute* isolation is required (e.g., the total prevention of covert timing channels), a multi-system, separate hardware architecture is recommended. Finally, this paper describes an implementation of the sHype architecture controlling virtual LAN (vLAN) resources in a fully-functional research hypervisor.

1. INTRODUCTION

As general-purpose workstation- and server-class computer systems grow in available power and capability, it becomes more attractive to aggregate the functionality of multiple standalone systems onto a single hardware platform. For example, a small business that originally used three computer systems—perhaps to take customer orders using a web server front-end, a database server in the middle, and a file server back-end—can reduce the required physical space, configuration complexity, management complexity, and overall hardware cost by running all three applications on a single system. Taking this one step further, several small businesses could achieve an even lower-cost solution by contracting out the management of their respective business computing applications to a centralized server managed by a nonpartisan third party.

This idea of *virtualization* of standalone computer systems on a single system has been around for decades [1, 2], often being employed in "big iron" mainframe systems whose hardware was explicitly designed with virtualized operation in mind. However, until recently it has not been feasible to build systems out of commodity PC hardware that meets the security guarantees required by mutually distrusted parties—i.e., that the data and execution environment of one party's applications are securely *isolated* from those of a second party's applications. For example, such systems were often vulnerable to DMA attacks where one party's application could break isolation by issuing DMA instructions to effect a copy into or out of the memory used by the second party's applications. Such systems were vulnerable no matter what software mechanisms were used for isolation—whether the property was enforced by the operating system, or by a virtual machine monitor (VMM) controlling multiple virtual machines (VMs).

Emerging technology, such as the I/O-MMU, eliminates these previous limitations on isolation for commodity systems and makes it feasible to ensure a VMM can control *all* memory accesses, especially those between mutually distrusted parties. This development, combined with the inability to make definitive statements about resource sharing among heterogeneous and potentially mutually distrusted operating systems running as guests in VMs, motivates us to claim that VMMs will not only need to provide isolation, but also they will need to provide a basis for control of information flows and sharing of resources among VMs which was formerly expected of operating systems.

As we recognize that one of the strengths of a well-designed VMM is a minimal software base—i.e., building only as much code and functionality into the VMM to accomplish its goals, in order to facilitate code inspection and behavioral verification—we are not proposing that a VMM take on all characteristics of a full-fledged operating system. Rather, we simply assert that a system's security will depend on VMM control and mediation of inter-VM resources. This is not an entirely new notion, as the VAX VMM systems provided

high assurance (e.g., Orange Book A1 [3] or Common Criteria EAL7 [4]) mediation of resources among VMs [5]. Also, the Terra trusted VMM uses a management VM to arbitrate resource usage among VMs [6]. However, we believe that changes in hardware support and trade-offs in how virtualization can be implemented need to be re-examined to justify the design of a VMM security architecture that best balances system flexibility, minimization of trusted code, and performance impact. For example, the VAX VMM predated I/O-MMUs, so drivers had to be included in the VMM which greatly limited system flexibility. With the respect to virtualization decisions, we can choose to support virtualized services either in the VM itself, a separate, trusted VM, or in the VMM. Each of these choices has performance and security considerations.

The problem we address in this paper is the design of a VMM *reference monitor* that enforces a comprehensive, mandatory access control (MAC) policies on inter-VM operations. A reference monitor is defined to ensure mediation of all security-sensitive operations, which enables a policy to authorize all such operations [7]. A MAC policy is defined by system administrators to ensure that system (i.e., VMM) security goals are achieved regardless of system user (i.e., VM) actions. This contrasts with a discretionary access control (DAC) policy which enables users (and their programs) to grant rights to the objects that they own. In this paper, we will focus on the design of the reference monitor interface, rather than the MAC policy. A reference monitor design that mediates all inter-VM resource usage will enable any MAC policy to be correctly enforced.

Our reference monitor is the basis for the sHype hypervisor security architecture. A hypervisor is a para-virtualization VMM, where modest (e.g., tens of lines of code) operating systems changes are made to enable the OS cooperation. sHype implements a reference monitor interface for (1) isolating virtual machines by default and (2) sharing resources among virtual machines when desired according to MAC policy. We examine which resources need to be controlled based on an analysis of security, flexibility, and performance properties. We describe how access control can be done in a manner that has low impact on VM performance. Finally, we demonstrate the enforcement using simple MAC policy based on coalitions of VMs.

We implemented the core hypervisor security architecture in an existing research hypervisor rHype [8] and, as a proof-of-concept, demonstrated access controls for the virtual network (LAN). Our modifications to the hypervisor are small, about 1000 lines of code. Extending access control to the remaining virtual resources will require only a few lines of code. The secure hypervisor architecture is designed to achieve medium assurance (Common Criteria EAL4 [4]) for hypervisor implementations. Our security-enhanced research hypervisor achieves near-zero security-related overhead on the performance-critical path.

Section 2 introduces the typical structure of a hypervisor environment for which we have developed a generic security architecture. Mutually suspicious applications and runtimes serve as an example to illustrate requirements and the use of our hypervisor security architecture. We introduce the design of the sHype hypervisor security architecture in Section 3 and its implementation in Section 4. Section 5 illustrates the use of sHype and Section 6 describes the related work.

2. PROBLEM STATEMENT

This work addresses the design and implementation of secure inter-VM communications in a hypervisor environment, with specific emphasis on four design issues: (1) strict hypervisor enforcement of a MAC policy for all inter-VM communications; (2) support for efficient communication among coalitions of VMs as defined by the MAC policy; (3) minimal communication-time overhead introduced by the run-time policy checker in the hypervisor; and (4) the maintenance of strong isolation between any VM pairs that are not otherwise covered by the MAC policy.

