RC23629 (W0506-051) June 8, 2005
Computer Science

IBM Research Report

Building a MAC-based Security Architecture for the Xen Opensource Hypervisor

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

Building a MAC-based Security Architecture for the Xen Opensource Hypervisor

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

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

Abstract

We present the sHype hypervisor security architecture and examine in detail its mandatory access control architecture. While existing hypervisor security approaches aimed at high assurance have proven useful for high-security environments which prioritize security over performance and code-reuse, our approach aims at commercial security where near-zero performance overhead, non-intrusive implementation, and usability are most important. We provide the rationale behind the sHype concepts and describe its tailored implementation for the Xen open-source hypervisor.

We anticipate that the availability of better isolation through new hardware support in commodity systems together with the broad availability of virtualization software will increase the demand for Virtual Machine Monitor (VMM) systems running mutually distrusted coalitions of Virtual Machines (VM). Because the VMM systems can provide reliable isolation, some controlled sharing responsibilities of operating systems will be moved to the VMM. 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 prevention of covert timing channels), a multi-system approach using separate hardware is recommended.

1 Introduction

As general-purpose workstation- and server-class computer systems increase in available processing power and decrease in cost, it becomes cost-effective to aggregate the functionality of multiple standalone systems onto a single hardware platform. This minimizes costs for system management and maintenance and maximizes system utilization. Virtualization technology, which enables a single system hardware to support multiple operating systems, is quickly becoming a commodity. This technology creates multiple virtual machines (VM) out of one real machine and carefully multiplexes multiple virtual resources onto a single real resource. The broad availability and use of virtualization technology is driven by improved hardware support, such as fully virtualizable CPUs and IO-MMU [1, 2] controlling direct memory access to devices, which enables very efficient implementation of virtual machines. Suddenly, multiple operating systems can be efficiently co-located inside virtual machines on a single general-purpose hardware platform.

In addition to its availability, the potential impact of virtualization on workload consolidation and load balancing is getting the attention of key industry players. Microsoft recently announced that their next generation security architecture NGSCB [3] will be based on virtualized environments and Intel hopes to run home entertainment in virtualized environments, while large companies selling servers have very successfully used virtualization for server consolidation, service provisioning, and workload-balancing for decades.

Although co-locating operating systems and their workloads on the same hardware platform offers great opportunities, it also requires us to carefully consider possible undesirable interactions between those systems sharing resources. Therefore, VMM environments by default do not allow to share real resources directly. Real system resources are virtualized by the hypervisor layer (e.g., memory, CPU) and can be accessed by VMs exclusively through their virtualized counterpart (e.g., virtual memory, virtual CPU). This hypervisor layer is strongly protected against the operating systems running in VMs on top of it and enforces isolation of these virtual resources. Peripherals, such as disk or network adapters, are exclusively assigned to a single VM. If necessary, such a VM can in turn virtualize its real resources to share it with other VMs (e.g., virtual disk server, VLAN). We will carefully examine under which conditions such VMs are allowed to share peripherals with other VMs without violating the isolation properties between VMs. Consequently, virtual machines that do not share virtual resources are considered isolated from each other.

There are currently at least two challenging security problems when broadly deploying virtualization technology:

- The sharing of virtual resources among co-operating virtual machines is defined statically and resulting isolation properties of VMs are a side-effect of administration rather than of well-defined security management. However, today's environments depend more on sharing of resources and interconnection of workloads than ever before and this trend promises to increase. Consequently, there is need for an architecture that efficiently defines and enforces access control between related groups (coalitions) of virtual machines.

- The isolation of virtual resources, while sufficient for commercial environments, is insufficient for high-security environments where leaking even of very small amounts of data is unacceptable. Such leaks are introduced by covert channels, which are based on observing system behavior (timing of events or storage patterns) rather than by explicit data sharing.

The first problem concerns the (explicit) sharing of virtual resources between VMs. On one hand, the current framework for controlling sharing is extremely static, offering only limited VM-isolation guarantees. Such guarantees are often a side-effect of a particular system configuration instead of a consciously architected and designed policy that can be reasoned about. On the other hand, co-operating workloads running in different virtual machines offer a unique opportunity to implement access control in the generic virtualization layer very efficiently. By enforcing access control in the self-protecting virtualization infrastructure, related access controls are protected against misbehavior of operating systems and workloads. The coarse-granular resources and VMs enable simple security policies that control their interactions.

The second problem concerns covert channels. While controlling the explicit information flows between VMs is efficient, preventing implicit information flows comes at the cost of increased complexity, rewriting of hypervisor code, and decreased performance. These disadvantages of eliminating covert channels outweigh the interests of most customers. We believe, that the existing isolation of virtual resources is commercial-grade, meaning that controlling explicit data flows from one to another virtual machine and minimizing covert storage channels by careful resource management is sufficient in commercial environments. Our position is not to eliminate covert channels but (i) to minimize them through careful resource management, and (ii) to enable users through configuration options to mitigate remaining covert channels where necessary. To mitigate remaining covert channels, we introduce security rules guaranteeing that certain workloads never run on the same real platform; protection against covert channels between these workloads thus approximates the protection by air-gaps as they exist between non-virtualized environments.

The main focus in this paper is on the controlled sharing of resources, which is of broad interest in commercial environments. The sharing of virtual resources is currently not controlled by any formal policy. This makes it extremely difficult to measure the effectiveness of isolation between VMs and current approaches do not scale when considering the management of groups of systems and workload-balancing through VM migration.

We explore in this paper the design and implementation of sHype, a security architecture for virtualization environments, which leverages this virtualization layer to control the sharing of resources among VMs according to formal security policies. The major goals are (i) non-intrusiveness with regard to existing code, (ii) near-zero overhead on the performance-critical path, (iii) scalability regarding the management of many machines (simple policies) and the migration of VMs between them (machine independent policies).

