RC23778 (W0511-035) November 4, 2005
Computer Science

IBM Research Report

Bridging Mandatory Access Control Across Machines

Jonathan M. McCune[1], Stefan Berger, Ramón Cáceres,
Trent Jaeger[2], Reiner Sailer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[1]currently at
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Pennsylvania State University
University Park, PA 16802

Bridging Mandatory Access Control Across Machines

Jonathan M. McCune[*] Stefan Berger[†] Ramón Cáceres[†] Trent Jaeger[‡] Reiner Sailer[†]

Abstract

We define and demonstrate an approach to securing distributed computation based on a *distributed reference monitor* that enforces mandatory access control (MAC) policy across machines. Securing distributed computation is difficult because of the asymmetry of trust in different computing environments and the complexity of managing MAC policies across machines, when they are already complex for one machine (e.g., Fedora Core 4 SELinux policy). We leverage recent work in three areas as a basis for our solution: (1) remote attestation as a basis to establish mutual acceptance of reference monitoring function; (2) IPsec with MAC labels to ensure the protection and authorization of commands across machines; and (3) virtual machines for isolation and to simplify the MAC policies. We define a distributed computing architecture based on these mechanisms and show how local reference monitor guarantees can be attained for a distributed reference monitor. We implement a prototype system on the Xen hypervisor with a trusted MAC VM built on Linux 2.6. This prototype enforces MAC between machines using IPsec extensions to SELinux that label secure communication channels. We show that through our architecture distributed SETI@HOME computations can be protected and controlled coherently across all the machines involved in the computation.

1 Introduction

Distributed applications have many security and privacy requirements that are not being met by today's computing and networking infrastructure. Consider volunteer distributed computing efforts such as SETI@Home. SETI@Home [2] uses large numbers of Internet-connected computers to analyze radio telescope data as part of the Search for Extraterrestrial Intelligence (SETI). The public at large donates idle compute cycles on their personal computers. These client machines obtain software and data from the SETI@Home server, perform their assigned portion of the analysis, and return their results to the server.

Presently, systems like SETI@Home are constrained by security concerns [33]. Administrators cannot ensure that clients run the analysis software as intended. They thus send the same analysis task to multiple clients and compare the results for consistency, which reduces throughput while still not guaranteeing correctness. On the other hand, volunteers are uncertain of the origin and integrity of the software they download. They must blindly trust that this software will not harm their personal computing environments.

Similar concerns arise in many other distributed computing scenarios. For example, web hosting services would like to run workloads from more than one enterprise customer on the same server machine, both to increase hardware utilization and to decrease operational costs. However, remote users of these servers need assurances that one enterprise's software does not interfere with the operation of another's. As another example, many companies outsource the management of their computing infrastructure to a service provider. They need to allow the provider access to management functions while protecting other aspects of their online operations.

To address these concerns, we need to isolate workloads that have different security requirements from each other. Previous distributed access control schemes such as the Logic of Authentication [28] and trust

[*]`jonmccune@cmu.edu` Carnegie Mellon University, Pittsburgh, PA 15213 USA This work was done during an internship at IBM Research.

[†]`{stefanb,caceres,sailer}@us.ibm.com` IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA

[‡]`tjaeger@cse.psu.edu` Pennsylvania State University, University Park, PA 16802 USA

management [8] are discretionary, meaning they let users grant rights to objects they own. They therefore cannot protect against misbehavior in programs that run on a user's behalf. In addition, validating their information-flow properties is complex because it requires consideration of all programs on a system.

In contrast, Mandatory Access Control (MAC) ensures that system security policies are enforced regardless of the behavior of users and their programs. Furthermore, the trusted computing base in a MAC system is much smaller, because it does not depend on the behavior of users or their programs, or even most system programs. Therefore, MAC works in the presence of malicious code, and its properties can be validated after inspecting a small portion of the system. MAC systems that operate on a single machine have a long history [3] and have recently been developed for commodity operating systems like Linux, e.g., SELinux [42, 45], and hypervisor systems like Xen [4], e.g., sHype [36].

We seek to enable MAC across multiple machines. We refer to such controls as *bridging* the systems that they include. A system that enables bridging of MAC policies is called a *distributed MAC system*, and bridged MAC policies are enforced by a *distributed reference monitor*.

We have designed and implemented such a distributed reference monitor. We use a hypervisor with MAC as a foundation. A key advantage of this platform is that we can compose distributed systems from collaborating groups of virtual machines, which we call *coalitions*, each with independent MAC policies. Our solution addresses the following requirements:

1. *Distributed tamper-proofness*: All systems forming the distributed reference monitor must be tamper-proof. A new coalition member is aggregated into a coalition's distributed reference monitor by using integrity attestation to determine that each system's reference monitor is protected against tampering before extending the distributed reference monitor. In addition, data flowing between two systems must be protected from access that would violate the policy. We use integrity-protected communication tunnels between systems.
2. *Distributed mediation*: All access by subjects to security-relevant objects must be mediated by the reference monitor. We again use integrity attestation to determine that each coalition member's reference monitor completely mediates the MAC policy operations before extending the distributed reference monitor. In addition, the semantics of security labels must be preserved on subject requests and object responses when they move between systems. We leverage recent work in labeling secure communication tunnels to determine the security labels of information flows between systems.
3. *Compatible security policies*: All systems must enforce compatible policies. In this case, the coalition's MAC policy is conveyed to the new system in the coalition after we use attestation to verify compatibility of MAC policies between remote systems. For example, MAC policies may be incompatible due to Chinese Wall [9] restrictions between coalitions on a system. In addition, it is important to keep the policies simple to facilitate verification that they express the desired security properties. We satisfy this goal by having policies operate at a coarse grain, for example at the granularity of whole virtual machines.

The rest of this paper is organized as follows. Section 2 establishes some necessary background for the ensuing material. Section 3 presents the architecture of our distributed reference monitor, and Section 4 describes our prototype implementation. Section 5 examines an experimental evaluation of the security features of the prototype implementation. Section 6 discusses some outstanding issues and areas for future work, while Section 7 surveys related work. Finally, Section 8 offers our conclusions.