This section begins by describing the need for hypervisor-controlled enforcement of resource sharing among VMs, and, by extension, the need for an in-hypervisor reference monitor. We then describe the characteristics and role of a reference monitor, and examine the problem of designing a hypervisor reference monitor to support coalitions of VMs. We close the section by identifying the system evaluation criteria that are useful for selecting among reference monitor design options.

2.1 Coalitions of VMs

In the near future, we believe that VM systems will evolve from a set of isolated VMs into sets of VM coalitions. The main reason for this is that as the reliance in hypervisors increases due to hardware improvements enabling reliable isolation, we believe that some control now done in operating systems will be delegated to hypervisors. We aim for hypervisors to provide isolation between coalitions and provide limited sharing defined by a system-wide MAC policy within coalitions.

Consider a customer order system. The web services and data base infrastructure that processes orders must be high integrity in order to protect the integrity of the business. However, the read-only operations advertising products and collecting possible items to be purchased need not be as high integrity. For example, an OEM's code advertising a product that the company distributes may be run in such a VM. Obviously, we would like the latter system to work also, but clearly the performance and flexibility requirements have greater weight than the security requirements.

Such a view is analogous to the concept of *privilege separation* [9]. A privilege-separated system consists of sets of privileged and unprivileged components (e.g., VMs). The privileged components can perform certain operations (e.g., use certain virtual resources) that the unprivileged cannot and they are entrusted with protecting those resources. However, the unprivileged components can communicate through limited interfaces with the privileged components. The security protection of such interfaces should be verifiable, so that the company can ensure that no unprivileged inputs can compromise the integrity of the privileged component and that the privileged component protects the secrecy of its secret, privileged data. The *analytic integrity model* [10] provides an example of such security guarantees in operating systems, but the smaller trusted computing base of hypervisors makes them a more attractive basis.

In the customer order example, the coalition of VMs performing customer orders are separated from the other VMs on the system. We separate them into the *order* VMs and the *other* VMs. The order VMs may communicate, share some memory network, and disk resources. Thus, they are coalition and protected from other VMs completely by the

hypervisor. Within the Order VM coalition, the hypervisor controls sharing using a MAC policy that permits inter-VM communication, sharing of network resources and disk resources, and some sharing of memory. All this sharing must be verified to protect security of the order system. However, the MAC policy also enables the hypervisor to protect the order data base memory from being shared with the unprivileged components.

2.2 Reference Monitor

The key component of the security architecture is the *reference monitor* which enforces system access control policy. The classical definition of a reference monitor [7] states that it possesses the following properties: (1) it mediates all security-critical operations; (2) it can protect itself from modification; and (3) it is as simple as possible.

First, a security-critical operation is one that requires MAC policy authorization. If such an operation is not authorized against the MAC policy, the system security guarantees can be circumvented. For example, if the mapping of memory among VMs is not authorized, then a VM in one coalition can leak its data to any other VM.

We identify security-critical operations in terms of the *MAC resources* whose use must be controlled in order to implement MAC policies. We must also identify the location of the *mediation points* for these resources. The combination of MAC resources to be mediated and mediation points forms the *reference monitor interface*.

Second, the reference monitor must protect itself. Fundamentally, this requirement implies that the hypervisor must protect itself from unauthorized modification. Further, some VMs may be entrusted with some reference monitor tasks. For example, a hardware device may be shared between two coalitions, so the VM serving the device must also control data flows to these coalitions. VM enforcement may also be necessary because the code is too complex to include in the We call these VMs entrusted to enforce MAC policy, *MAC VMs*.

Because the hypervisor is more fundamental to the system than any MAC VM (e.g., the hypervisor can always restart a MAC VM), we distinguish between hypervisor self-protection and MAC VM self-protection. The basis of self-protection is a *trusted computing base* (TCB), so we identify two distinct TCBs: (1) a *TCB-VMM* upon which the hypervisor protection depends and (2) a *TCB-MAC* upon which the MAC VM protection depends.

Third, simplicity of design is determined by the complexity of the reference monitor interface and the complexity of the MAC policy. Because a reference monitor interface that meets the mediation requirement (1) can enforce any MAC policy, we do not consider any specific MAC policy in this paper (see references for prominent examples of MAC policies [11, 12]). Thus, the simplicity is dependent on the reference monitor interface only. We aim to control only the minimal set of resources necessary to enforce a MAC policy. If this is achieved, then the reference monitor can be made sufficiently simple.

We will also consider the impact of the reference monitor on overall system function in Section 3.

2.3 Resources and Mediation Points

Resources that impact MAC policies are those that enable inter-VM information flows. That is, use of these resources may enable one VM to communicate information to another VM. For example, memory may be shared between two VMs unless there are controls on memory distribution. Historically, both *overt* and *covert* information flows have been addressed in VMM designs. An overt information flow uses resources as the communication medium, whereas covert information flows result from observing system behaviors that are not defined as resources. We discuss these two types of resources separately below.

2.3.1 Overt Information Flows

We divide overt resources into the following categories: (1) hypervisor resources; (2) VM resources; (3) virtualized system resources; and (4) physical system resources.

Hypervisor resources include the *CPU*, *I/O memory* (I/O-MMU), and *hypervisor memory* that are necessary for the hypervisor itself to run. The hypervisor must control these itself.

VM resources include *inter-VM communication (IPC)*, *VM memory*, and *VM* objects themselves that enable VMs to be created and communicate. All could be implemented by VMs, so we have a choice. The choices depend on size of trusted code (e.g., small for IPC and large for VM creation) and flexibility requirements (e.g., low for IPC and sometimes useful for user-level VM). Performance is generally better if the hypervisor serves these resources due to context switch times.

Virtualized system resources include *virtual LANs*, *virtual block devices*, *virtual TTYs*, etc. that support virtualization of the single physical platform across multiple operating systems. Again, these can be implemented in either the hypervisor or VMs. Since these are indirect resources, code size is the major factor. For example, vLAN implementations are generally small whereas a vSCSI device is complex. Flexibility is not an issue, and performance considerations are the same as for VM resources.