We implemented the core hypervisor security architecture (sHype) into the Xen hypervisor [4] where it controls all inter-VM communication according to formal security policies. Our modifications to the Xen hypervisor are small and add about 2000 lines of code. The secure hypervisor architecture is designed to achieve medium assurance (Common Criteria EAL4 [5]) for hypervisor implementations. Our hypervisor security enhancement achieves near-zero overhead on the performance-critical path. While this paper describes sHype for the Xen hypervisor, the presented architecture proves flexible; it was originally implemented for the rHype research hypervisor [6] and is being implemented into the PHYP [7] hypervisor.

Section 2 introduces the typical structure of a Xen hypervisor environment for which we have developed a generic security architecture. Mutually suspicious workload-types serve as an example to illustrate requirements and the use of our hypervisor security architecture. After discussing related work in Section 3, we introduce the design of the sHype hypervisor security architecture in Section 4 and its implementation in Section 5. Section 6 evaluates our architecture and implementation.

2 Background

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 [8, 9], 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 meet 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 Direct Memory Access (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.

2.1 The Xen Hypervisor

As an example of VMMs, we use the Xen [4] open-source hypervisor throughout this paper. Figure 1 illustrates a basic configuration of the Xen opensource hypervisor. The Xen hypervisor consists of a small software layer on top of the real system hardware. It implements the virtual resources vMemory, vCPU, event channels, and shared memory on top of the system hardware and controls I/O and memory access to devices.

Virtual machines (also called domains in Xen) are built on top of the Xen hypervisor. A special VM, called DOM0 (domain zero) is initially created. It serves for the management of other VMs (create, destroy, migrate, save, restore) and controls the assignment of I/O devices to other VMs.

VMs started by DOM0 are called DOMU (user domains); they can run any paravirtualized [4] operating system, e.g., Linux. Guest operating systems running on Xen are minimally changed, for example by replacing privileged operations with calls to the hypervisor. Such privileged operations cannot be called directly by the guest OS 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 virtual resources (e.g., event channels, shared memory); (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.

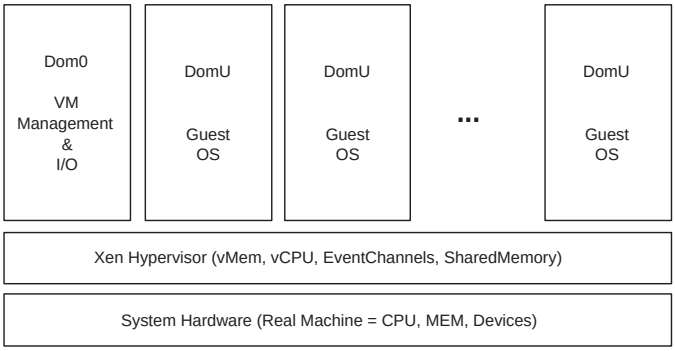

Figure 1: Xen Hypervisor Architecture.

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 controls these by itself and protects them against the user domains (DOMU VMs).

Exclusive VM resources include virtual memory, vCPU. Xen offers just two shared virtual resources on top of which all inter-VM communication and cooperation is implemented:

- Event channels: An event-channel hypervisor call enables a VM to setup an event channel to another VM (point-to-point). Event-channels are used for synchronization.
- Shared memory: A grant-table hypervisor call enables a VM to allow another VM access to virtual memory pages it owns. Synchronization of access to shared memory is implemented using event channels.

Shared virtual resources, such as network adapters, virtual block devices, vTTY and other devices inside VMs, are efficiently implemented as device drivers inside the Guest-OS and use event-channels and shared memory for synchronization and communication.

Physical system resources differ from virtualized resources in a couple of key ways: (1) I/O-MMUs are needed to restrict direct memory transfers to a VM's memory space. (2) Performance is best if the devices are co-located with the code using them in the same VM and consequently the optimal case is a physical resource per VM, which may not be practically feasible. (3) Driver code is too complex for inclusion in the hypervisor, so a device that is to be shared by multiple

coalitions is required to be implemented in a device domain (VM), which then makes this device available through inter-VM sharing to other VMs. In Xen, a SCSI disk or Ethernet device, for example, can be owned by a device server domain and accessed by other VMs through virtual disk or Ethernet front-end drivers, which communicate with the device domain using event channels and shared memory provided by the hypervisor.

2.2 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. 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 mandatory access control (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, browsing and collecting possible items to be purchased need not be as high integrity. At the same time, an OEM's code advertising a product that the company distributes may be run as another workload that should be isolated from the order workloads (web service, db, browsing).

In the customer order example, the coalition of VMs performing customer orders are protected from the other VMs on the system. We merge them into the *Order* coalition. The order VMs may communicate, share some memory, network, and disk resources. Thus, they are as a coalition confined 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 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 from being shared with other VMs outside the *Order* coalition.

2.3 Problem Statement

The problem we address in this paper is the design of a VMM *reference monitor* that enforces 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 [10]. 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.

We apply the reference monitor to control all references of shared virtual resources by VMs and to allow coalitions of workloads to communicate or share efficiently within a coalition while efficiently confining workloads of different coalitions.