2 Background

In this section, we examine the requirements for bridging systems and review the technologies that we hypothesize will be fundamental to enabling bridging.

2.1 Example

We return to our volunteer distributed computing example from Section 1 to motivate the need for distributed MAC. SETI@Home [40] is built on the Berkeley Open Infrastructure for Network Computing (BOINC) [1]. The BOINC architecture includes client software which automatically runs – when users' CPU would otherwise be idle – a compute application provided by SETI@Home (e.g., processing data from radio telescopes).

Security is a significant problem in volunteer distributed computing projects [33]. Large numbers of users are downloading and executing unknown software on their computers, where the software environment is likely to be insecure. Presently, there is no simple mechanism by which clients can protect themselves from downloaded executables. Likewise, the server has no means to enforce that the client has executed its software correctly, or in an acceptable environment.

Risks to which the server is exposed include: (a) accepting invalid results; (b) accepting duplicate results; (c) giving a user undeserved credit; and (d) accepting results carefully crafted by a malicious entity in an attempt to exploit vulnerabilities in the server software (e.g., buffer overflows). A security system that protects servers should enable strong authentication of users and validate the integrity of their systems to correctly distribute credit (c) and identify messages from legitimate systems (a, b, d). If a user's system is verified to be legitimate, but it provides invalid or malicious messages, then our security system will not be able to detect this. This is an application-level problem.

Even if we assume today's web security technologies (e.g., SSL/TLS) are secure, spoofing attacks continue to be a significant threat to clients. Thus, a user cannot be sure that the client software she downloads is legitimate. To reduce the level of risk to which the user is exposed, it is desirable to execute the client software in an isolated environment. However, this is at odds with the primary goal of the BOINC project – to allow ordinary users with commodity PCs to perform computations. A security system that protects clients should enable strong authentication of servers and verification of server integrity as well as providing an isolated environment for executing remote code.

2.2 Technologies

We identify three technologies as fundamental in bridging MAC across machines: (1) remote attestation; (2) secure, MAC-labeled network communication; and (3) virtual machine monitors.

Remote Attestation A *remote attestation* mechanism enables a party (e.g., process) on one machine to prove security properties about itself to another party on another machine [44]. Since software can be modified to forge such information, a trusted hardware platform, such as the Trusted Computing Group's Trusted Platform Module (TPM) [46], is used to implement attestation mechanisms.

Fundamentally, attestation using TPMs aims at proving that a deterministic boot process is performed on a system [31]. Remote attestation has also been proposed as a means for proving system integrity [35] and computation integrity [41]. In the former case, the remote party can determine whether the system is using acceptable programs, libraries, and configurations in addition to its successful booting. In the latter case, the computation integrity can be verified based on attestation of inputs and the code used in each step.

BOINC use of Remote Attestation: To build a distributed reference monitor for a BOINC system, we want to: (1) verify the identity of reference monitor components that comprise the distributed reference monitor and (2) verify that these components provide the necessary mediation abilities (e.g., access checks and base MAC policies). First, the BOINC server and clients can verify that each run acceptable reference monitoring software and MAC policies (e.g., tamper-proof TCB components). Second, the BOINC server and clients can each verify that the mediation abilities of this software and MAC policy are acceptable for the computation. We use the open-source IMA attestation system for Linux [35] to achieve these goals.

Secure, MAC-Labeled Network Communication We leverage secure, MAC-labeled communication for both tamper-responsive data transfer (i.e., enable dropping of tampered messages) and MAC mediation of network requests. Recent work that integrates IPsec and Linux MAC provides both functions [18]. IPsec provides mutually-authenticated, tamper-responsive network communication. In the extended Linux system, the IPsec security associations are given labels based on the MAC policy, such that Linux MAC (e.g., the Linux Security Modules framework [50] using the SELinux module [49]) can control network communications between remote applications. Such a mechanism is an improvement over previous mechanisms, such as the IP Security Options (IPSO) extensions [23] that carry labels in the IP packet headers. The latter involves significant packet processing overhead even for unlabeled packets [50].

Mediation of network communications has been a challenge due to the difficulty in converting secure channels to authorizations that should be permitted. A variety of network security architectures have been devised over the years, such as Lampson's Logic of Authentication [28] and its descendants (e.g., WebOS [6] and Distributed Proving [5]) and trust management systems [8, 12, 29, 30]. Such approaches enable the construction of DAC policies from fine-grained statements, so MAC guarantees are difficult to achieve.

BOINC use of Secure MAC-Labeled Network Communication: The BOINC server's reference monitor shares the BOINC system's MAC policy with the BOINC client's reference monitor. The BOINC client's reference monitor must determine whether the MAC policy covering the BOINC client in the distributed system is compatible with its overall policy. For example, the BOINC client's system may be running another computation whose secrecy requirements do not permit the BOINC client from running (e.g., to protect against covert channel leaks). Note that the BOINC server already uses the MAC policy and has verified the ability of the BOINC client's reference monitor to enforce it earlier. We use the MAC-labeled IPsec implementation in Linux [18] to support secure communication and MAC control on those remote communications once the BOINC MAC policy is accepted.

Virtual Machines A *virtual machine monitor* (VMM) enables the execution of multiple virtual machines on one platform. While virtual machine systems have been around for many years [11, 32] and high assurance VMMs have been built [22], the recent emergence of Xen open source hypervisor, available on common platforms and supporting MAC enforcement [36], has energized a large community.

Virtual machine monitors provide isolation of individual VMs running different operating systems. Also, the MAC enforcement in Xen enables mediation of inter-VM communications on the same platform. By using VMs, coarse-grained, but simpler, MAC policies can be enforced. For example, a BOINC client can run isolated from other local VMs, so that if all code in the BOINC VM can be similarly trusted, it can all run using the same subject label. This contrasts with a SELinux policy for a Linux system where many programs run in different labels, resulting in tens of thousands of policy statements to manage.