Physical system resources differ from virtualized resources in a couple of key ways: (1) I/O-MMUs removes the need to trust such devices and (2) performance is best if the devices are co-located with the code using them in the same VM. Thus, the optimal case is a physical resource per VM, which may not be practically feasible. Driver code is too complex for inclusion in the hypervisor, so a device that is to be shared by multiple coalitions requires a MAC VM.

2.3.2 Covert Information Flows

Covert information flows are categorized into two types: (1) *storage covert channels* and (2) *timing covert channels*. A storage channel uses storage limits on devices shared by multiple coalitions (e.g., via a MAC VM) to leak data. If the VM a can fill up the disk in a manner that VM b can detect, then a communication from a to b is possible. Storage channels are typically closed by preventing the use of shared devices or by providing limits on use such that detection of use is not feasible. It is often feasible to close storage channels. A timing channel uses other resources as virtual clocks to measure modulated system behavior. Schaefer [13] and Karger [14] show that disk arm optimization can allow timing channels based on the sequence of completed write requests. Timing channels are thwarted by normalizing performance across all operations to the slowest operation in which a timing channel is possible (e.g., fuzzy time [15]). Timing channels have a tremendous performance and sys-

tem flexibility impact, so almost any timing channels is much more expensive to address than the solution of moving the coalitions to separate machines.

2.3.3 *Mediation Points*

Once we know which resources require control, we must consider the mediation points for controlling these resources. There are three ways in which resources can be controlled: (1) in the hypervisor; (2) in a MAC VM; and (3) upon the communication with an untrusted, general-purpose VM.

In the first two cases, the hypervisor and any trusted VMs serve the objects and control access to them. This is the model of the Flask security architecture [16] where resource managers are responsible for authorizing access to the resources that they serve. Unlike the Flask security architecture, not every VM can be a resource manager. Only the hypervisor and MAC VMs trusted to implement the MAC policy may be MAC resource managers. We do not prevent other VMs from serving resources, but their actions are limited by the MAC policy.

In the last case, the hypervisor and/or a MAC VM controls communication between VMs. In this case, the VM effectively serves objects that all have the same MAC label. For example, a general-purpose VM memory server can be used to map of its own label to other VMs that are permitted by the MAC policy to map pages of that label. It does not matter which individual page is served, as pages with the same MAC label are in the same equivalence class from a security perspective.

2.4 Design and Evaluation

The reference monitor design task is to determine the resources to be controlled by the MAC policy and to identify the mediation points for enforcing that policy. The design should be evaluated to determine that it is the better (or no worse) than another possible reference monitor design. Since a global assessment across all resources is not likely to be feasible, we propose assessment for the cross-product of all distinct resource types and mediation points. Thus, we can examine the impact of the system evaluation criteria for each resource and reasonable mediation point. We discussed the approach to collecting the resources and choosing the mediation points above, so in this section we discuss the evaluation criteria.

From a security perspective, the quality of the reference monitor design is based on the size of the trusted computing base. In Section 2.2, we distinguished between the hypervisor's TCB and the MAC VM's TCB. The minimization and stability of the former is a higher priority than for the latter because the fundamental integrity of the system depends on the hypervisor.

Further, we identify that a reference monitor design must meet other system design criteria, in particular system flexibility, performance overhead, and resource utilization. Flexibility implies changes in implementations or system behavior policies. In most cases, the hypervisor provides no flexibility. Performance overhead is a comparison of the system performance metrics, such as throughput and latency. Concrete measurements should be used where available, but relative performance comparisons based on system architecture is often possible. Resource utilization implies the amount of physical resources consumed. For example, control at a MAC VM may incur more resource utilization in order to prevent storage channels than a hypervisor that may be assured to prevent such channels. Also, mediation point (3) where communication is mediated and devices are served from one VM per coalition may result in significant costs for independent devices.

3. DESIGN

In this section, we outline the design of the sHype security architecture. We first describe the system architecture that results from an sHype system. Next, we describe how sHype reference monitors are implemented in the hypervisor and MAC VMs. Finally, we examine the choice of mediation points for some key system resources based on the evaluation criteria discussed in Section2.4.

3.1 System Architecture

Figure 1 illustrates a canonical system based on the sHype security architecture. The hypervisor creates VMs which are virtual copies of individual systems running on the platform. The hypervisor defers the handling of specific I/O devices to a MAC VM VM0, sometimes called the I/O VM. VM 1 and VM 2 run guest operating systems (e.g., Linux). Guest operating systems running on sHype are minimally changed to replace access to essential but privileged operations with specific hypervisor calls. Such privileged operations cannot be called directly by guests because they are powerful enough to compromise the hypervisor. In general, hypervisor calls implemented in the hypervisor have three characteristics: (1) they offer access to purely virtual resources (e.g., virtual LAN); (2) they speed up critical path operations such as page table management; and (3) they emulate privileged operations that are restricted to the hypervisor but might be necessary in guest operating systems as well. The hypervisor can, under some circumstances, regain control over (revoke) resources already allocated.

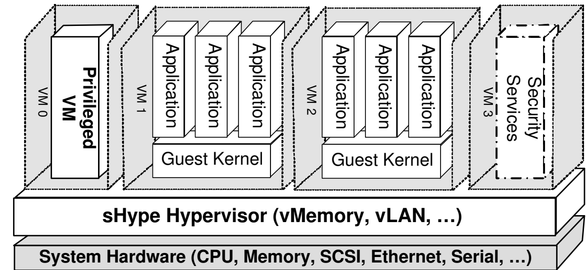

Figure 1: Canonical system on an sHype security architecture

Security services run in separated and trusted run-time environments (VM 3) in Figure 1. As an example of a security service, we will introduce a policy management service that manages the formal rules describing access authorization of VMs to shared resources in our sHype security architecture. Other security services could include auditing or integrity attestation.