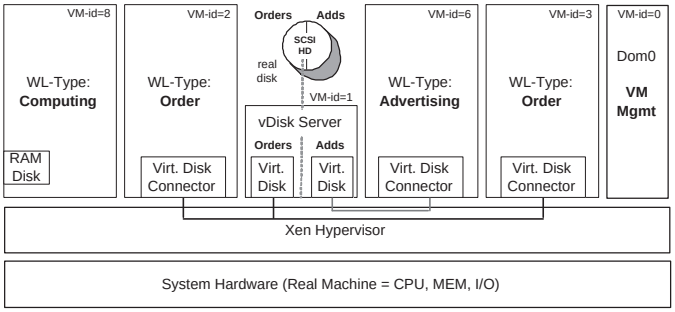

Figure 2: VM Coalitions and payloads in Xen.

Figure 2 shows an example of VM coalitions. Domain 0 has started 5 user domains (VMs), which are distinguished inside the hypervisor by their domain id (VM-id in Fig. 2). Domains 2 and 3 are running *order* workloads. Domain 6 is running an *advertising* workload, and domain 8 is running an unrelated generic *computing* workload (provisioned CPU time). Finally, domain 3 runs the virtual block device driver that offers two isolated virtual disks $vdisk_{order}$ and $vdisk_{adds}$ to the *Order* coalition and the *Advertising* domain. In this example, we want to enable most efficient communication and sharing among VMs of the *Order* coalition but contain communication of VMs inside this coalition. For example, no VM with *Order* workload is allowed to communicate or share information with any VM running *Computing* or *Advertising* workloads and vice versa. While the hypervisor controls the ability of the VMs to connect to the device domain, device domain is trusted to keep data of different virtual disks securely isolated inside its VM and (on the real SCSI disk) and to assign them correctly to the coalitions. This is a reasonable requirement since device domains are not application specific and can run minimized run-time environments (e.g., micro-kernel).

2.4 Solution Outline

We address this problem by implementing a reference monitor into the hypervisor layer. It mediates references of VMs to those resources that enable inter-VM communication (information flows). Once we know which resources require control, we must consider the mediation points for controlling these resources. We will see that there are three ways in which resources can be controlled: (i) in the Xen hypervisor; (ii) within a trusted domain (MAC-VM); or (iii) in a trusted domain upon the communication with an untrusted, general-

purpose VM or with other (hypervisor) systems. Finally, we define simple policies tailored to the hypervisor environment and based on workload types and resources. We will show an example where these policies and their enforcement play together to confine VM coalitions as illustrated in Figure 2.

3 Related Work

While there have been instances of highly secure operating systems that have been successfully commercialized – e.g., GEMSOS [11, 12], KSOS [13], or Multics [14, 15] – widespread or ubiquitous use of secure operating systems has to-date proven unsuccessful. The huge design, development, evaluation cost proved to be justified only for specialized application domains with very high security requirements. Access control with process and file granularity in general purpose OS while possible is very complex as illustrated by SELinux [16] policies. Expressing and enforcing a simple TCB model in general purpose OS can be very difficult due to interdependencies between processes [17]. VMMs can supplement OS security and provide confinement in case of OS security controls fail [18].

This previous work demonstrated that virtualization of real hardware enabled the execution of multiple single-level virtual systems on a single hardware platform ensuring that those virtual systems where strongly isolated from each other. The prevalent approach to create multiple virtual machines on a single real hardware platform is the VMM approach [19]. In VMMs, the principal subjects and objects are virtual machines and virtual peripherals (e.g., disks), rather than conventional processes and files.

Based on VMs, a single system could implement a multi-level secure system by dividing it into multiple single-level virtual systems, guaranteeing secure separation. Separation Kernels are virtual machine monitors that completely isolate virtual machines. Rushby [8] proved that complete isolation and separation of VMs is possible. Based on Rushby's work, Kelem et al. [9] derived a formal model for Separation Virtual Machine Monitors. One example of a more recent separation kernel design based on virtualization is NetTop [20]. NetTop implements virtual systems that are isolated from 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, these secure sharing VMM approaches [21, 22] tend to 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 the highest levels of assurance, requiring them to address covert channels at the cost of increased complexity and decreased performance. Karger et al. [23] report for the KVM approach a performance range of 10-50% (50-90% overhead) and the effort of rewriting 50% of the VMM code; the VaxVMM performs at 30-90% (10-70% overhead) and the effort of rewriting the entire VMM code base. Today, there are a number of virtualization technologies that are deployed successfully in the commercial domain, such as PHYP [7] and VMWare [24], and several promising research implementations, such as Terra [25], Xen [4], and the IBM Research Hypervisor [6]. All of these offer a basis for a broad application of sHype, while none were built for highest levels of assurance, nor do any use either of the KVM or VaxVMM approach.

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

In summary, the sHype approach –targeting the commercial hypevisors space– is supplementary to existing operating system security approaches and orthogonal to existing hypervisor security approaches.

4 sHype Design

Figure 3 illustrates the overall sHype security architecture and its integration into the XEN VMM system.

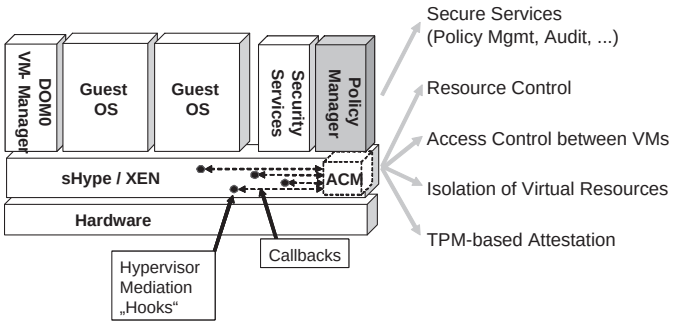

Figure 3: sHype – Hypervisor Security Architecture.

sHype is designed to support a set of security requirements: secure services, resource control, access control between VMs, isolation of virtual resources, and TPM-based attestation. sHype supports interaction with secure services in custom-designed minimized and carefully engineered VMs. An example is the policy management VM, which we use to establish and manage the security policies for the Xen hypervisor. Resource accounting provides control of resource usage. This enables enforcement of service level agreements as well as addressing denial of service attacks on hypervisor or VM resources. The mandatory access control enforces a formal security policy on information flow between VMs.

sHype leverages existing isolation between virtual resources and extends it with MAC features. TPM-based attestation [29] provides the ability to generate and report run-time integrity measurements on the hypervisor and VMs. This enables remote systems to infer the integrity property of the running system.

