RC23511 (W0502-006) February 2, 2005
Computer Science

IBM Research Report

sHype: Secure Hypervisor Approach to Trusted Virtualized Systems

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

sHype: Secure Hypervisor Approach to Trusted Virtualized Systems[1]

Reiner Sailer *Enriquillo Valdez* *Trent Jaeger* *Ronald Perez*
Leendert van Doorn *John Linwood Griffin* *Stefan Berger*
{sailer,rvaldez,ronpz,jaegert,leendert,jlg,stefanb}@us.ibm.com
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA

Abstract

We present an operating system independent hypervisor security architecture and its application to control information flow between operating systems sharing a single hardware platform. New computing paradigms -such as Grid computing, On-demand services, or Web Services- increasingly depend on the security of the underlying computing infrastructure. A fundamental security problem today is that almost all available security controls for protecting the computing infrastructure rely on the security expected from the operating system. However, common off-the-shelf operating systems are too large and complex to provide the security guarantees required for critical applications. Hypervisors are becoming a ubiquitous virtualization layer on client and server systems. They are designed to isolate operating systems by running them in isolated run-time environments on a single hardware platform. Thus, a malicious or manipulated OS can be isolated and security breaches can be contained within it. However, since distributed services need resource sharing, operating systems must be allowed to co-operate. Our contribution in this paper is the extension of a full-isolation hypervisor with security mechanisms that enable controlled resource sharing between virtual machines to secure this co-operation. We have successfully implemented our hypervisor security architecture (sHype) into a fully functional multi-platform research hypervisor (vHype). sHype implements a security reference monitor interface in the hypervisor to enforce information flow constraints between virtual machines.

1 Introduction

Information services such as Grid computing, On-Demand services, Intranet, Web-Services, data bases, text processing and program development are distributed and rely strongly on the security of the underlying computing infrastructure. Related functional components such as run-time environments, data bases, Web servers, authentication modules, authoring programs, or Web browsers have not only varying but often conflicting security requirements. Consequently, the functional components must be securely isolated against each other to enforce different security requirements. At the same time, components with similar security requirements must be able to effectively and efficiently share information. Usually, security mechanisms are setup to guard isolation and information sharing properties between functional components.

Almost all security controls available in research and both business and personal computing environments to securely separate and control information flow between run-time environments rely on the secure computation environment of their operating system. This means that application sets with varying or conflicting security requirements (e.g., mutually suspicious applications) have to run on separate hardware platforms or rely on the operating systems to isolate the sets and enforce within the sets the different security requirements.

This reflects a fundamental security problem today: the currently available operating system security controls do not solve this isolation issue because they share many critical resources such as shared libraries, file systems, network, and display without strong separation. Additionally, prevalent discretionary access controls (defined as allowing users to determine access right to their data) cannot solve the generic problem of malicious code (viruses etc.) since they cannot separate what a user intends to run from what a user is unintentionally executing. Also, discretionary controls assume that users are acting in an authorized way. Vulnerable applications or careless users may allow malicious code to enter and compromise a system. Similar arguments hold for the protection of data confidentiality.

In an effort to realize strong isolation guarantees and controlled mediation between processes, SELinux [1] has introduced mandatory access control into the Linux operating system. However, the complexity of its security policy makes it impossible to validate security guarantees against security requirements. At the same time, the operating system is still susceptible to attacks (e.g., unauthorized access, running malicious code) by the most powerful user from within the sys-

[1]This is a preliminary publication. Please do not redistribute.

tem. While SELinux represents a significant step towards increasing security in operating systems, the SELinux isolation model is not sufficient to co-locate and securely separate distrusted programs (peer-to-peer applications) from critical applications (Internet banking or critical data base servers).

These problems cannot be solved by adding a higher-level security infrastructure. Considering the most important predicted threats against system security [2] – e.g., malicious developers, trap doors during distribution, boot-sector-viruses, and compiler trap-doors –, effective security cannot be implemented in layers above the operating system (i.e., middleware or applications) because related security controls could be by-passed by those threats. Although integrity checkers, anti-virus scanners and similar security applications are useful to mitigate the risk, they are utterly inappropriate to achieve security guarantees since they may become compromised by the malicious code they are intended to detect.

Hypervisors are becoming a ubiquitous virtualization layer on client and server systems. They are protected against the virtual machines (VMs) and any malicious code running in them. The system that we describe in this paper addresses the above concerns by using a low-complexity and high-performance trusted *hypervisor* layer *below* the OS to implement mandatory security controls for (1) isolating virtual machines by default and (2) sharing resources among virtual machines when desired. By design, hypervisors isolate virtual resources (e.g., virtual LAN, disk, memory, or CPU) they export to operating systems. However, they do not control the sharing of virtual resources between OS.