3.2 Reference Monitor

sHype strictly separates access control enforcement from the access control policy according to the Flask [16] archi-

tecture. We describe the control architecture in the context of the hypervisor, but it also is used in the MAC VMs.

Figure 2 shows the sHype access control architecture design as part of the core hypervisor and depicts the relationships between its three major design components. *Enforcement hooks* implement the reference monitor. They are distributed throughout the hypervisor and cover references of VMs to virtual resources. Enforcement hooks retrieve access control decisions from the *access control module* (ACM).

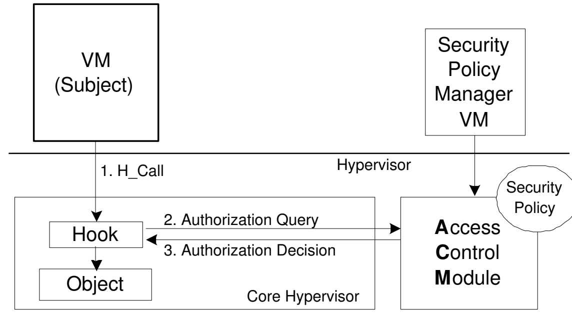

Figure 2: Hypervisor-based security reference monitor.

The ACM applies access rules based on security information stored in security labels attached to VMs (subjects) and resources (objects) and the type of operation to make an access control decision. The *formal security policy* defines these access rules as well as the structure and interpretation of security labels for VMs and logical resources. Finally, a hypervisor interface enables trusted policy management VMs to efficiently manage the ACM security policy.

Our sHype access control architecture is designed to meet the following three practical requirements:

- high performance ($\leq$ 1% security-related overhead on the critical path)
- ability to enforce policies autonomously (without assuming co-operation of general VMs)
- allow for the flexible enforcement of various MAC policies, which guard the information flow between multiple VMs

The position of a hypervisor in the system software stack mandates the *highest possible performance* because any penalty will affect the VMs, which depend on the hypervisor for many privileged operations (e.g., memory management, scheduling, or resource sharing among VMs). Consequently, a security architecture that introduces non-negligible overhead will not become a core component of a hypervisor. To minimize performance overhead, we perform access control decisions are resource *bind* time (e.g., when connecting a virtual Ethernet adapter to a logical LAN), rather than on each resource access.

As described above we only control access to resources that enable inter-VM communication. In general, the coarser the granularity of the resources, the less code that will MAC enforcement will depend. We have a trade-off between the flexibility of MAC VMs and amount of code that have to depend on. For example, if a MAC VM serves data base records for multiple coalitions, then the MAC controls may have to describe individual records. The sHype architecture enables such decisions to be made, such that MAC control at the granularity desired is possible.

To support various business requirements, a security architecture must support *various kinds of MAC policies*. The sHype architecture supports Biba [11], Bell-LaPadula [12], Caernarvon [17], as well as other MAC security policies.

3.3 Resource Mediation Points

Table 1 shows the relationship between: (1) the resources and their possible mediation points and (2) the impact on system evaluation criteria compared to a reference case. The table provides an abstract comparison between factors that identify preferred mediation points for system resources. Note that the table is not exhaustive, but represents at least one member of each resource aggregate discussed in Section 2.3. The evaluation criteria are taken from Section 2.

The cases marked with either <<< or >>> are the most important. These indicate either a significant increase >>> or a significant decrease <<< in the dimension. For example, mediation of the I/O-MMU in a MAC VM greatly increases the size of the TCB that the hypervisor depends upon for its protection. As a result, it is only reasonable to consider I/O-MMU mediation in the hypervisor.

For other resources, first the differences of VM memory management implemented in a VM compared to its implementation in the hypervisor is large if this VM implements a general-purpose operating environment; the difference is smaller if this VM implements a minimal run-time environment. Note that VM memory management does not include hypervisor memory, which must be controlled by the hypervisor or the hypervisor TCB would increase dramatically. Second, we note the difference between the two virtualized resources, vLAN and vBlocks. Since the vLAN implementation is simple, it can be added to the hypervisor. Note that the MAC VM TCB would grow significantly only if a new MAC VM would have to be added. vBlocks on the other hand would greatly increase the hypervisor TCB. Lastly, we note that I/O devices have the best performance if they are in the same VM as the request. Otherwise, context switches are necessary which add significant overhead. On the other hand, the resource requirements may make this design impractical because a device per coalition may not be possible.

4. IMPLEMENTATION

First, we describe the rHype isolation properties on which sHype builds. Then, we describe the sHype reference monitor implementation including the security policy and how it is used to make access control decisions.

4.1 Isolation Properties

Access control in sHype depends on the strong isolation between virtual resources (VMs) and between the hypervisor and the virtual machines, i.e. the code running in them.

The hypervisor protects itself against malicious programs running in VMs by retaining complete control over the physical resources it depends on (e.g., CPU, memory). On x86 platforms for example, rHype uses *CPU protection rings* to ensure that VMs cannot execute privileged instructions and gain control over resources the hypervisor depends on. The hypervisor runs in "hypervisor" mode in CPU ring 0, the

Resource x Mediation / Evaluation	reference	TCB-hype (size)	TCB-MAC (size)	Flexibility (malleability)	Performance (throughput)	Resource Usage (quantity)
I/O-MMU in MAC VM		>>>	o	o	<	o
I/O-MMU in Hypervisor	x	o	o	o	o	o
VM Memory in MAC VM		o	>->>>	>	<	o
VM Memory in Hypervisor	x	o	o	o	o	o
vLAN in General VM		<	o	>	<	>>>
vLANs in MAC VM		<	>->>>	>	<	>
vLANs in Hypervisor	x	o	o	o	o	o
vBlocks in General VM	x	o	o	o	o	o
vBlocks in MAC VM		o	>->>>	o	o	<
vBlocks in Hypervisor		>>>	>->>>	o	o	<
I/O device in Same VM	x	o	o	o	o	o
I/O device in Other VM		o	o	o	<<<	<<<
I/O device in MAC VM		o	>>>	o	<<<	<<<
I/O device in Hypervisor		>>>	o	<<<	<<	<<<