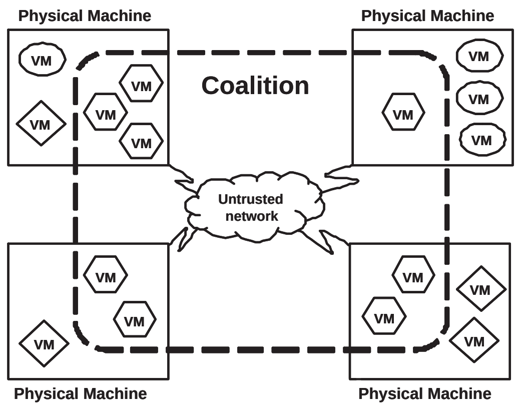
BOINC use of Virtual Machines: The BOINC client is run in a dedicated virtual machine running on the open-source, Xen hypervisor [4] with the sHype MAC controls [36]. The BOINC server ships the code to the BOINC client machine where its Xen trusted VM manager `dom0` initiates the BOINC client VM with the appropriate label based on the BOINC MAC policy. Xen controls local inter-VM communication, and the BOINC server can get an attestation that this VM is running the client code that it sent prior to accepting the results. We also use IMA for this attestation [35].

3 System Architecture

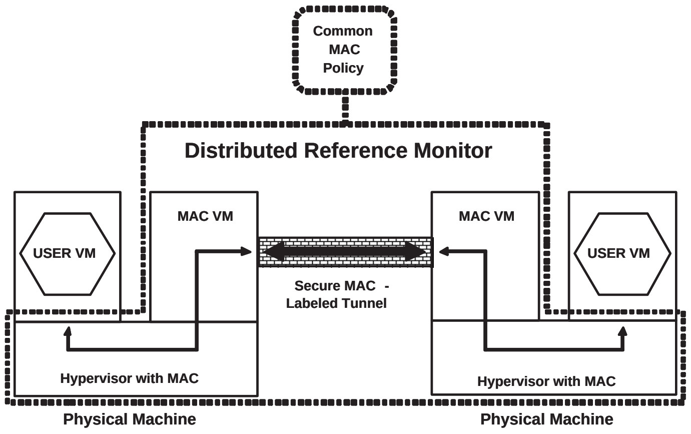
In this section, we outline the system architecture for a distributed reference monitor and examine its ability to achieve the guarantees of a host reference monitor across a distributed environment.

3.1 Architecture Overview

The distributed reference monitor architecture is shown in Figure 1(a-b). In Figure 1(b), each machine has two types of software running on it: (1) user virtual machines (user VMs) where the application processing



(a)



(b)

Figure 1: (a) Example of a distributed VM coalition. (b) Example of a distributed reference monitor.

is performed and (2) supervisor software, such as a hypervisor and trusted MAC virtual machine (MAC VM) on a hypervised system, that service and control the inter-VM operations of user VMs. For example, the BOINC server and clients run as user VMs. The supervisor implements the local system reference monitor. On a hypervised system, the hypervisor determines the MAC policies for the VMs and mediates operations that it performs (on VMs and inter-VM communication on the local machine). Network communication is implemented in the MAC VM, so the MAC VM is responsible for mediating network communications using the MAC policy of the hypervisor.

Sailer et al. define a *coalition* as a set of one or more user VMs that share a common policy [36] as shown in Figure 1(a). For example, a set of user VMs that may communicate among one another, but are isolated from all other user VMs would form a coalition. Each user VM would run under the same label, and all would have read-write access to user VMs of that label. We note that other access control policies are possible within a coalition. In another case, the coalition user VMs can be labeled with secrecy access classes and interaction is controlled by the Bell-LaPadula policy [7].

The problem addressed in this paper is the extension of these coalitions across multiple machines. As shown in Figure 1(b), the resultant inter-machine coalition has its MAC policy enforced by a *distributed reference monitor*. The distributed reference monitor constructs a secure communication tunnel to protect the secrecy and integrity of communications over the untrusted network between them. Further, the tunnel is labeled, such that both endpoint reference monitors in the distributed reference monitor can control which user VMs can use which tunnels.

We define *bridging* to be the act of constructing or extending distributed reference monitors. A bridged coalition has been verified to enforce its MAC policy across all reference monitors that are bridged. This verification depends on two additional policies that must be common to the coalition: (1) a secure communication policy to protect the secrecy and integrity of communication and (2) an attestation policy to cover how tamper-proofing, mediation, and compatibility guarantees are verified. First, because interactions between reference monitors must be tamperproof and may require secrecy, a secure communication channel is required between each pair of reference monitors in the distributed system. Second, local reference monitors must be able to verify the tamper-proof and mediation properties of another reference monitor prior to constructing the bridge and incrementally during the collaboration.

3.2 Setting up a Bridge

When the first user VM of a system joins a coalition, its reference monitor bridges with the coalition's distributed reference monitor. The following steps are necessary to complete the bridging process: (1) the new reference monitor needs to obtain the coalition's configuration: its MAC, secure communication, and attestation policies; (2) using the attestation policies, the reference monitor and the distributed reference monitor mutually verify that the tamper-responding and mediating abilities are sufficient for the bridging; (3) the new user VM is installed, if it had not already; and (4) the secure, MAC-labeled network communication of the bridge is enabled. When we say distributed reference monitor below, we mean a reference monitor that is a representative for the coalition.

First, the reference monitor joining the coalition must have the coalition's configuration. There are different ways that the reference monitor can obtain this configuration. For example, the reference monitor may have its own configuration and a means for translating coalition configurations to its labels. In a coalition that uses a single label, the label name may be translated to one the reference monitor understands. In this case, coalitions may more easily interact, but effort is required to predefine a universal label semantics and syntax into which coalition labels of the local system can be translated. An alternative is to have the distributed reference monitor push a configuration to the reference monitor and have the reference monitor enforce coalition-specific policies. In this case, coalitions would be isolated since the knowledge of how to combine them is not included. Our prototype uses the former approach, so the MAC policy is fixed at the hypervisor level and coalition policies are mapped to it.