Our contribution in this paper is the extension of the existing hypervisor resource-level isolation to include access control on virtual resources. Our extension enables administrators to define classes of virtual machines and describe in a formal and verifiable way the necessary requirements for sharing virtual resources between those virtual machines. This formal description of security policy is then interpreted and enforced by access control mechanisms in the hypervisor. To control the sharing of virtual resources based on security policies, we integrated a reference monitor interface into an existing research hypervisor (vHype) for the Intel x86 platform. We implemented the core hypervisor security architecture 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 [3]) 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 run-times serve as an example to illustrate requirements and the use of our hypervisor security architecture. Section 3 describes the related work. We introduce the design of the sHype hypervisor security architecture in Section 4 and its implementation in Section 5. Section 6 illustrates the use of sHype to implement a multi-level secure LAN. Section 7 evaluates our results from a security perspective.

2 Problem Statement

The problem we are addressing is how to effectively and efficiently control information flow between VMs running on the same hypervisor system. Hypervisors offer isolation capabilities at the virtual resource level. However, information flow between VMs can occur through shared virtual resources (e.g., virtual network) or by re-assigning exclusive virtual resources (e.g., virtual disk) from one partition to another.

In this section, we describe what virtualization means and how virtualization is generally implemented. Following this introduction, we motivate how including mandatory access control into hypervisors can generally help to improve the management and run-time security of systems. We then identify the problems that need to be solved to enforce with a high degree of confidence mandatory access control inside the hypervisor system.

2.1 Hypervisor Background

The prevalent approach to create multiple virtual machines on a single real hardware platform is the Virtual Machine Monitor (VMM) approach [4]. A VMM is in complete control of the real system resources. It provides isolated run-time environments by virtualizing and sharing these hardware resources. Such a virtualized environment is called a Virtual Machine (VM). Programs running in a VM show at worst only minor decreases in speed, this is, VMs are taken to be an efficient, isolated duplicate of the real machine [5].

Similarly to VMMs, a hypervisor virtualizes the real system hardware to allow resource sharing and offers VMs (hereafter also called logical partitions) to guest operating systems. To keep its code base small, the hypervisor separates any higher-level services into modular isolated components residing in specialized privileged partitions. Such isolated components implement partition management and include hardware device drivers as an essential part of any virtualization technology.

Figure 1 illustrates the architecture of our vHype research hypervisor into which we have integrated our security architecture. The hypervisor directly controls the real system hardware, such as memory and CPU and I/O interfaces. It creates logical partitions (LPARs), which are virtual copies of the system hardware they share. The hypervisor defers the handling of specific I/O devices to a privileged partition LPAR0 (logical partition 0 in Figure 1), sometimes called the I/O partition. LPAR1 and LPAR2 run guest operating systems (e.g.,

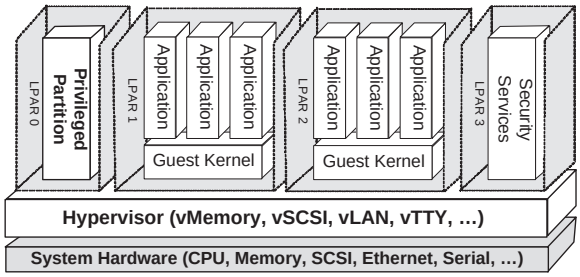

Figure 1: Hypervisor architecture augmented with security services running in an isolated partition.

Linux). Guest operating systems running on vHype 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. Therefore their functionality is exported to logical partitions in a controlled and safe way through the hypervisor call interface. In general, hypervisor calls implemented in the hypervisor have three characteristics: (i) they offer access to purely virtual resources (e.g., virtual LAN), (ii) they speed up critical path operations such as page table management, and (iii) 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.

Security services run in separated and trusted run-time environments (LPAR3) 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 logical partitions to shared resources in our sHype security architecture. Other security services include auditing or partition content attestation.

2.2 Example

As a specific example, we illustrate how mutually suspicious run-times (e.g., corporate servers running in partitions 1, 2 and program development systems running in partitions 3, 4) can safely and securely share a single real system platform by strictly isolating their virtual resources and controlling resource sharing (information flow) between them. Currently, we pursue a medium assurance approach focusing on isolation and explicit resource sharing while minimizing covert storage channels. Nevertheless, we aim to define an architecture that can be extended to meet stronger isolation guarantees by additionally considering covert timing and storage channels between run-times.

To explore how information flows between partitions, we use as an example a hypervisor system running 4 logical partitions in addition to the I/O partition. Partitions 1 and 2 share a virtual LAN labeled A and partitions 3 and 4 share a virtual LAN labeled B. Partitions 3 and 4 also have virtual SCSI devices that can be mapped onto separate partitions of the same real SCSI disk (depicted in Fig. 2) or onto separate real SCSI devices.