This paper focuses on the sHype mandatory access control architecture, consisting of: the *policy manager* maintaining the policy; the *mediation hooks* controlling access of VMs to shared virtual resources based on decisions delivered by callbacks; and the *access control module* (ACM) delivering access control decisions according to the security policy through these callbacks to mediation hooks upon request.

4.1 Goals and Decisions

The implementation of sHype is governed by two design goals and three major decisions. The first sHype design goal is to permit the hypervisor to retain a stable, near-minimal code base, allowing significant security assurances (e.g., Common Criteria) to be achieved and ensuring strictly non-intrusive code changes in the base VMM system. The second sHype design goal is to incur near-zero security-related performance overhead on the critical path.

Three major decisions shape the design of sHype:

(1) By *building on existing isolation properties of virtual resources*, sHype inherits the medium assurance of the existing hypervisor isolation but ensures minimal code changes in the virtualization layer (hypervisor).

(2) By using *bind-time authorization* and by controlling the access to spontaneously shared resources only at first time access (caching access control decisions) and when the policy changes, sHype ensures very low performance overhead on the critical path.

(3) By *enforcing formal security policies*, sHype enables reasoning about the effectiveness of specific policies, provides the basis for effective defense against denial of service attacks (through resource policy enforcement), and enables QoS or SLA-style security guarantees (through TPM-based attestation of system properties).

4.2 Access Control Architecture

In this section, we discuss the mandatory access control of information flows between VMs based on the reference monitor concept. To support various business requirements, sHype supports various kinds of MAC policies: Biba [30], Bell-LaPadula [14], Caernarvon [31], Type Enforcement [32], as well as Chinese Wall [33] policies.

The key component of the access control architecture is the reference monitor, which in sHype isolates virtual machines by default and allows sharing of resources among virtual machines when desired according to a mandatory access control (MAC) policy. The classical definition of a reference monitor [10] 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 to enable validation of its correct implementation.

We examine the first requirement in more detail. The second and third requirement are covered by generic hypervisor requirements: it is protected against the VMs and consists of a thin and minimal software layer. Accordingly, we will discuss requirements two and three only where the hypervisor delegates some enforcement to dedicated VMs (MAC-Domains).

Mediating security-critical operations. 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 resources whose use must be controlled in order to implement MAC policies. We also identify the location of the mediation points for these resources. The combination of resources to be controlled and their mediation points forms the reference monitor interface. We discuss only virtual resources, because real resources can only be used exclusively by one VM or shared in form of virtual resources. The following resources must be controlled in a typical Xen VMM environment:

- sharing of virtual resources (event channels, shared memory, and domain operations) between VMs access controlled and isolated inside the Xen hypervisor
- sharing of local virtual resources among local VMs (e.g. local vlans and virtual disks , see device domain in Fig. 2) access controlled and isolated within MAC-domains; and
- sharing of distributed virtual resources, e.g., VLANs spanning multiple hypervisor systems, access controlled and isolated in MAC-bridging domains of multiple systems.

The hypervisor reference monitor enforces direct access control and isolation on virtual resources in the Xen hypervisor. While sHype enforces mandatory access control on MAC-domains regarding their participation in multiple coalitions, it relies on MAC-domains to isolate the different virtual resources from each other and allow access to virtual resources only to domains that belong to the same coalition as the virtual resource. A good example of a MAC-domain is the device domain in Fig. 2, which participates in both the `Order` and the `Advertising` coalition. MAC-domains become part of the trusted computing base and should therefore be of minimal size (e.g., secure micro-kernel design). Since MAC-domains are generic, the cost of making them secure

will amortize fast by using them in many application environments. We sketch the implementation of MAC domains in Section 5.4.

If coalitions are distributed over multiple systems, we need MAC-bridging domains to control their interaction. The virtual resource that enables VMs on multiple systems to cooperate is typically a VLAN. Mac-bridging domains build bridges between their hypervisor systems over untrusted terrain to connect VLANs on multiple systems. To do so, they first establish trust into required security properties of the peer MAC bridging domains and their underlying virtualization infrastructure (e.g., using TPM-based attestation). Afterwards, they build secure tunnels between each other, and can from now on be considered as forming a single (distributed) MAC-domain belonging to multiple systems. The requirements on the resulting distributed MAC-domain are akin the requirements described above for the local MAC-domain. MAC-bridging domains become part of the reference monitor TCB similarly to MAC-domains. Noteworthy, multi-system VLANs and underlying MAC-bridging domains are a pre-requisite for securely migrating VMs between hypervisor systems and for implementing secure access to network file systems.

5 Implementation

First, we will define simple policies tailored to the Xen hypervisor environment based on workload types and resources that must be controlled. Then we will describe the management of the policies and the labeling of VMs and resources. Finally, we introduce the access control enforcement in the hypervisor, which guards access of VMs to resources based on the policies.

5.1 Security Policies

We implemented two formal security policies for Xen: (i) a Chinese Wall policy, (ii) a simple type enforcement policy. Both policies work on their own set of types (ChWall- or TE-types), which are assigned to VMs as a function of the workloads they can run. The ChWall- and TE-types form the granularity upon which VMs and resources can be distinguished. The assignment of types to VMs is an administrative task (i.e., part of the policy management).