Table 1: Impact of resource location. The implementation marked with 'x' serves as the reference. Legend: (<) smaller, (>) larger, (o) comparable to the reference.

highest privileged protection mode. VMs and the guest operating systems run in ring 2 and applications on top of the guest operating systems run in ring 3. From a CPU point of view, programs can access configurations in their own ring and rings with higher numbers. This way, the operating system inside a VM running in ring 2 is still protected against its applications running in ring 3. The privileged VM implementing VM management and hardware device drivers runs in ring 2 as well, however its I/O privilege level is set to 2 (as compared to 0 for normal non-I/0 VMs) and enables direct access to I/O device memory. The I/O VM is seen as part of the virtual machine monitor infrastructure. The hypervisor depends on its co-operation to manage the peripheral system hardware devices.

rHype isolates virtual resources against each other, such as virtual memory, CPU, and vLAN. For example, the rHype *memory* management ensures that VMs see only virtual addresses, which are mapped under the control of the hypervisor. The only way to share memory between VMs is through a shared virtual memory resource. The *CPU* represents a resource that has long been virtualized to enable interleaved execution of multiple programs on a single real CPU. The conventional context switch ensures that the CPU state is saved and replaced with the saved CPU state of the next VM that will be running on this CPU. Isolation is achieved since no explicit information can flow from the former VM context to the new VM context. Different *vLANs* are also isolated against each other and must be bridged inside VMs, which are subject to sHype access control when connecting to the vLANs.

Note that other resources, such as virtual disks, are implemented in the I/O VM which assumes the role of a MAC VM for resource.

4.2 Access Control Enforcement

Mediation is implemented by inserting *security hooks* into the code path inside the hypervisor where VMs access virtual resources. A *security hook* is a specialized access enforcement function that guards access to a virtual resource. In this case, it enforces information flow constraints between VMs according to the security policy. Each security hook adheres to the following general pattern:

1. gather access control information (determine VM labels, virtual resource labels, and access operation type)
2. determine access decision by calling the ACM
3. enforce access control decision

Using security hooks, sHype minimizes the interference with the core hypervisor while enforcing the security policy on access to virtual resources.

Figure 3 shows the hook that mediates the attachment (binding) of VM 2 to a virtual LAN vLAN A inside the hypervisor. First, the security hook looks up the VM security label range void pointer SSLRP in the VM data structure and retrieves the security label void pointer OSLP of vLAN A. Then it queries the ACM for an access control decision based on the label pointers and the *joinvLAN* operation. The ACM decides, whether a subject with the security range to which SSLRP points is allowed to perform the operation *joinvLAN* on the object with the security label to which OSLP points. If the hook receives the decision *permitted*, then the hook continues with normal operation and connects the VM to vLAN A. The subsequent sending and receiving of packets via the connected adapter will not be mediated explicitly. If the hook receives the decision *denied*, it will deny this VM access to vLAN A and indicates this to the VM.

To keep access control overhead on the performance-critical path near-zero most of the time, we use *bind-time authorization* and *explicit caching* of access control decisions.

Bind-time authorization restricts access control decisions to the time a VM binds to a virtual resource (see joining a virtual LAN in Figure 3). Subsequent access to this resource (e.g. sending or receiving packets) is implicitly covered by the access control at binding time. This method applies to virtual resources that require explicit binding before they can be used and which can be revoked if necessary. It works for most high-performance resources, such as vLAN, vSCSI, shared memory, and vTTY. With bind-time authorization, access control decisions occur on the non-critical performance path and the overhead on the critical path will be near-zero. Even bind-time authorization might require some security-related control on the performance-critical path, e.g., to enforce isolation between virtual LANs.

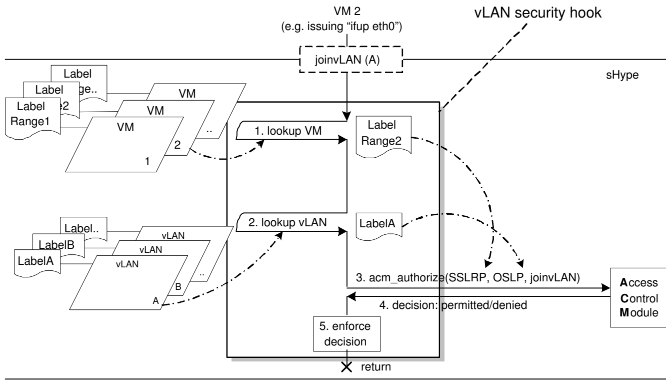

Figure 3: Security hook guarding a vLAN.

The binding must be revoked if the policy changes and the access control decision does not hold any more (see Section 4.5).

Explicit caching supports access control for virtual resources that do not follow the binding-before-use paradigm (e.g., signals or spontaneous inter-VM communication). In this case, we cache access control decisions locally in the `VM` structure (preserving cache locality) in a single bit per VM and resource: The bit being "1" means that access is allowed. The bit being "0" means that a new access control decision is necessary. If only permitted access is performance-critical and if access control decisions are rate-controlled (to counter DOS attacks of malicious VMs causing repeatedly time-consuming access control decisions that yield denied), the cache resolves the access control decision most of the time through local cache-lookup. If the system configuration enables prediction of usage patterns for non-binding virtual resources, cache pre-loading can move even initial access decisions out of the critical path. Explicit caching mechanisms pay for near-zero overhead on the critical path with additional management and complexity.