Second, the attestation policy is used to verify reference monitor tamper-responding and mediation abilities. For example, we require attestations of the hypervisor and MAC VM code, as well as the MAC policy each system has used. This identifies the initial state of the system, its isolation mechanism, its reference monitoring mechanism, its filtering mechanism, and which low integrity flows use the MAC VM. Our prototype attests to the Xen hypervisor code, MAC VM code, and the hypervisor MAC policy.

Third, the distributed reference monitor may provide code for the new user VM. In this case, the reference monitor constructs the new user VM, and assigns it a MAC label (e.g., based on attestation of the code). If the user VM is already present, then this step is skipped and we proceed to the fourth step. In our case, the user VM is already present and labeled (e.g., *green*), but it receives the BOINC code from the server and attests its integrity to prove to the BOINC server that this code was used.

Fourth, we construct a secure, MAC-labeled tunnel for the bridge in the MAC VM. The secure communication policy is selected when the user VM attempts to communicate with a coalition member and determines the secrecy and integrity requirements of the communication (e.g., AES encryption with message authentication code integrity protection) as well as the MAC label for the tunnel. The MAC label determines which endpoint VMs have access to the tunnel. For example, only a *green* user VM may have access to *green* tunnels and only to *green* tunnels, so an isolated coalition can be constructed. A MLS MAC policy may be used to enable information flows among labels. Our prototype uses the MAC-labeled Linux IPsec implementation in the MAC VM to construct and control access to tunnels for user VMs.

3.3 Distributed Reference Monitor Guarantees

In this section, we examine how the bridging approach described in the last section results in a distributed reference monitor that achieves classical reference monitor guarantees.

A *reference monitor* is a tamperproof component in the MAC system that mediates access to all security-relevant operations [3]. Further, the reference monitor component is ideally simple enough to enable formal verification. There are a variety of classical systems that can be classified as MAC systems, such as Multics [38], PSOS [13], and GEMSOS [37], and recently, there has been motivation to convert commercial operating systems, such as Linux with SELinux [42, 45] and TrustedBSD [47], to MAC systems, including executing such systems within a virtual machine of a hypervised system [4].

A *distributed reference monitor* consists of a reference monitor that spans multiple machines. The challenges are to: (1) provide guarantees of tamperproofing and complete mediation across the distributed reference monitor components and (2) manage the complexity of the reference monitor components.

Making Tamperproof

A tamperproof system is one that is initialized in a correct manner and subsequently isolated from outside tampering. A classic example is that of securely booting an operating system where the secure boot process ensures correct initialization and the hardware protection ring mechanism provides isolation. In addition, the system must be capable of filtering low integrity requests without compromising its integrity, such as in the manner of Clark-Wilson integrity [10] (i.e., be guaranteed to either discard or upgrade their integrity).

If we look at this formally, we find that for a single MAC system it must satisfy initialization (i.e., $init(s)$), isolation (i.e., $isolate(s)$), and filtering (i.e., $filter(s)$) requirements to be deemed tamperproof.

$$\mathit{init}(s) \wedge \mathit{isolate}(s) \wedge \mathit{filter}(s) \rightarrow \mathit{tamperproof}(s)$$

In a distributed system, there are three additional challenges: (1) any system s must be able to prove to any other system in S that its initialization, isolation, and low integrity input filtering are sufficient; (2) the other systems must be able to detect tampering on system s; and (3) the communication between reference monitor components over an untrusted network must be protected. First, the attestation policy defines the requires that must be met by s to be untampered. Second, attestation does not prevent tampering; it only enables detection of tampering via a so-called *authenticated boot*. Thus, we do not prove that s is tamperproof, but that we can respond to tampering with s (i.e., $responding(s)$ by dropping messages). Third, we must add a statement to protect the integrity of inter-reference monitor communications over an untrusted network (i.e., $comm(s)$). The result is the *distributed tamperproof rule* below.

$$\forall s_i, s_j \in S : \mathit{comm}(s_i, s_j) \wedge \mathit{responding}(s_i, s_j) \rightarrow \mathit{responding}(S)$$

Note that both the secure communication and the tamper-responsiveness of a pair of systems are bidirectional properties. The distributed reference monitor architecture satisfies the distributed tamperproof rule by: (1) checking attestation policies to detect tampering in initialization, isolation, and filtering and (2) setting up secure communication channels between pairs of systems. An obvious optimization is to for all systems in the coalition to reuse the same attestations and secure communication channels, rather than generating them for each pair. For example, MAC-Labeled IPsec can use the same IPsec policy for all hosts in the same coalition, regardless of location.

Ensuring Mediation

Complete mediation for system s implies that the reference monitor can authorize all execution paths to process a request req that is able to access any security-relevant operation [*].

$$\forall req \in REQ(s) : \mathit{authorize}(s, op, label(req)) \iff \mathit{access}(s, op)$$

Upon processing a request req, a security-relevant operation op may be encountered. In order to perform the authorization, the MAC label of the subject responsible for the request must be determined. In a host MAC system, such as SELinux, the operating system knows the identity of the processes and can easily retrieve their labels. In addition, any subject label transitions are local to the reference monitor.

[*] Here an operation op refers to both the object and the operation to be performed.

In a distributed environment, a source system generates a request and a destination system (perhaps the same) performs the request. Interestingly, the ability of the source to perform a request on the destination is mapped to the destination's ability to receive the request. For example, a high integrity subject cannot process a low integrity request. If the source subject is authorized to use the communication channel on the source machine, it can perform requests on the destination that is authorized to receive it. The function $label(channel(req))$ returns the label of the communication channel.

We briefly note that the object labeling semantics are the same between local and distributed reference monitors. The object label is determined by the destination and when read by the subject, the subject can create a new object with a label of its choosing. For example, opening a file into memory and writing to a new file with a new label is permissible whether the file is local or remote.

The source and destination both require the destination to mediate all security-relevant operations. Mediation properties of code will require a justification, such as a complete mediation analysis [52]. However, this is no different than the single system case, except that a remote attestation is necessary to verify remote mediation properties (i.e., $mediate(s_d)$).