Chinese Wall policy: The first policy enables administrators to ensure that certain VMs (and their supported workload types) cannot run on the same hypervisor system at the same time. This is useful to mitigate covert channels or to meet other requirements regarding certain workload types (e.g., workload types of competitors) that shall not run on the same physical system at the same time. This policy ensures an "air-gap" between such workloads and approximates the situation given without virtualization and related co-location of workloads.

The Chinese Wall policy defines a set Chinese wall types (ChWall-types), and these are assigned to a VM in function of the workloads it can run. It also defines conflict sets using these ChWall-types and ensures that VMs that are assigned ChWall-types of the same conflict set never run at the same time on the same system.

$$ChWall\text{-}types = \{IBM, Hertz, Avis, ...\}$$

$$ChWall\text{-}conflictset = \{Hertz, Avis\}$$

The hypervisor keeps a set of ChWall-types of all running VMs and allows a new VM only to be started or resumed or migrated-in, if the new VM's ChWall-types do not appear together with any running ChWall-type in any conflict set. If we assign the ChWall-type Hertz to VMs that can run Hertz workload and Avis to those VMs that can run Avis workloads, then the above ChWall conflictset will ensure that those to VMs never run at the same time on the same hardware.

Type Enforcement policy: The second policy specifies which running VMs can share resources and which cannot. It implements the coalitions introduced in Section 2.2 by mapping coalition membership one-to-one onto TE-types, e.g.,

$$TE\text{-}types = \{IBMOrder, HertzAds, IBMAds, AvisCo, ...\}$$

The TE policy consequently (a) defines the set of TE-types (coalitions), and (b) assigns TE-types to VMs (coalition membership). The TE policy rules enforce that VMs only share virtual resources if they have a TE-type in common, i.e., they are member of at least one common coalition.

Security information assigned to VMs and resources (ChWall- and TE-types) are maintained inside the policy management domain and inside the hypervisor. It is always protected against VMs.

5.2 Policy Management

The policy management is responsible for offering means to create and maintain (store, change, validate) policy instantiations for the Chinese Wall and Type Enforcement policies. To minimize code complexity inside the hypervisor, the policy management translates the easy-to-manage XML based policy representation into a binary policy representation that is both system independent and easy to use by the hypervisor layer.

The binary policy created by the Policy Management includes the assignment of VMs to ChWall-types and TE-types as well as the Chinese Wall conflict sets to be enforced on the ChWall-types. No other information is needed by the hypervisor to enforce the policies. The access class of a VM as sHype sees it is exactly a set of ChWall-types and TE-types. Access classes of virtual resources such as virtual disks comprise one TE-type only.

The policy management can either run in a minimized policy management domain on the managed system (current Xen approach) or it can run on a separate special purpose system, such as the Hardware Management Console (HMC) as used

by PHYP and other commercial virtualization solutions. The policy management is needed to change or validate a policy; it is not necessary to run the system and enforce the instantiated policies.

Since the policy management is affecting the access control decisions by determining the rules and access classes, it must be at least as secure (trusted) as the hypervisor enforcement itself. While envisioning a user-friendly graphical user interface to create the XML specification of the policy, we validate the XML policy independently of the GUI against user requirements before translating it into a binary policy. This way, the complex GUI can run on any convenient platform (OS) and stay out of the TCB, while the policy manager domain managing, translating, and enacting a signed-off XML policy will be part of the trusted computing base and must be kept simple and protected (reference monitor requirements).

5.3 Policy Enforcement

Mandatory access control is implemented as a reference monitor. The mediation of references of VMs to shared virtual resources is implemented by inserting *security enforcement hooks* into the code path inside the hypervisor where VMs share virtual resources. Hooks call back into the access control module (ACM) for decisions and enforce them locally at the hook. Isolation of individual virtual resources is inherited from the Xen hypervisor since it is a general design issue for virtualizing hardware rather than a security specific requirement.

5.3.1 Reference Monitor

sHype strictly separates access control enforcement from the access control policy according to the Flask [34] architecture. We describe the control architecture in the context of the hypervisor, but it will also be used in the MAC domains.

Figure 4 shows the sHype access control architecture as part of the core hypervisor and depicts the relationships between its three major design components. *Security enforcement hooks* are carefully inserted into the core hypervisor and cover references of VMs to virtual resources. Enforcement hooks retrieve access control decisions from the *access control module* (ACM).

The ACM decides access of VMs to resources based on the policy rules and the security labels attached to VMs (ChWall-types, TE-types) and resources (TE-types). The *formal security policy* defines these access rules as well as the structure and interpretation of security labels for VMs and resources. Finally, a hypervisor interface enables trusted policy management VMs to efficiently manage the ACM security policy.

5.3.2 Access Control Hooks

A *security enforcement hook* is a specialized access enforcement function that guards access of VMs to a virtual resource.

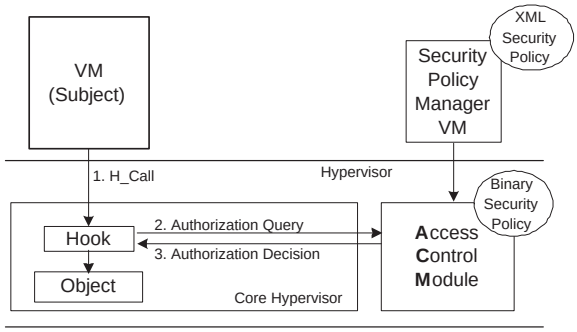

Figure 4: Hypervisor-based security reference monitor.