4.3 Access Control Module

The ACM stores the current policy, enables flexible policy management, makes policy decisions based on the current policy, and triggers call-back functions to re-evaluate access control decisions in the hypervisor when the policy changes.

The ACM stores all security policy information locally in the hypervisor and supports efficient policy management through a privileged H_Security hypervisor call. A privileged policy management VM uses this hypervisor call to manage the security policy stored in the ACM. Privileged VMs are explicitly assigned an access right for managing access to the ACM control data structure, which is verified by a security hook in the `H_Security` hypervisor call. This right may only be assigned to a trusted VM. Consequently, the access control for security management is integrated into the general mandatory access control framework.

For initial labeling of virtual resources, the ACM exports an `acm_init` call that determines the security label of a virtual resource based on its resource type and ID, and links the label to the virtual resource. Calls to `acm_init` are inserted where VMs or virtual resources are created and initialized in the hypervisor core. The ACM also exports the `acm_authorize` function, which takes a VM label, a virtual resource label, and an operation type as parameters and decides whether access is *permitted* or *denied* according to the security policy. The enforcement hooks call this function to enforce the policy on access of VMs to virtual resources. We describe re-evaluating access decisions in case of policy changes in Section 4.5.

4.4 Security Policy

To specify security policy, we attach security labels to virtual resource data structures (e.g., vLAN). To specify authorizations, we attach security labels to VMs structures. Those security labels store information needed to make access decisions inside the hypervisor. sHype assigns security labels through a call to `acm_init` when the respective virtual resource or VM data structures are initialized inside the hypervisor (see Section 4.3). It retrieves type-specific security labels from the current security policy and stores pointers to them in a pointer variable added to the initialized data structure (e.g. VM, vLAN). Following the implementation of the Linux Security Modules [18], we use `void` pointers to attach label structures to virtual resources and VMs in sHype. This way, the hypervisor-core remains independent of the policy representation. This promotes modularity of the code for maintenance and assurance reasons.

To allow access of VMs to virtual resources, the VM's label must reflect the required authorization to access the resource. We say the VM's security label must "dominate" the resource's security label with regard to the access type. The interpretation of security labels and the implementation of the "dominates" predicate are specific to the security policy. We use the Caernarvon [17] security policy. Caernarvon is a static security policy that does not re-label resources during normal operation. Because of the static resource labels, access control decisions change only if the underlying security policy itself changes. The benefit is that we can move access control decisions out of the critical path into the binding phase of virtual resources (e.g., mounting a virtual disk or connecting to a virtual LAN). This access decision holds during subsequent use of the resource until the policy is explicitly changed.

VMs serving multiple coalitions must be assigned multiple labels in order to receive such requests. The VMs ability to enforce information flow control constraints inside the VM determines whether it should be given MAC VM status. Once the hypervisor allows a VM to control multiple security labels, the hypervisor can no longer independently enforce the boundaries but must rely on the co-operation of the MAC VM to prevent leakage of information to VMs of these labels.

4.5 Change Management

When the policy changes, we must explicitly revoke a shared resource from a VM that is no longer authorized to use it. Since we use extensive caching, we must propagate access authorization changes into the caches near the enforcement hooks. For this purpose, each enforcement hook defines a re-evaluation callback function. When invoked by the ACM, the re-evaluation function (i) re-evaluates the original access control decision and (ii) revokes shared resources in case the authorization is no longer given.

Figure 4 shows the hooks for evaluating and re-evaluating access control decisions for a VM X joining vLAN Y (bind-

time authorization). Binding an adapter of a VM to a virtual LAN initially triggers an access control decision. This decision is assumed valid for subsequent send and receive operations. Once the policy changes, the ACM calls the re-evaluation callback at the enforcement hook inside the vLAN implementation to validate this initial access control decision. If access is still permitted under the changed policy, no further action is taken and continuous access to the vLAN is granted. If access is not permitted under the changed policy, the original joinvLAN operation is reverted and the access to the vLAN is disabled for the VM. We revoke the binding of a virtual Ethernet adapter by disconnecting the link between the virtual Ethernet adapter and the vLAN structure. This way, sending further packets to a vLAN, the related hypervisor call will return an error code to the VM. Receiving data packets on this adapter is no longer possible because its virtual Ethernet MAC address is no longer registered with the vLAN. Packets in the sending and receiving queues can be removed. To the operating system inside the VM a revocation looks as if the network cable was unplugged.

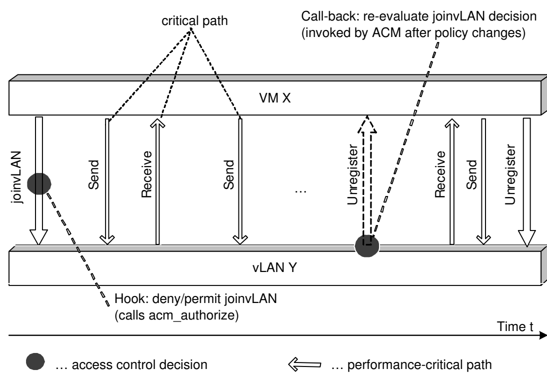

Figure 4: Re-evaluating a joinvLAN AC decision.

Where we use explicit caching, e.g. spontaneous IPC or events, policy changes result either in invalidating the access control cache entries by writing "0" into the cache entry (inducing re-evaluation at the next use) or in re-loading the access control entries (inducing re-evaluation immediately). In both cases, changing the policy involves re-evaluating affected cached access control decisions. We currently re-evaluate all policy decisions for all hooks when a policy change occurs, i.e., we don't interpret the policy changes to reduce the re-evaluation to those hooks that actually are affected. If policy changes occur more often, then the trade-off might change and affording more control code to restrict the necessary re-evaluations to affected VMs or resources might prove worthwhile.