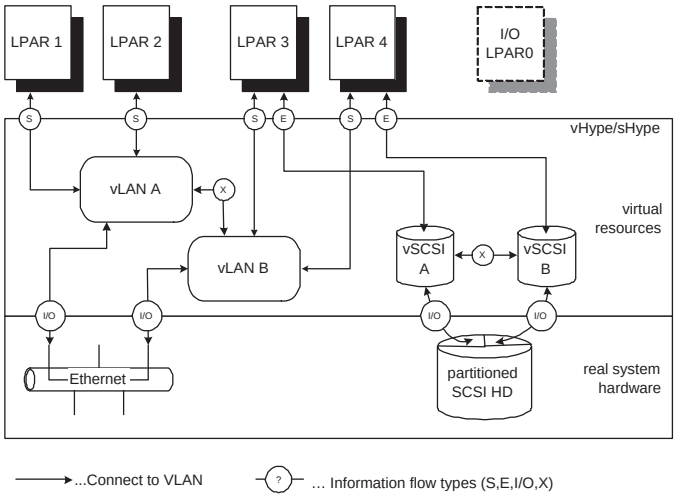

Figure 2: Potential information flow on hypervisors through access to shared resources (S), access to exclusive resources (E), access to real resources actually redirected over the I/O partition (I/O), cross-virtual resource sharing (X).

Before we look at information flow in general, we illustrate a practical scenario by assuming that a virus has infiltrated partition 3 and resides in files on vSCSI A. This might have happened due to running vulnerable applications in a badly protected operating system. This virus can easily infect partition 4 if the operating system in partition 4 mounts vSCSI A, e.g., over the shared vLAN B. If the virus can spread over the network (i.e., is a worm), then it can infect partition 4 directly over vLAN B assuming the partition runs vulnerable server applications. The virus cannot spread into partitions 1 and 2 because there is no sharing of any resources among partitions 3/4 and partitions 1/2 (e.g., no vLAN, no vSCSI). This thought experiment shows that a hypervisor can permit limited resource sharing among partitions, while simultaneously preventing the spread of malicious code to other partitions resident on the system. Similarly to integrity-related confinement, confidentiality-related confinement can be achieved by controlling information flow.

2.3 Information Flow

In general, information flow between two or more partitions can happen if (i) they access the same shared virtual resource (cf access type S in Fig. 2), e.g., a virtual LAN, (ii) they are

assigned the same exclusive resources at different times, e.g., virtual disk, (iii) they access different exclusive virtual resources (cf access type E in Fig. 2), e.g., vSCSI or virtual memory, that are not correctly isolated against each other, or (iv) they access virtual resources that are mapped onto uncontrolled shared real resources (e.g., Ethernet) or exclusive real resources (e.g., SCSI device or real memory) that are not correctly isolated from each other. Some examples will illustrate these scenarios.

Information flow through shared or exclusive resources. Partition 1 and 2 can exchange information by sending and receiving packets over vLAN A; they share the virtual LAN resource (vLAN A). Partition 3 and 4 can exchange information if their disks can be re-assigned from one to the other partition. In this case, one partition could write to the virtual disk and the other one could read from it.

Information flow through non-isolated virtual resources. Partitions 1 and 3 can exchange information if their virtual LANs are connected. Partitions 3 and 4 could share information via their virtual disks if their distinctive virtual disks (vSCSI A and vSCSI B) are insufficiently isolated and allowed addressing of data outside the virtual disk.

Information flow through non-isolated real system resources. Partitions 1 and 3 can exchange information even if their virtual LANs are not directly connected but are connected both to real Ethernet devices or any other shared real medium, the sharing of which is not fully controlled by the hypervisor. Additionally, incomplete separation of real resources can result in information flow. Partitions 3 and 4 can exchange information if their exclusive virtual disks are mapped onto physical storage areas that change (expand, resize) and respective storage is not cleared (object re-use) or if the real memory mappings change from one partition to another one without properly cleaning the information contained in the mapped memory areas.

Information flow through covert channels. Information can be exchanged by observing behavior of other partitions, e.g., one partition modulating information onto system resources and another partition observing changes in these resources (e.g., available bandwidth, memory, disk space, or disk response times). These information flows are usually called covert channels if both parties work together or side-channels if the observer is an adversary. Their bandwidth varies depending on the implementation. This is not directly addressed by our architecture.

2.4 Secure Hypervisors

To provide mandatory access control isolation requires in-hypervisor control of the information flows illustrated in Figure 2 for all virtual resources accessible to partitions. This is the goal of the sHype access control design and implementation described here.

We are applying policy-based access control enforcing inter-partition information flow constraints to achieve the following requirements:

- non-exclusive access of partitions to virtual resources must be guarded and strictly controlled according to the security policy (sHype mandatory access control)
- exclusive assignment of virtual resources to partitions must be guarded and strictly controlled according to the security policy (sHype mandatory access control)
- direct information flow between different virtual resources (type X in Figure 2) must be prevented (hypervisor isolation guarantee)
- information flow between real resources that are mapped to different virtual resources must be prevented (hypervisor isolation guarantee and administration)

sHype guarantees the first two properties. It controls explicit information flow between partitions by use of explicitly shared virtual resources (e.g. vLAN). It also guards the assignment of exclusive virtual resources (e.g. virtual disk). This control cannot be by-passed by partitions but is enforced independently inside the hypervisor.