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. We have placed security enforcement hooks at the following places inside the hypervisor in order to enforce the Chinese Wall and Type Enforcement policies:

- *Domain management operations* (create, destroy, save, restore, migrate) are mediated by a `dom_op` security hook, which is functionally transparent if the access is allowed and which returns with an error code from the domain operation hypervisor call if the operation is denied by the ACM.
- *Event channel operations* (setup, destroy) are mediated by an `event_op` security hook, which is functionally transparent if the access is allowed and which returns with an error code otherwise.
- *Shared memory operations* (setup, grant access, remove access) are mediated by a `shmem_op` security hook operating transparently if access is denied and returning an error code otherwise.

Domain operation hook: This hook calls back into the ACM reporting the security reference of the domain originating the operation and of the domain that is being created or destroyed etc. Callbacks from these hooks are used by the ACM (1) to assign security labels to created domains and to free labels of destroyed domains, (2) to check Chinese Wall

conflict sets before creating, resuming, or migrating-in domains and (3) to adjust the set of running ChWall-types when destroying, suspending, or migrating-out domains. Caching access decisions for these hooks does not make sense since ACM uses these callbacks to update security state. This is not a problem because domain operations are typically off the performance critical path.

Event channel hook: Event channels hooks (create, destroy) mediate the setup and destruction of event channels between domains. The ACM uses callbacks from these hooks to decide whether the two domains setting up an event channel are actually member of a common coalition. Figure 5 shows the hook that mediates the setup of an event channel (event_channel hypervisor call) between the VM issuing the request and the VM with VM-id=`id`. First, the event hook looks up the security references for the local VM and for VM `id` and calls back the ACM via the `event_op` ACM callback. If the ACM returns permitted, the event channel setup continues beyond the hook. The subsequent sending and receiving of events via the connected channel do not need to be mediated because they would yield the same result (unless the policy changes, see below). If the hook receives the decision denied, the event channel setup is aborted and the hypervisor call returns with an error.

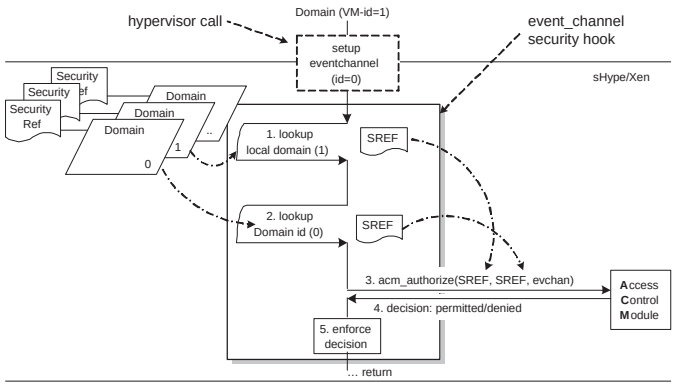

Figure 5: Security hook guarding the setup of event channels.

Shared memory hook: Grant-table hypervisor calls allow one VM to grant access to some if its own memory pages to another VM. This mechanism (synchronized via event channels) enables efficient communication between VMs running on the same hypervisor. Since the shared memory may in some cases be established dynamically during the communication (e.g., sending and receiving network packets or reading and writing from virtual disks), this operation together with the security hook guarding may be on the performance critical path. The hook itself is very similar to the event channel hook illustrated in Figure 5.

Decision caching. Since neither the event channel hook nor the shared memory hook callbacks induce any state change in the ACM, we use *explicit caching* of access control decisions to minimize the overhead introduced by the security hooks calling back into the ACM and the ACM deciding about access.

We cache access control decisions locally in the `VM` structures of the domains involved in the grant-table or event channel operation at the first time an access control decision is required between two VMs. Since access decisions neither depend on the direction of an access nor on the kind of access, we need only a single cache for each VM, which covers both event channel and shared memory access decisions. For example, the event channel setup access control decision retrieved in Figure 5 is cached as general access control decision between domains VM-id1 and VM-id2 and used by subsequent event channels or shared memory setup hooks between these domains. The cache is established symmetrically in both VMs. This is redundant caching, which can be optimized by keeping the cached decision only in one of the VMs (e.g., the one with the lower id) and looking it up there when needed. The decision cache is not used for domain operation hooks for two reasons: (i) the ACM must be involved to maintain its security state for such events, (ii) the type of operation matters.

We are experimenting with multiple cache layouts to find the best trade-off between memory requirements and lookup speed (direct indexing versus hash table lookup, i.e., performance versus memory and CPU cache locality). The direct indexing cache will use the VM-id as index and a single bit per decision. The bit being "1" means that access of the local VM to the VM id (index) is permitted. The bit being "0" means that a new access control decision is necessary. If only permitted access is performance-critical, then the cache resolves the access control decision most of the time through local cache-lookup. Since the possible ids are large numbers, keeping an indexed cache for all possible ids will lead to poor memory usage. Hash table based caching can help here by reserving a field for the peer VM-id and then directly storing the access control decision or indirectly storing the index into a bit-field (indirect indexed caching).

Explicit decision caching achieves near-zero overhead on the critical path at the cost of additional management and complexity. When a VM is destroyed or migrated-out and its ID could be re-used, then the access decision cache-entries regarding this VM-id must be cleared in all caches of running VMs. The induced overhead to clear these caches is very low.

Since our policies treat shared memory and event channels the same way when deciding access, once the cache is filled, it can be used for both types of virtual resources. Since all communication between domains, e.g., access to virtual disks in MAC-domains, is mapped onto the event channels and shared memory mechanisms, they will all benefit from the same decision cache.

Policy Changes. 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 of VMs. Additionally, we define a re-evaluation function for both event-channel and grant-table hooks because these hooks check permissions only once: when an event-channel or a shared memory area is setup, and not when it is used. When invoked by the ACM, the re-evaluation function (1) re-evaluates the original access control decision, and (2) revokes shared resources in case the authorization is no longer given.