The *distributed mediation rule* reflects the additional requirements on labeling of requests.

$$\forall req \in REQ(s_s, s_d) : mediate(s_d) \wedge authorize(s_s, channel : write, subj_s) \wedge$$
$$authorize(s_d, op, label(channel(req)) \iff access(s_d, op)$$

In the distributed reference monitor architecture, the initial attestation verifies the mediation abilities of the hypervisor and MAC VM. The source authorizes the local subject $subj_s$ to write the request to the channel, and the channel's label determines whether the request can be fulfilled on the destination.

Simplifying the Design

An original requirement for a reference monitor is that its design be sufficiently simple to enable formal verification [3]. While experience has shown that formal verification is very difficult and costly to complete, simplifying the system will make it easier to define practical semantics for the functions in the distributed tamperproof and mediation rules above.

Stating such a requirement formally is tantamount to stating a formal verification requirement, so we only address this goal informally at present. Simplifying the design involves limiting the sizes of the code in the TCB and the policies the TCB relies upon. The use of hypervised systems and specialized VMs may enable a near minimal TCB size, but much engineering work is necessary to achieve this. The current Xen MAC VM, called `dom0`, is a complete Linux system. The use of a MAC VM should radically simplify the policy required relative to a normal Linux MAC policy (e.g., SELinux [43]). Attestation policies are not yet well understood, so this work will help to define the requirements.

3.4 Limitations

The distributed reference monitor architecture is not without some limitations discussed below.

Hardware Attacks This architecture does not protect the system against cracking of keys via hardware attacks. As such, attestation needs to obtain guarantees regarding protection from such (e.g., TPM within location that assures such protections).

Initialization While discovering that an initial value is wrong is easy (e.g., Tripwire [26]), proving that an initial configuration is correct is difficult, particularly for the mutable input data to a system. The proposal for Integrity Verification Procedures (IVPs) for the Clark-Wilson integrity model has been met with very few examples [10]. Attestation enables verification of the initial state of code and static data, but not for mutable data.

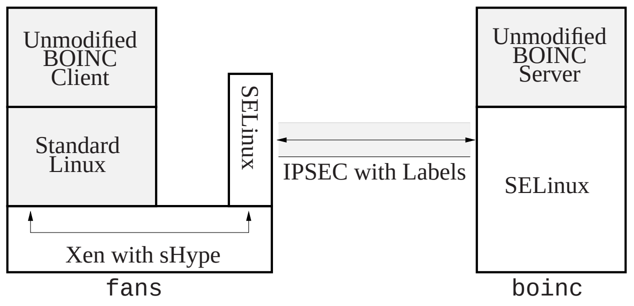


Figure 2: MAC Bridging in our distributed computing prototype. Shaded regions indicate software that is not modified by MAC Bridging. `fans` and `boinc` are the local DNS names for the machines in our experimental setup, which we use to simplify discussion.

Runtime Tamper-Responsiveness The Integrity Measurement Architecture (IMA) [35] measures inputs before loading. Thus, runtime tampering may go undetected. Other techniques, such as Copilot [19] and BIND [41], aim to provide some runtime guarantees, as well, but they face other obstacles, such as preventing circumvention and annotation effort.

Misbehaving Coalition Member This architecture does not protect a user VM from a coalition member that is misbehaving in ways that are not detected by the tamper-responding mechanisms. Since load-time guarantees do not cover all runtime tampering, such issues are possible.

Enforcement Limits The individual reference monitors will not have complete formal assurance, so some information flows, such as covert channels, may not be enforced. The protections afforded by reference monitors should be stated in attestation policies, so that the creation of incompatible coalitions on the same system is not allowed. The Xen MAC policy enables some expression through Chinese Wall conflict sets [36].

4 Implementation

We implemented a distributed MAC system for volunteer distributed computation. We configured a dedicated BOINC [1] server on top of SELinux [43] and a hypervisor system running Xen [4] and sHype [36] to support BOINC clients. We have an SELinux-based MAC VM running on the hypervisor system which performs the necessary policy translation from labels on an IPsec [24, 25] tunnel. Our implementation is based on Simple Type Enforcement (STE) policy. Figure 2 shows the architecture of our prototype. We provide a high-level description of its operation, followed by a detailed description of the implementation including unexpected obstacles we encountered during its construction.

4.1 Machine Configuration

We used two machines in our experiments, `boinc` and `fans`. `boinc` is a 2.4 GHz Pentium IV with 1 GB of RAM and a 512KB cache. `fans` is a 3.4GHz Pentium IV with 3GB of RAM and a 512KB cache. We will provide more details as appropriate, referencing the machines by name for convenience.

`boinc` runs a minimal Fedora Core 3 installation with SELinux configured in *strict* mode. `fans` runs the latest Xen-unstable with sHype and a Simple Type Enforcement policy. The supervisor VM on `fans` is a minimal Fedora Core 3 installation with SELinux configured in *strict* mode.

Device Driver and MAC VMs on Xen We built and maintain our distributed computing client prototype (`fans` in Figure 2) on the current unstable development version of Xen 3.0: `xen-unstable`. While one of the design goals for Xen 3.0 is the ability to assign various physical resources to device driver VMs, such functionality is not currently implemented by `xen-unstable`. When `xen-unstable` boots, it starts a special privileged VM with ID 0 called domain 0, or `dom0`. `dom0` has access to all devices on the system, thus, in our prototype, we have only a single device driver VM – `dom0`.

Our version of `xen-unstable` has sHype enabled and enforces a Simple Type Enforcement (STE) policy. `dom0` runs SELinux and serves as the MAC VM that does policy translation between the labeled IPsec tunnel and local sHype types. The SELinux policy needed on `dom0` is significantly smaller than an SELinux policy for a typical Linux distribution.

Distributed Coalition Membership without Xen Our distributed computing server prototype is an Apache / MySQL / PHP server running on top of SELinux (`boinc` in Figure 2). It runs the BOINC server software and issues compute jobs to clients, collects and tabulates results, and makes status information available via the website it hosts.