We build our architecture on the strong isolation properties between virtual resources, which is offered by the hypervisor. Thus, we assume no information flow of type (x) as described in Figure 2. Virtual resources connected by an information flow of type (x) may be considered a single shared virtual resource by the access control primitives in the hypervisor.

We restrict the scope of information flow controls in this paper to a single hypervisor platform and introduce mechanisms that restrict information flow over virtual LANs: no packets are directly passed onto a real network interface. The only way to send or receive data packets from other systems is through the I/O partition, which owns the real network device and can connect to virtual LANs to forward packets as permitted by the security policy. We are currently experimenting with securely connecting multiple hypervisor systems and expanding information flow control beyond a single system.

2.5 Ultimate sHype Security Goals

We have identified the following six security goals for hypervisor environments, all of which are addressed or enabled by the sHype security architecture:

(SG1) strong isolation guarantees between multiple partitions;

(SG2) controlled sharing (communication and co-operation) among partitions;

(SG3) platform and partition content integrity guarantees;

(SG4) platform and partition content attestation;

(SG5) resource accounting and control; and

(SG6) secure services (e.g., auditing).

This paper focuses on information flow constraints (SG1 and SG2) imposed by sHype on partitions. We have work ongoing to investigate all these security goals.

3 Related Work

Mandatory access control has been designed and implemented for the Linux operating system (c.f. SELinux [1]). However, controlling access of processes to kernel data structures has led to an extremely complex security policy. Therefore, SELinux does not enforce strong isolation properties equivalent to those offered when running applications on separate hardware platforms. Operating system security controls, such as those offered by SELinux are more appropriate for enforcing mandatory access control among a set of closely related and cooperating applications, which naturally share a hardware platform. In a hypervisor system, there are few resources shared on the virtualization level. This results in simple security policies when compared to those for operating system controls.

The improved virtualization support of future Intel processors [6] will further contribute to higher performance and allow guest operating systems to run with few or no changes at all inside virtual machines.

Achieving strong isolation between workloads –while minimizing the trusted computing base– has been a major goal of micro-kernel and virtual machine research [4, 7, 8, 9, 10, 11, 12]. Madnick and Donovan [13] show formally that the VMM/OS approach to information system isolation provides substantially better software reliability and security than a conventional multiprogramming OS approach. They argue with redundant security mechanisms that are inherent in the design of most VMM/OS systems.

We take some steps beyond isolation by enforcing formal security models that control resource sharing among virtual machines. This control is based on formal descriptions of access rights between partitions and virtual resources (security policy) enforced inside the hypervisor core. Karger et. al. [14] examine high-assurance virtualization technology of a quite complex Virtual Machine Monitor, while our approach aims at medium-assurance solutions that integrate into existing hypervisor technology.

4 Reference Monitor Design

sHype deploys a reference monitor interface inside the hypervisor to enforce information flow constraints between partitions. Anderson [15] introduces the *reference monitor* as the central mechanism for controlling information flow based on mandatory access controls: A reference monitor enforces the authorized access relationships between subjects and objects of a system; its name refers to the notion that all references by a program to any program, data, or device are validated against a list of authorized types of reference based on user and/or program function. Anderson also states essential design requirements consisting of the reference monitor validation mechanism to be (DR1) *tamper proof*, (DR2) *always* invoked, and (DR3) *small enough* to be subject to analysis and test, the completeness of which can be assured.

Hypervisors offer the ideal place for a reference monitor design that satisfies design requirements DR1, DR2, and DR3. They wrap the shared system hardware, completely control the shared and exclusive virtual resources, and are protected by hardware protection of the CPU from code running in logical partitions. Complementing the reference monitor design, sHype strictly separates access control enforcement from the access control policy according to the FLASK [16] architecture.

Figure 3 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 logical partitions to virtual resources. Enforcement hooks retrieve access control decisions from the *access control module* (ACM).

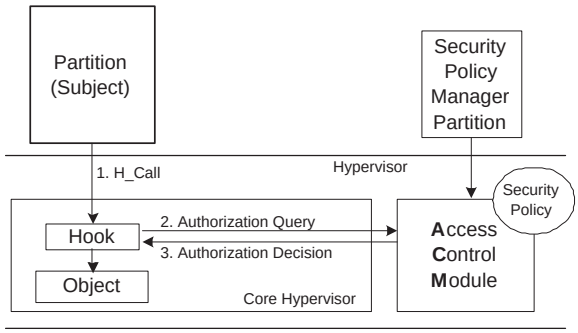

Figure 3: Hypervisor-based security reference monitor.