Revocation of event-channels from inside Xen is straightforward. VMs trying to use revoked event-channels will receive error codes which must be handled regardless of access control. Memory shared between VMs will typically not be directly handed over by the guest OS to applications but rather used exclusively inside device drivers (e.g., virtual disk or network front-end and back-end drivers). Consequently, device drivers might run into a memory access fault when trying to send a request via shared memory to which their access was revoked. We are currently working on a call-back mechanism, initiated by the hypervisor, so that revoked shared memory can be reported to affected VMs and handled there in an more controlled fashion, allowing for more graceful failure.

5.3.3 Access Control Module (ACM)

The ACM maintains policy state, makes policy decisions based on the current policy, interacts with the policy manager VM to establish a security 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 hypervisor call interface. This interface is access-controlled by a specialized hook and will only be accessible by policy-management-privileged domains.

During domain operations, the ACM is called by security hooks and allocates and de-allocates security labels for created and destroyed domains according to the policy. These labels are later used for access control decisions. The virtual machine configuration includes references for the ACM that are used to determine the label for a newly created domain. In our example, such a label consists of a set of TE-types (specifying to which coalitions the domain belongs) and a set of ChWall-types as described in Section 5.1.

The ACM maintains the policy state needed to enforce the *Chinese Wall policy*. For this purpose, the ACM maintains a *Running ChWall Types* array indexed by the ChWall-type and containing a reference count that describes the number of running VMs that have assigned this ChWall-type. Whenever a domain is started, the ACM determines those conflict sets with which this VM shares a ChWall-type. Then it verifies if any of the other ChWall-types of these conflict sets is running. If any of these ChWall-types' reference count is non-zero, then we have a Chinese Wall conflict and the current domain is not permitted to start. Otherwise, the current domain is permitted to start and the *Running ChWall Types*' reference counts are incremented for those ChWall types that are assigned to the started VM. If a domain is destroyed, the *Running ChWall Types*' reference counts of this VM's ChWall-types are decremented.

Access control decisions for the *Type Enforcement policy* are simple. The ACM looks up the Coalition set of those domains that are trying to establish an event channel or shared memory. If both domains share a common TE-type (coalition membership), then the access is permitted. Otherwise it is denied. It can be implemented as an n-bit AND-operation over the TE-type vectors of the VMs where n is the number of known TE-types (coalitions).

5.4 MAC-domains

MAC-domains enable multiple coalitions to share a real resource by creating isolated virtual resources based on the real resource (recall the `vdisk` device domain in Figure 2). If sufficient hardware resources are available, MAC-domains are not necessary because hardware can be exclusively assigned to a single coalition. We sketch briefly how we envision MAC-domains to work. They must offer following guarantees in order to conform to reference monitor requirements:

1. Isolate the virtual resources (e.g., the two virtual disks for the *Order* and the *Advertising* coalition) inside the MAC-domain at least as well as the hypervisor isolates its virtual resources (event-channels, shared memory, virtual memory).

2. Control access of VMs to those resources according to the Type Enforcement Policy (only allow VMs that are members of the coalition to which the virtual resource is assigned to access it).

The isolation property can be achieved using MAC, e.g., based on SELinux, inside the domain while minimizing the run-time environment.

The access control property requires a MAC-domain to discover necessary coalition membership information (TE-types) of the requesting domain. For this reason, sHype offers to MAC-domains a hypervisor call that returns the coalition membership information of a connected domain using the protected policy information of the ACM. The hypervisor will return those coalitions (TE-types) of which both the MAC-domain and the requesting VM are members. Based on the returned membership information, the MAC-domain permits access of the requesting VM only to virtual resources that share membership in the same coalition(s). In the example in Fig. 2, the MAC-domain (VM 1) will permit VM 2

access to the *Orders* virtual disk and the domain with VM 6 access to the *Ads* virtual disk. It can retrieve the respective VM's membership information with the appropriate hypervisor call since the MAC-domain itself is member of both of these coalitions. The MAC-domain will get no memberships for the domain with VM 6, as it does not share any coalition membership with this domain. Consequently, VM 6 will not get access to any virtual disk on VM 1.

6 Evaluation

6.1 sHype-Covered Resources

Figure 6 shows the virtualized resources sorted according to where they are implemented. The `TCB coverage` column shows how well their isolation and mandatory access control is covered by the sHype reference monitor. We distinguish whether the implementing entity is serving a `single` coalition or `multiple` coalitions since the latter requires MAC control. If event channels, shared memory, virtual disks, virtual TTY, or VLANs are shared within a single coalition, sHype fully covers the TCB for sharing between coalitions. While the sHype architecture is comprehensive and its policy enforcement completely covers the communication between domains, sHype relies on domains that create virtual resources and offer them to multiple coalitions to correctly isolate virtual devices from each other (see Section 5.4). Such multi-coalition MAC-Domains are necessary if real peripherals must be shared between multiple coalitions or if different coalitions shall be able to co-operate using filtering and fine-granular access control implemented inside a MAC-Domain.

resource / implementation	event channel	shared memory	virtual disk	virtual TTY	virtual LAN	TCB coverage single / multi
Hypervisor	X	X				● / ◐
local VM			X	X	X	● / ◐
VMs on multiple systems					X	◔ / ◔

● ..fully covered by sHype ◐ ..partly covered by sHype

Figure 6: Coverage of Xen resources by the current sHype implementation.