5. EXPERIMENT

Figure 5 illustrates how labels, label ranges, and access control decisions are related. This example considers confidentiality requirements only. In the figure, two different virtual LAN domains *vLAN A* and *vLAN B* are defined. vLAN A has the object security label {`none`,`none`} and is used for legacy VMs that do not have confidentiality or integrity requirements. Such a label identifies resources that do not participate in the security model. Only VMs that do not participate in the controlled sharing are allowed to access such labeled resources (legacy support). vLAN B is controlled and has the object security label {`ibm_secret`, `none`} requiring at least `ibm_secret` level VMs to connect to it (no integrity requirements). Without loss of generality, this example does not consider the category component of the UAC. If the UAC includes a category component, any VM connecting to vLAN B would have to be cleared for vLAN B's category as well as the confidentiality level.

Figure 5 shows VM 1, VM 2, and VM 3 connecting to the virtual LANs A and B. VM 1 and VM 2 have the same security label range $\{\{$`ibm_secret`, `none`$\}_{from}$, $\{$`ibm_secret`, `none`$\}_{to}\}$. In contrast, VM 3 has the subject security label range $\{\{$`none`, `none`$\}_{from}$, $\{$`none`, `none`$\}_{to}\}$.

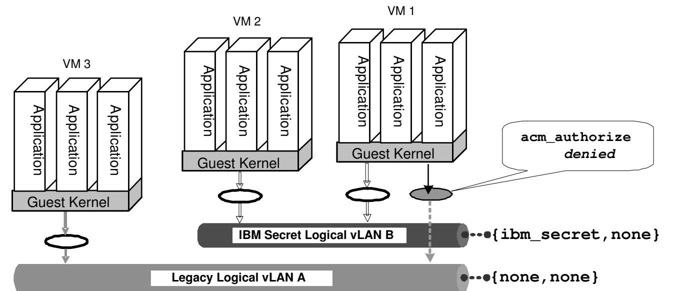

Figure 5: Multi-level secure vLAN based on sHype.

In the above configuration and security policy setting, sHype will allow VM 1 and VM 2 to connect to vLAN B because they are cleared for confidentiality level `ibm_secret` and there are no integrity requirements specified. None of them will be allowed to connect to vLAN A. Only VM 3 will be able to connect to vLAN A.

Results. To implement vLAN mandatory access control in rHype, we added one hook into the hypervisor call registering a VM's virtual Ethernet adapter to a virtual LAN. This hook calls `acm_authorize` as described in Figure 3. We inserted calls to `acm_init` into the initialization phase of virtual Ethernet adapters and VM data structures for initial labeling. Due to the bind-time authorization, there is no need for access control decisions when sending or receiving packets over the vLAN. Thus, we achieve zero overhead for access control on the critical path. There are only a few lines of sHype code in the architecture-dependent part of the rHype implementation (PowerPC versus x86), e.g., the definition of a new `H_Security` hypervisor call allowing policy VMs to manage the policy within the ACM and the insertion points for the `acm_init` call for labeling the architecture dependent VM data structure. We have successfully tested the revocation of vLAN access for VMs in case of policy changes. We revoke the binding of a virtual Ethernet adapter by disconnecting the link between the virtual Ethernet adapter of a VM from the vLAN structure. To the operating system inside the VM a revocation looks as if the network cable was unplugged.

6. RELATED WORK

Secure operating systems have long been the subject for challenging research. Only few of them found their way into

real use, such as GEMSOS [19, 20], KSOS [21], or Multics [12, 22]. The huge design, development, evaluation cost proved to be justified only for specialized application domains with very high security requirements.

One result was that virtualization of real hardware enabled the execution of multiple single-level virtual systems on a single hardware platform that aimed to ensure that those virtual systems where strongly isolated against each other. The prevalent approach to create multiple virtual machines on a single real hardware platform is the Virtual Machine Monitor (VMM) approach [23]. In VMMs, the principal subjects and objects are virtual machines and virtual disks, rather than conventional processes and files. That is the inherent difference between a VMM and a traditional operating system.

Based on VMs, a single system could implement a multilevel secure system by dividing it into multiple single-level virtual systems and securely separating them. Separation Kernels are virtual machine monitors that completely isolate different virtual machines. Rushby [1] proved that complete isolation and separation of VMs is possible. Based on Rushby's work Kelem et. al. [2] derived a formal model for separation Virtual Machine Monitors. One example of a more recent separation kernel design based on virtualization is NetTop [24]. NetTop implements different virtual systems that are isolated against each other on a single hardware platform to allow processing of data belonging to multiple sensitivity levels on a single system.

Recognizing that a strictly-separated VM approach does not map well into cooperating distributed applications, some research examined kernels that enabled secure sharing between VMs. However, existing secure sharing VMM approaches [5, 25] suffer from high performance overhead as well as large trusted computing bases due to necessary I/O emulation inside the hypervisor layer. Additionally, they are constructed to achieve highest assurance, requiring them to address covert channels on cost of complexity and performance.

Microkernel system architectures also struggled with the problem of determining how to control access to system resources. Some systems focus on minimality, forgoing all but most basic security. Others concentrate system-wide security features in the kernel. Notable examples include EROS [26], L4 [27], and Exokernel [28].

Today, there are a number of virtualization technologies that are deployed successfully in the commercial and research domain, such as VMWare [29], Terra [6], Xen [30], and rHype [8] that offer a basis for a broad application of sHype.

7. CONCLUSION

We presented a secure hypervisor architecture, sHype, which we are implementing into the rHype IBM research hypervisor. sHype provides boot and run-time guarantees currently lacking in most systems, addresses prevailing operating system security weaknesses by providing confinement opportunities, and enables secure communication and sharing between workloads on the same platform and potentially across multiple platform and organizational domains. Compared to operating system security controls, our resulting hypervisor-based security architecture offers stronger isolation of workloads and a security evolution path through sHype. We described the design and the implementation of the basic hypervisor security architecture and have successfully applied it to enable flexible policy-driven confinement of virtual LANs.