The ACM applies access rules based on security information stored in security labels attached to logical partitions (subjects) and virtual 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 partitions and logical resources. Finally, a hypervisor-interface allows trusted policy management partitions to efficiently manage the ACM security policy.

sHype follows the FLASK access control architecture by strictly separating the enforcement hooks from the policy. This has several advantages: (i) it promotes a clear design and

implementation, (ii) it confines the implementation-specific impact of policy changes to the ACM module, which is largely independent of the core hypervisor, and (iii) it promotes security policies that are independent of the specific hypervisor implementation, enabling a separation of security management from system administration.

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 partitions)
- allow for the flexible enforcement of various policies, which guard the information flow between multiple partitions

The position of a hypervisor in the system software stack mandates the *highest possible performance* because any penalty will affect the partitions, which depend on the hypervisor for many privileged operations (e.g., memory management, scheduling, or resource sharing among partitions). Consequently, a security architecture that introduces non-negligible overhead will not become a core component of a hypervisor. We adhere to performance requirements by taking access control decisions mostly out of the critical path (e.g., sending packets) and moving them into the less critical binding time of partitions to resources (e.g., when connecting a virtual Ethernet adapter to a logical LAN).

High performance and strong access control demand for trade-offs in the granularity with which resources are controlled. Therefore, we follow a layered approach to systems security. Our hypervisor security architecture controls access to virtual resources on partition-granularity only – for example, sHype can control network-based communication between partitions. Any finer-grained access control is deferred to the partition software stack – for example an operating system implementing SELinux with a policy that restricts communication to specific processes inside different partitions.

Formal security policies specify security relationships between partitions independently of their technical implementation. The unified representation of a security policy enables to define, compare, and validate isolation and sharing properties over multiple hypervisor systems efficiently. Today, static configurations require comparing multiple configuration files that depend on the hypervisor system release and are managed by system administrators that not necessarily understand the overall security goals. Misconfiguration of a single of these files can go easily undetected for a long time, effectively undermining the corporate security policy. A formal policy states the access rules in a way that is common to a large range of hypervisor systems and enables the *separation of the duties* of system administration (functionality-based) and system security (policy-based).

To support business requirements, a trusted hypervisor must support *various kinds of policies*. The sHype architecture supports Biba, Bell-LaPadula, LOMAC, Chinese Wall, Caernarvon, as well as other security policies and adapt to specific business needs.

5 Implementation

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

5.1 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 logical partitions by retaining complete control over the physical resources it depends on (e.g., CPU, memory). On x86 platforms for example, vHype uses *CPU protection rings* to ensure that partitions cannot execute privileged instructions and gain control over resources the hypervisor depends on. The hypervisor runs in "hypervisor" mode in CPU ring 0, the highest privileged protection mode. Logical partitions 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 logical partition running in ring 2 is still protected against its applications running in ring 3. The privileged partition implementing partition 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 partitions) and allows direct access to I/O device memory. The I/O partition is seen as part of the virtual machine monitor infrastructure. The hypervisor depends on its co-operation to manage the peripheral system hardware devices.

vHype isolates virtual resources against each other, such as virtual memory, CPU, vLAN, virtual disks, virtual LAN or shared virtual memory. For example, the vHype *memory* management ensures that logical partitions see only virtual addresses, which are mapped under the control of the hypervisor. The only way to share memory between partitions is through a shared virtual memory resource. Virtual disks are statically assigned to partitions and respective physical storage is safely erased before it is assigned or re-assigned to virtual disks. The *CPU* represents a resource that has long been virtualized to allow 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 partition that will be running on this CPU. Isolation is achieved since no explicit information can flow from the former partition context to the new partition context. Different *vLANs* are also isolated against each other and must be bridged inside partitions, which are subject to sHype access control when connecting to the vLANs.

5.2 Access Control Enforcement

To control the sharing of virtual resources between partitions, sHype mediates access of partitions to exclusive and shared virtual resources. Mediation is implemented by inserting *security hooks* into the code path inside the hypervisor where partitions 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 logical partitions according to the security policy. Each security hook adheres to the following general pattern:

i. gather access control information (determine partition labels, virtual resource labels, and access operation type)

ii. determine access decision by calling the ACM

iii. 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 4 shows the hook that mediates the attachment (binding) of partition 2 to a virtual LAN vLAN A inside the hypervisor. First, the security hook looks up the partition security label range void pointer SSLRP in the partition 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 partition 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 partition access to vLAN A and indicates this to the partition.