If virtual resources (e.g. VLANs) are distributed over multiple hypervisor systems and communicate over a network, sHype relies on the domains bridging those systems (bridging domains) to securely isolate the VLAN traffic from other traffic on the connecting network and to control access of VMs on the connected systems to the VLAN (MAC). In consequence, sHype controls which domains are able to connect to MAC-bridging domains but defers isolation and MAC guarantees for VLAN traffic into these MAC-bridging domains.

6.2 Code Impact

The sHype access control architecture for Xen comprises 2600 lines of code. We insert three MAC security hooks into Xen hypervisor files to control domain operations, event channel setup, and shared memory setup. Two out of three hooks are off the performance critical path. One hook (shared memory setup) can be on or off the critical path depending on how shared memory is used by a domain. We implemented a generic interface (akin to the Linux Security modules interface but much simpler) upon which various policies can be implemented. We have implemented the Chinese Wall and the Type Enforcement policies for Xen as well as the caching of event channel and grant table access decisions. Maintaining sHype within the developing Xen hypervisor code base has proven very easy.

6.3 Performance

By design (authorization at binding-time not per-use) and by extensive decision caching, we minimize the overhead sHype introduces on the performance-critical path. Policy changes are an exception and related overhead, being in lower bounds, is not critical. **NOTE to the reviewer:** We will include a table showing performance overhead benchmark results for event channel and shared memory security hooks.

7 Conclusion

We presented a secure hypervisor architecture, sHype, which we successfully implemented into the Xen opensource hypervisor. We showed how access control in the hypervisor can be implemented in a way that incurs very low impact on VM performance and is least intrusive to existing and maintained VMM code. The hypervisor virtualization layer is becoming a standard component in the system software. With its coarse-grained resource management capability, protection against its workloads, and its relatively small footprint, it proved to be the ideal vehicle in which to implement a flexible security framework that is capable of supporting a range of security policies that can be tailored to the particular workload profile.

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 based on the Trusted Computing

Group's Trusted Platform Module [35] and a related Integrity Measurement Architecture [29]. While Xen de-aggregates drivers and management functions from DOM0 into their own domains, we are experimenting with MAC-domains that will become essential for sharing limited physical resources (e.g., in the mid-range server and desktop space). 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.

References

[1] Intel, "Intel Virtualization Technology Specification for the IA-32 Intel Architecture," April 2005, ftp://download.intel.com/ technology/ computing/ vptech/C97063-002.pdf.

[2] Advanced Micro Devices, "AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual, Rev 3.01," May 2005, http://www.amd.com/us-en/assets/ content_type/white_papers_and_tech_docs/33047.pdf.

[3] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *IEEE Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[4] 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.

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

[6] IBM Research, "The Research Hypervisor – A Multi-Platform, Multi-Purpose Research Hypervisor," http://www.research.ibm.com/ hypervisor.

[7] IBM, "PHYP: Converged POWER Hypervisor Firmware for pSeries and iSeries.," http://www-1.ibm.com/servers/enable/site/peducation/ abstracts/ abs_2bb2.html.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] National Security Agency, "Security-Enhanced Linux (SELinux)," http://www.nsa.gov/selinux.

[17] T. Jaeger, R. Sailer, and X. Zhang, "Analyzing Integrity Protection in the SELinux Example Policy," in *12th USENIX Security Symposium*. USENIX, 2003, pp. 59–74.

[18] S. E. Madnick and J. J. Donovan, "Application and analysis of the virtual machine approach to information system security and isolation," *Proceedings of the ACM workshop on virtual computer systems*, pp. 210–224, 1973.

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

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

[21] 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.

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

[23] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn, "A Retrospective on the VAX VMM Security Kernel," in *IEEE Transaction on Software Engineering*, November 1991.

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

[25] 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.

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

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

[29] 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.

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

[31] 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.

[32] W. E. Boebert and R. Y. Kain, "A practical alternative to hierarchical integrity policies," *8th National Computer Security Conference*, 1985.

[33] D. F. C. Brewer and M. J. Nash, "The Chinese Wall Security Policy," *Proc. IEEE Symposium on Security and Privacy*, pp. 206–214, May 1989.

[34] 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 Eighth USENIX Security Symposium*, August 1999.

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

Appendix

This appendix briefly discusses the TE policy for the simple example illustrated in Figure 2. Figure 7 shows the XML policy file for the Type Enforcement policy confining domains to coalitions, each coalition being defined with a TE-type. The XML policy file for the Chinese Wall policy looks similar but only assigns ChWall-types to VMs, not to resources. The pol-

```
<?xml version="1.0" ?>
<SecurityPolicySpec xmlns="http://www.ibm.com"...">
<Policy>
    <PolicyHeader>
        <Name>Xen sample policy</Name>
        <DateTime>2005-05-20T16:56:00</DateTime>
    </PolicyHeader>
    <VM> <id>0</id> <TE>VMMgmt</TE> </VM>
    <VM>
       <id>1</id> <TE>Order</TE> <TE>Advertising</TE>
       <vdsk> <id>/dev/sda1</id> <TE>Order<TE> </vdsk>
       <vdsk> <id>/dev/sda2</id> <TE>Advertising</TE> </vdsk>
    </VM>
    <VM> <id>2</id> <TE>Order</TE> </VM>
    <VM> <id>3</id> <TE>Order</TE> </VM>
    <VM> <id>6</id> <TE>Advertising</TE> </VM>
    <VM> <id>8</id> <TE>Computing</TE> </VM>
</Policy>
</SecurityPolicySpec>
```

Figure 7: sHype Type Enforcement policy example.

icy manager translates this representation into a binary policy that is represented by a table of different TE-type sets. These TE-type sets are referenced in the VM configuration and passed through to the hypervisor when a VM is created. Using this reference, the VM is assigned the ChWall- and TE-types during the the `domain_ops` hook callback into the ACM.