4.2 Labeled IPsec Tunnels

We use labeled IPsec connections operating in tunnel-mode [25] as the secure communication mechanism between `boinc` and `fans` (the machines in our distributed coalition). We first describe the role of the labeled tunnels in the distributed MAC system, and then describe their implementation.

Packets arrive in `dom0` on `fans` having come in over the labeled IPsec tunnel from another machine in the distributed coalition. The first check is that these packets are destined for some `domU` on the local hypervisor system (packets with any other destination are trivially dropped using `iptables` rules in `dom0`).

The packets in a flow destined for a `domU` on the local hypervisor system must pass through a reference monitor before being delivered. It is the responsibility of the MAC VM code in `dom0` to perform the translation between SELinux subject labels on the IPsec tunnel and the sHype labels on each `domU`. As illustrated abstractly in Figure 1, reference monitor functionality exists in both the endpoint of the IPsec tunnel (*OS type check* – `dom0`) and in the hypervisor (*hypervisor type check* – sHype).

The OS type check occurs automatically as part of the normal operating behavior of our IPsec configuration. The IPsec tunnels that we employ use tunnel-mode extensions to a prior patch by Jaeger et al. [18]. These researchers added support for SELinux subject labels to be included in the negotiation process when IPsec connections are established. This functionality is achieved through additions to three code bases: (1) the `racoon` Internet Key Exchange (IKE) [15] daemon which does all negotiation for IPsec connection establishment; (2) the `setkey` application which adds and removes entries from the IPsec Security Policy Database (SPD); and (3) the netfilter and Linux Security Modules (LSM) hooks in the Linux kernel where IPsec packets are processed.

The functionality provided by the enhanced `racoon` of Jaeger et al. provides the necessary guarantee that all IPsec packets will have subject labels that are known to both endpoints. That is, an IPsec connection cannot be established without both endpoints having an entry for the tunnel label in their respective IPsec and SELinux policies. Thus, packets with unknown labels will never arrive via an established IPsec tunnel.

In our current implementation, the IPsec policy for each `dom0` (acting as a MAC VM) in a distributed coalition must be preconfigured with all possible SELinux subject types that may be needed by `racoon` in a negotiation to establish an IPsec tunnel. However, recent work by Yin and Wang shows that it is possible to add new IPsec policy on the fly [51].

4.3 Bridging Reference Monitor

The IPsec tunnel(s) between machines in a distributed coalition provide authenticated, encrypted communication while conveying MAC type information. This information is applied in the enforcement of sHype policy. That is, the IPsec tunnel and MAC VM are tools which help to ensure that machines in a distributed coalition enforce semantically equivalent sHype policies. To achieve this goal, we must translate between SELinux types and sHype types.

We modified the authorization hook in the IPsec extensions of Jaeger et al. to call our own authorization function for IPsec packets destined for some `domU`. SELinux subject labels for making authorization decisions are inferred from the sHype label of the `domU` to which flows are destined, or from which they originate. On `xen-unstable`, the OS running in each `domU` has a virtual network interface driver known as a *frontend*. The *backend* drivers for all these virtual network interfaces reside in `dom0`, manifested in the form of additional network interfaces. sHype mediates communication between frontends and their corresponding backends inside the hypervisor.

Our authorization function (see pseudocode in Figure 3), `get_sid_from_flowi()`, returns an SELinux SID (the `sid`) when given a `flowi` and direction. A `flowi` is a small kernel struct which maintains state for a generic Internet flow. The state which interests us includes the input interface (`iif`), output interface (`oif`), and source and destination IP addresses.

By default in `xen-unstable`, the backend drivers do not consistently maintain the `iif` and `oif` all the way through to our authorization function. IP routing further alters the `iif` and `oif`. The variables exist in the `flowi`, but their values are not maintained (presumably to save CPU cycles by not doing the extra copy). We augmented the backend logic and parts of the IP routing logic to properly maintain the `iif` so that it can be used in the authorization logic for outgoing flows (flows going from `domU` to the IPsec tunnel). Note that we did not modify the frontend driver, as this resides in a `domU` and is not part of the TCB for the distributed reference monitor.

In our prototype, the mapping from sHype types to SELinux subject types is configured statically. SELinux subject types have the form *user:role:type*, while sHype types can be arbitrary strings. Since currently we have no type transitions for the types of `domU`s, we use the user `domu_u` and the role `domu_r`. We adopted the convention that we interpret the sHype type label as an SELinux type. For example, an sHype type `green_t` will map to SELinux type `domu_u:domu_r:green_t`.

We added two data structures (linked lists of small `struct`s) to the `dom0` kernel to maintain additional information necessary for policy translation between SELinux and sHype types. The first list maintains metadata for each `domU`: its domain ID, Internet-visible IP address, and backend interface name. The second maintains a mapping between sHype textual labels and their binary equivalents in compiled sHype policy. Both of these lists are manipulated by reading and writing to entries in `/proc/dynsa` (for dynamic security association). Maintenance of the first list (`domU` metadata) is performed automatically by extensions we made to the Xen scripts which start and stop `domU`s. The second list (sHype mapping) is populated whenever the sHype policy is loaded or changed (typically once per boot, although it is possible to change the policy while a system is running).

4.4 Integrity Measurement

We establish trust into the VMM environment by using an existing software attestation approach (IMA [35]) based on the platform's hardware TPM or equivalent functionality on the virtualization platform. We attest to `dom0` by attesting to the system's boot sequence, the hypervisor image, the pre-compiled security policy, and the `dom0` image and `initrd`.

To attest to the software loaded into individual `domU`s, we use virtualized TPMs (vTPM). Each `domU` is associated with an instance of a vTPM that is created when the `domU` is defined and automatically connected to when the `domU` is started.