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 partition binds to a virtual resource (cf, joining a virtual LAN in Figure 4). Subsequent access to this

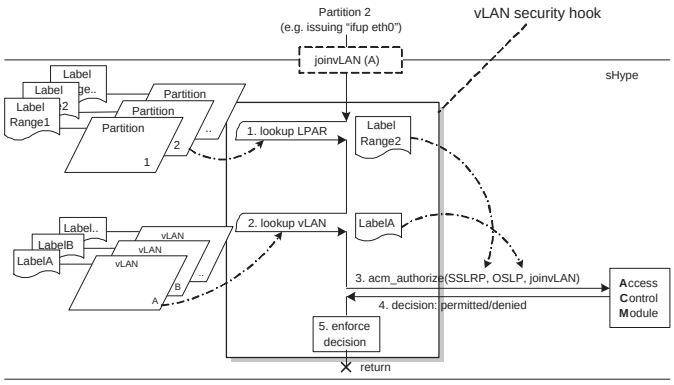

Figure 4: Security hook guarding the joining of a vLAN.

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. The binding must be revoked if the policy changes and the access control decision does not hold any more (cf Section 5.5).

Explicit caching supports access control for virtual resources that do not follow the binding-before-use paradigm (e.g., signals or spontaneous inter-partition communication). In this case, we cache access control decisions locally in the `lpar` partition structure (preserving cache locality) in a single bit per partition 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 partitions 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 allows predicting 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.

5.3 Access Control Module

The ACM stores the current policy, allows 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 partition uses this hypervisor call to manage the security policy stored in the ACM. Privileged partitions 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 partition. 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 partitions or virtual resources are created and initialized in the hypervisor core. The ACM also exports the `acm_authorize` function, which takes a partition 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 partitions to virtual resources. We describe re-evaluating access decisions in case of policy changes in Section 5.5.

5.4 Security Policy

The security policy performs three major tasks. For each virtual resource object, it defines the requirements to access this virtual resource. For each partition subject, it defines the authorizations to access resources. Finally, it defines the access rules that decide whether a logical partition's rights suffice to apply a certain operation to a virtual resource object.

To specify access requirements, we attach security labels to virtual resource data structures (e.g., vLAN). To specify authorizations, we attach security labels to partitions 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 partition data structures are initialized inside the hypervisor (c.f. Section 5.3). 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. lpar, vLAN). Following the implementation of the Linux Security Modules [17], we use `void` pointers to attach label structures to virtual resources and partitions 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 partitions to virtual resources, the partition's label must reflect the required authorization to access the resource. We say the partition'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 [18] 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. Figure 5 illustrates the label structure and interpretation for sHype.

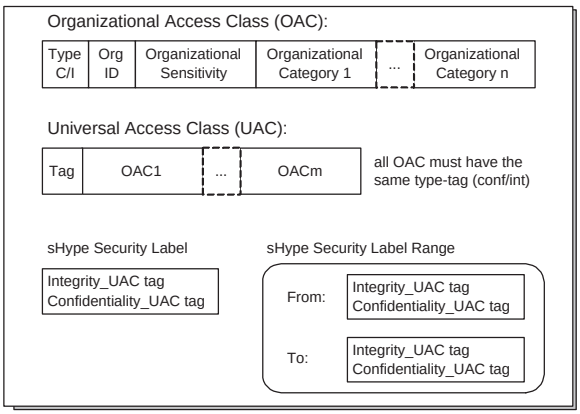

Figure 5: sHype security label structure.

Security labels are composed of organizational access classes (OAC) and universal access classes (UAC). An OAC consists of a type field, describing whether this is a secrecy or an integrity class. Following is the organizational identifier (orgID), which describes the organization that is controlling the meaning of the level information. OACs can only be compared if they have the same orgID (e.g., IBM). We define unclassified, secret, top secret levels for confidentiality classes and low, medium, and high levels for integrity classes. It follows a number of categories, such as Research or Human Resources. UACs allow to aggregate multiple OACs with the same OAC type and to easily reference them by their UAC tag in the security label. For legacy support, we define a special UAC tag *UAC_NONE*, used to label resources that are not part of the security model.

Each virtual resource (object) has attached a security label (describing requirements to access this resource) consisting of one integrity UAC and one confidentiality UAC. Each partition (subject) is assigned a security label range consisting of a *from* security label and a *to* security label denoting its clearance.

For most partitions, the *from* and the *to* labels of the security label range are identical. We call such partitions *single-*

level partitions because they are confined to a single security domain defined by a single confidentiality and integrity UAC. Single-level partitions cannot leak data between different security domains due to our access controls. A partition must be a *single-level partition*, if it is not trusted to keep data of different security levels safely apart. Examples are partitions that can protect the confidentiality of corporate customer data but cannot guarantee that their run-time environment isolates such data securely against unclassified data. Allowing such a partition to handle multiple levels of data would allow data assigned to one security level to leak into data assigned to another security level. From an information-flow perspective, a single-level partition cannot create a data-flow between different security levels or categories inside the partition. It is securely confined by the sHype access control architecture. An integrity-related example would be a partition that is known to run vulnerable software but produce high-integrity output as long as it consumes only high-integrity input. Allowing such a partition to receive malicious (low-integrity input) would allow this partition to produce low-integrity output and thus affect other high-integrity partitions with it. Such a partition must be confined into a high-integrity security domain.