A secure hypervisor, such as sHype, enables its users to run a trusted operating system securely alongside a distrusted operating system on a single platform. These capabilities enable corporations to run text processing applications or do program development in function-rich operating environments and –securely isolated from it– to run sensitive applications processing confidential data in more secure and restricted operating environments. End users can start a trusted Web browser for Internet-banking (or for accessing classified data bases) in its own VM, which might prove a valuable step to on-demand security environments and enable the proliferation of user-friendly, functionrich, and less secure applications alongside highly sensitive applications.

Currently, we are extending the security architecture to cover multiple hardware platforms – involving policy agreements and the protection of information flows crossing the hardware platform boundary (i.e., leaving the control of the local hypervisor). We need to establish trust into the semantics and enforcement of the security policy governing the remote hypervisor system before allowing information flow to and from such a system. To this end, we are experimenting with establishing this trust through the Trusted Computing Group's Trusted Platform Module [31] and a related Integrity Measurement Architecture [32]. Future work includes the accurate accounting and control of resources (such as CPU time or network bandwidth) and generating audit trails appropriate for medium assurance Common Criteria evaluation targets.

8. REFERENCES

[1] John Rushby, "Proof of Separability—A verification technique for a class of security kernels," in *Proc. 5th International Symposium on Programming*, Turin, Italy, 1982, vol. 137 of *Lecture Notes in Computer Science*, pp. 352–367, Springer-Verlag.

[2] N. L. Kelem and R. J. Feiertag, "A Separation Model for Virtual Machine Monitors," in *Proc. IEEE Symposium on Security and Privacy*, 1991.

[3] Department of Defense, Ed., *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28STD, December 1985.

[4] Common Criteria, "Common Criteria for Information Technology Security Evaluation," http://www.commoncriteriaportal.org.

[5] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn, "A VMM Security Kernel for the VAX Architecture," in *Proc. IEEE Symposium on Security and Privacy*, May 1990.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proc. 9th ACM Symposium on Operating Systems Principles*, 2003, pp. 193–206.

[7] James P. Anderson et. al., "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, Vol. I+II, Air Force Systems Command, USAF, 1972.

[8] IBM Research, "The Research Hypervisor – A Multi-Platform, Multi-Purpose Research Hypervisor,"

http://www.research.ibm.com/hypervisor.

[9] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," *Proceedings of the* 12th *USENIX Security Symposium*, August 2003.

[10] U. Shankar, T. Jaeger, and R. Sailer, "Practical Integrity Verification for Existing Applications," *submitted for publication*, 2005.

[11] K. J. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep. MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[12] D. E. Bell and L. J. LaPadula, "Secure computer systems: Unified exposition and multics interpretation," Tech. Rep., MITRE MTR-2997, March 1976.

[13] M. Schaefer, B. Gold, R. Linde, and J. Scheid, "Program Confinement in KVM/370," *Proc. of the 1977 ACM Annual Conference*, pp. 404–410, October 1977.

[14] P. A. Karger and J. C. Wray, "Storage Channel in Disk Arm Optimization," *Proc. IEEE Symposium on Security and Privacy*, May 1991.

[15] W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *Proc. IEEE Symposium on Security and Privacy*, May 1991.

[16] Ray Spencer, Peter Loscocco, Stephen Smalley, Mike Hibler, David Anderson, and Joy Lepreau, "The Flask Security Architecture: System support for diverse security policies," in *Proceedings of The Eight USENIX Security Symposium*, August 1999.

[17] Helmut Scherzer, Ran Canetti, Paul A. Karger, Hugo Krawczyk, Tal Rabin, and David C. Toll, "Authenticating Mandatory Access Controls and Preserving Privacy for a High-Assurance Smart Card," in *(ESORICS)*, 2003, pp. 181–200.

[18] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Eleventh USENIX Security Symposium*, August 2002.

[19] W. R. Shockley, T. F. Tao, and M. F. Thompson, "An Overview of the GEMSOS Class A1 Technology and Application Experience," *11th National Computer Security Conference*, pp. 238–245, October 1988.

[20] R. R. Schell, T. F. Tao, and M. Heckman, "Designing the GEMSOS Security Kernel for Security and Performance," *8th National Computer Security Conference*, pp. 108–119, 1985.

[21] E. J. McCauley and P. J. Drongowski, "KSOS – The design of a secure operating system," in *Proc. In AFIPS Conference*, 1979, pp. 345–353.

[22] Paul A. Karger and Roger R. Schell, "Thirty Years Later: Lessons from the Multics Security Evaluation," in *Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[23] R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer Magazine*, vol. 7, no. 6, pp. 34–45, 1974.

[24] R. Meushaw and D. Simard, "NetTop - Commercial Technology in High Assurance Applications," *Tech Trend Notes*, Fall 2000.

[25] B. D. Gold, R. R. Linde, and P. F. Cudney, "KVM/370 in Retrospect," in *Proc. IEEE Symposium on Security and Privacy*, 1984.

[26] J. Shapiro, J. Smith, and D. Farber, "EROS: A fast capability system," *Proceedings of the* 17th *Symposium on Operating System Principles*, 1999.

[27] J. Liedtke, "On μ-kernel construction," *Proceedings of the* 15th *Symposium on Operating System Principles*, 1995.

[28] D. Engler, M. Kaashoek, and Jr. J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," *Proceedings of the* 15th *Symposium on Operating System Principles*, 1995.

[29] Vmware, "vmware," http://www.vmware.com/.

[30] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[31] "TCG TPM Specification Version 1.2," http://www.trustedcomputinggroup.org.

[32] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Thirteenth USENIX Security Symposium*, August 2004, pp. 223–238.