Attestation based on a vTPM requires us to first establish trust into the environment where the vTPM is running, which is `dom0` in our case. We establish this trust by using the platform's hardware TPM and attesting to the software loaded in `dom0`. After that we attest to the `domU` based on its vTPM instance. The

```
structures, types, & enumerations:
100: $list\_entry\_t \equiv \langle domid, ipaddr, iface\_name \rangle$                 /* domU metadata list entry. */
101: $sid\_t \equiv \langle Integer \rangle$                                              /* SELinux Security ID. */
102: $ssid\_t \equiv \langle Integer \rangle$                                             /* sHype Security ID. */
103: $flowi\_t \equiv \langle src\_ip, dst\_ip, src\_port, dst\_port, iif, oif, ... \rangle$   /* kernel-defined flowi. */
104: $dir\_t \in \{IN, OUT\}$                                                          /* kernel-defined enumeration. */
get_sid_from_flowi(flowi_t fl, dir_t dir):                                        /* return SELinux SID given flow info. */
200: $list\_entry\_t\ e$
201: if $(dir == OUT)$ then
202:     $e =$ find_list_entry_by_iface$(fl.iif)$                                    /* domU to tunnel, search by interface. */
203: else if $(dir == IN)$ then
204:     $e =$ find_list_entry_by_ipaddr$(fl.dst)$                                   /* tunnel to domU, search by IP addr. */
205: end if
206: if $e == NULL$
207:         then $return\ \perp$                                                    /* Fail if no entry found. */
208: $ssid\_t\ ssid =$ hcall_ssid_from_domid$(e.domid)$                  /* get sHype SSID for domain domid via hCall. */
209: $string\_t\ label =$ get_label_from_ssid$(ssid)$                             /* map sHype SSID to string label. */
210: $label =$ "domu_u : domu_r :" $+\ label$                                   /* convert sHype label to SELinux label. */
211: $sid\_t\ sid =$ security_context_to_sid$(label)$                          /* obtain SELinux SID for textual label. */
212: $return\ sid$
```

Figure 3: Pseudocode for the authorization function in our bridging reference monitor. The string `label` is the human-readable type label, which gets converted from an sHype label to an SELinux subject label by prepending `domu_u:domu_r:`. The `security_context_to_sid()` function is part of a normal SELinux installation; the remaining functions are all part of our implementation.

initialization of a vTPM instance includes the insertion of boot measurements. This is usually done by a trusted boot loader, but has to be emulated in the Xen environment.

To securely connect the domU software-root-of-trust (vTPM) back to the platform hardware-root-of-trust (hardware TPM), we connect the measurement lists of the core hypervisor VMM environment (BIOS, boot-loader, hypervisor, security policy, and management domain), which is measured into the hardware TPM, with the vTPM instance measurements that concern a particular domU. This enables the attesting party to first establish trust into the properties of the vTPM and the environment it is running in, and then, based thereupon, establish trust into the domU measured into this vTPM. We have achieved this by dividing the number of TPM Platform Configuration Registers (PCR) into two regions. The lower 8 PCR registers are designated for the vTPM-hosting environment - currently dom0- and reflect the accumulation of boot measurements taken therein. Queries for their values by the domU return the current values of the hardware TPM. Requests for extending their values, however, are rejected, since the registers do not belong to the domU. The upper set of PCR registers, on the other hand, are free for use by the domU and their values can be extended as needed, for example for accumulating measurements of launched applications.

The remote party first validates the vTPM-hosting environment using the hardware TPM measurements. If this environment shows the expected properties, then the domU measurements protected by the vTPM are evaluated. Finally, the attested domU can be assigned properties based on its software run-time environment and configuration.

5 Experiments

We ran a number of experiments to verify the workload isolation and software integrity properties of our MAC bridging prototype. In all these experiments we used the prototype system shown in Figure 2 and described earlier in this section.

We constructed IPsec, SELinux, and sHype policies for `fans` and IPsec and SELinux policies for `boinc` to enable our distributed computing prototype to function. We added types to the SELinux policies on `boinc` and `fans` named for colors, e.g., `red_t`, `green_t`, and `blue_t`. We added similarly named types to the sHype policy on `fans`, and gave `dom0` access to all colors. We created labeled IPsec policies on `boinc` and `fans` based on the static IP addresses of `boinc` and `fans`, and the IP addresses of the `domU`s (i.e., BOINC clients that wish to donate spare CPU cycles) to be created on `fans`.

To be able to verify the software integrity of the client system (`dom0` on `fans`), we built a database of measurements where we collected known hashes, their associated file names as well as individual trustworthiness ratings. We built a similar database for the system hosting the BOINC client (a `domU` on `fans`) and added an entry for a test application that we rated as *untrusted*. We use this application as a trigger to mark its hosting `domU` as *untrusted* once the application is executed. A quote daemon running on each one of the systems returns a signed quote of the current values of PCR registers as well as the list of measurements taken by the Integrity Measurement Architecture. A verification of that list against the corresponding database as well as accumulating all measurements and comparing the resulting values against the reported values allows us to establish a rating for the overall attested system. We periodically run the attestation test to monitor the trustworthiness of the BOINC client `domU` and take appropriate action once it loses its *trusted* state. We can trigger this by simply launching the aforementioned *untrusted* application.

In our experiment we first enabled our MAC bridging extensions in `dom0` and confirmed that the BOINC client `domU` and the BOINC server (`boinc`) could not communicate unless the proper IPsec, SELinux and sHype policies were in place at both endpoints. We make sure that `boinc` and `dom0` on `fans` will not establish the IPsec tunnel until the necessary entries have been added to the IPsec Security Policy Database and the SELinux policy at each endpoint. We also verify that the client system will not forward packets between the IPsec tunnel in `dom0` and the `domU` running the BOINC client until the necessary entries have been added to the SELinux policy in the MAC virtual machine (`dom0`) and the sHype policy in the Xen hypervisor.