Allowing only single-level partitions would fully isolate different security domains represented by different UACs. However, often there is need to allow controlled information flow between different security domains to implement distributed services. In Caernarvon, trusted entities can be assigned label ranges allowing them to access multiple security domains. In sHype, we call trusted partitions that have label ranges *multi-level partitions*. Multi-level partitions are cleared for multiple security domains. Their clearing is equivalent to all labels that fall into the range spanned by their *from* and *to* labels. However, such multi-level partitions must be carefully designed to maintain the separation of those domains.

The capability to enforce information flow control constraints determines whether a partition can assume multi-level status. Once the hypervisor allows a

partition to participate in different security levels, the hypervisor can no longer independently enforce the boundaries but must rely on the co-operation of the multi-level partition to prevent leaking of information between the levels inside the partition. For example, a trusted router can connect both to a low-integrity and to a high-integrity vLAN and ensure that data can only pass from the high-integrity vLAN towards the low-integrity vLAN or that low-integrity data is properly sanitized before being processed inside the partition or forwarded into the high-integrity vLAN. Another example is a partition implementing the network pump [19], which can enforce network traffic to flow from unclassified to classified domains but not vice versa. To promote the least-privilege principle, sHype allows restricting label ranges to certain virtual resources (e.g., vLAN).

5.5 Change Management

When the policy changes, we must explicitly revoke a shared resource from a partition 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 6 shows the hooks for evaluating and re-evaluating access control decisions for a partition X joining vLAN Y (bind-time authorization). Binding an adapter of a partition 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 partition. 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 partition. 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 partition a revocation looks as if the network cable was unplugged.

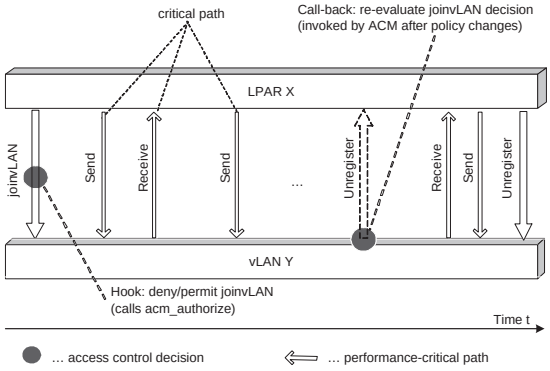

Figure 6: Re-evaluating a joinvLAN access control decision.

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 partitions or resources might prove worthwhile.

6 Multi-level Secure LAN Experiment

Figure 7 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 partitions that do not have confidentiality or integrity requirements. Such a label identifies resources that do not participate in the security model. Only partitions 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 clearing of partitions connecting 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 partition connecting to vLAN B would have to be cleared fore vLAN B's category as well as the confidentiality level.

Figure 7 shows logical partitions LPAR1, LPAR2, and LPAR3 connecting to the virtual LANs A and B. LPAR1 and LPAR2 have the same security label range $\{\{\texttt{ibm\_secret}, \texttt{none}\}_{from}, \{\texttt{ibm\_secret}, \texttt{none}\}_{to}\}$. In contrast, LPAR3 has the subject security label range $\{\{\texttt{none}, \texttt{none}\}_{from}, \{\texttt{none}, \texttt{none}\}_{to}\}$.

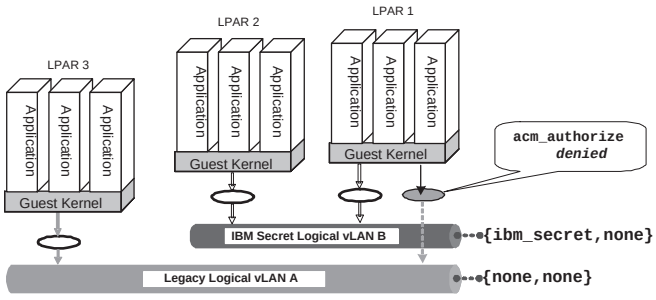

Figure 7: Multi-level secure vLAN based on sHype.

In the above configuration and security policy setting, sHype will allow LPAR1 and LPAR2 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 LPAR3 will be able to connect to vLAN A.

Results. To implement vLAN mandatory access control in vHype, we added one hook into the hypervisor call registering a partition's virtual Ethernet adapter to a virtual LAN. This hook calls acm_authorize as described in Figure 4. We inserted calls to acm_init into the initialization phase of virtual Ethernet adapters and partition 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 vHype implementation (PowerPC versus x86), e.g., the definition of a new H_Security hypervisor call allowing policy partitions to manage the policy within the ACM and the insertion points for the acm_init call for labeling the architecture dependent partition data structure. We have successfully tested the revocation of vLAN access for partitions 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 partition from the vLAN structure. To the operating system inside the partition a revocation looks as if the network cable was unplugged.

7 Security Evaluation

General Approach. From the standpoints of reliability and security, one of the most important aspects of a hypervisor-based system is the high degree of isolation and its potential for controlling access of virtual machines to resources and related information flow. Its ability to provide independent security mechanisms contributes to increased security. We emphasize that mandatory access control, residing completely inside the hypervisor, is beneficial even if the operating system running in a virtual machine deploys its own mandatory access control. It provides a layered approach to security and offers a safety net in case operating system controls fail.

Reference Monitor Design Limitation. The reference monitor concept is predicated upon positive identification of the identity of a partition and the correct correlation of this identity with security control information (here: labels, label ranges). Thus, we assume the correct identification of partitions and resources when initially labeling them. Further, faulty security policies invalidate the security model and offer no provable protection. The model also requires properly operating or fault-tolerant hardware and physical components. Sharing the hardware enables economies of scale of investments in this respect.

Next, following Anderson's study [15], any program that is not subject to invoking the reference monitor to access shared resources must be considered as part of the security "apparatus" and becomes subject to design goals DR1 and DR3. These require that the security apparatus be tamper-proof and small enough to be subject to analysis and tests. Such code includes in our case the hypervisor and any code the hypervisor relies on for its correct functioning and access control. Therefore, separating hardware device drivers into a logical partition aims at an architecture that allows excluding much of the device driver code from becoming subject to DR1 and DR3. Prevalent VMM approaches including non-separated device drivers, partition management, and many more value-added functions in the same protection domain as the core virtualization layer will necessarily fail to achieve DR1 and DR3 due to the large code base.

Finally, there exist *covert channels* between partitions that are not controlled by our reference monitor approach and not prevented by existing isolation mechanisms. These covert channels reflect information that a partition gains through observing changes in the system state that is influenced deterministically by another partition. Available bandwidth, memory, disk space, or CPU are examples of possibly observable system state influenced by partitions. There are well-known ways to limit the bandwidth of such covert channels. However, eliminating all covert channels is not a feasible goal using common off-the-shelf system components.

Reference Monitor Implementation Limitations. The security guarantees implemented by sHype/vHype depend on the hypervisor and the policy management. The implementation of the hypervisor and I/O partition depends on the expected behavior of the hardware against which the developers have written the driver code. Additionally, our reference monitor model requires full control over the hardware, e.g. persistent storage, in order to ensure that controlled sharing above and inside the hypervisor layer is not circumvented by uncontrolled sharing below it. Allowing attackers to access the system hardware or hardware I/O devices would not only question the integrity of the hypervisor and reference monitor implementation but also violate the formal design requirement DR2 (cf Section 5.2) because access to the shared resources through the hardware would not invoke the reference monitor access control.

More specifically, sHype guarantees that any system-internal information flow between partitions is subject to mandatory access control governed by a formal security policy. Our threat model includes malicious behavior of controlled partitions; it excludes malicious behavior or corruption of the hypervisor and privileged partitions (policy partition, I/O partition). To confine the impact of vulnerabilities inside the complex I/O partition, we plan to further divide it into independent parts and restrict their capabilities (security label ranges) by following the principle of least privilege. As opposed to operating systems, which change often, device drivers of the I/O partition in a VMM environment don't change often and therefore justify investments to assure they work correctly. Additionally, separating the device drivers from the core hypervisor and from each other eliminates unnecessary dependencies between these entities by introducing controlled boundaries that contain possible vulnerabilities. Minimizing the trusted computing base for the sHype –i.e., the hypervisor, I/O partition, and policy partition– is a continuous design goal.

8 Conclusion and Outlook

We presented a secure hypervisor architecture, sHype, which we are implementing into the vHype 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 partition, 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 [20] and a related Integrity Measurement Architecture [21]. 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] Peter Loscocco and Stephen Smalley, "Integrating flexible support for security policies into the Linux operating system," in *Proceedings of the FREENIX track: 2001 USENIX annual technical conference*, June 2001.

[2] 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.

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

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

[5] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, July 1974.

[6] Intel, "Vanderpool Technology," http://www.intel.com/technology/computing/vptech/.

[7] Jonathan S. Shapiro et.al., "EROS: The Extremely Reliable Operating System," http://www.eros-os.org/.

[8] M. Hohmuth, H. Haertig, and J. S. Shapiro, "Reducing tcb size by using untrusted components – small kernels versus virtual-machine monitors," in *11th ACM SIGOPS European Workshop (EW 2004)*, September 2004.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[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] 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.

[18] 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.

[19] M. H. Kang, I. S. Moskowitz, and D. C. Lee, "A Network Pump," *IEEE Transactions on Software Engineering*, vol. 22, no. 5, pp. 329–338, 1996.

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

[21] 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.