In a second step we enabled our integrity measurement extensions and confirmed that the BOINC client `domU` and server `boinc` could not communicate unless the client system had satisfactorily attested its integrity to the server system. For this we ensured that the server system would not establish an IPsec tunnel until it had verified the integrity of the Xen hypervisor and the SELinux software running in the MAC virtual machine. Further, we verified that the server system would not allow the BOINC client to communicate with the BOINC server until it had verified the integrity of the Linux operating system and BOINC client software running on the user virtual machine. Through periodic attestations to the user VM hosting the BOINC client, we constantly monitored the trustworthiness of that system and would tear down any previously established IPsec tunnel if it changed its state from *trusted* to *untrusted*. In that case we would flush the Security Association and Policy Databases using the IPsec setkey tool, thus preventing further communication between the BOINC client `domU` and server `boinc`. This effectively cut off the BOINC client `domU` from either downloading new jobs or trying to submit possibly corrupted results.

6 Discussion and Future Work

In this section we discuss the limitations of our current prototype and briefly mention lessons we learned during its construction. We also point out areas for future exploration.

Retrospective In Section 1, we presented three requirements for distributed reference monitors. We now revisit these requirements in light of our architecture, prototype and experiments.

1. *Distributed tamper-proofness:* Our prototype requires a VM to successfully attest its ability to uphold the security policies relevant for membership in a particular distributed coalition. We perform both bind-time checks and periodic checks – resulting in tamper-responding behavior. The labeled IPsec

tunnel protects the flow of information between members of a distributed coalition.

2. *Distributed mediation* The labeled IPsec tunnel, SELinux policy in the MAC VM, and sHype policy in Xen ensure that all communication involving members of a distributed coalition is subject to the constraints of the distributed reference monitor.
3. *Compatible security policies* Our prototype uses statically configured policy semantics on all members of a distributed coalition. Attestation ensures that the security policies for each member of the distributed coalition are consistent with expected values. A significant amount of work remains in policy development and verification of security goals. First, we must explore attestation policies that dictate reference monitor tamper-responding requirements and mediation abilities. Second, we must examine how distribution of these policies interacts with reference monitoring function. For example, is the risk of information leakage between distributed coalitions greater than acceptable for the reference monitor? Third, we need to examine composition of distributed coalitions and transfer of VMs from one distributed coalition to another.

Minimizing MAC VMs and the Distributed Reference Monitor In our prototype, the size of the MAC VM is on the order of a regular Linux distribution. The quantity of code in this VM violates the code size constraints for reference monitors—a problem which has plagued every commercial reference monitor we know of. The majority of the code even in a minimal Linux installation is extraneous in a MAC VM. The critical components for the MAC VM in a bridging system are (1) the operating system which boots in the VM; (2) the interface with the hypervisor MAC system; (3) the interface with the labeled secure tunnel to other machines in the distributed coalition; (4) the policy for the labeled secure tunnel; (5) the attestation mechanisms in the MAC VM; (6) the attestation policy; and (7) the mechanism for determining policy compatibility (e.g., when joining a distributed coalition). Instead of running a full Linux kernel in the MAC VM, specialized code can be run which drives the network interface over which the secure labeled tunnel connects, and supports the critical components just described. Hypervisors that can assign the responsibility for a particular device to a particular VM (a *device driver* VM) can help to reduce the code size of a MAC VM. We note that such device driver VMs exist in enterprise-grade hypervisors (e.g., IBM's PHYP) and are a planned feature for the next major release of Xen.

Layering Security Policy Our distributed MAC architecture enables layering of security policies. A distributed MAC system is arranged such that the most important security properties are achieved by the lowest-complexity (most assurable) mechanisms. In other words, the bridging system enforces coarse-grained policies. We do not consider *intra*-VM security controls—these remain in the scope of application developers. Thus, a full commercial system cannot be built on distributed MAC alone. We expect that finer-grained, application-specific mechanisms will be in place in the user VMs running on top of distributed MAC. This structure is advantageous since the most security-critical components are also the most robust. Recall our example where the distributed MAC system enables a service provider to host competing enterprises on the same physical platform—a practice which is rare today because of the difficulty of enforcing service-level agreements.

7 Related Work

Virtual Private Networks (VPNs) allow roaming individuals to connect to a geographically constrained network as though they were located within those constraints. Today, IPsec [25] is commonly used in the implementation of VPNs. While VPNs enhance the security of communication across the untrusted Internet, they are founded on the assumption that all users of the network are benign. As the size of organizations increases, this assumption becomes increasingly troublesome.

Kang et al. explore distributed MLS computing in high-assurance environment [20]. The authors combine single-level systems to multi-level distributed environments by using the network pump [21] to safely connect systems of different security levels. Reeds [34] describes the networking of similar machines that are mutually trusted by administration. He outlooks into connecting heterogeneous machines and states the requirements of such interconnected systems to mutually discover each other's TCB, policy, and software implementation properties. These are among the problems we are addressing here.

Lampson's Logic of Authentication [28] defines a general approach for administering authorizations in a distributed system based on discretionary management of access and delegation statements. Trust management approaches add programs that compute authorizations to certificates to enable more general delegations [8, 12, 29, 30]. These approaches are also discretionary, and the flexibility of delegation presents challenges for MAC information flow control.

Bauer et al. present a scheme for distributed proving in access control systems [5]. Distributed proving schemes such as this are effectively an extension of discretionary access controls across machines. They do not scale as well as mandatory access controls, since policy size increases with the number of subjects. With bridging, the policy size can remain the same even as the number of systems in the distributed coalition grows.

Seshadri et al. propose Pioneer [39], a system for achieving run-time attestation of code executing on a particular hardware platform. This work is currently preliminary, but the approach shows promise as an alternative for TPM-based attestation. This work is complementary to our distributed MAC system, which could leverage run-time attestation as well as TPM-based attestation.

XenoServers is a distributed computation effort that served as the original motivation for the development of Xen [27]. The XenoServers effort is still under development and does not yet enjoy widespread use.

Globus is an open architecture for grid computing [14]. Globus has been designed with attention to security, concentrating on using a certification authority for a particular project that can issue certificates for all participants [48]. The security design for Globus assumes that Globus is run on dedicated, administered machines. Globus is not designed to be securely run alongside commodity applications (e.g., untrustworthy downloads from the Internet).

8 Conclusions

We developed a distributed systems architecture in which MAC policies can be enforced across physically separate systems, thereby *bridging* the reference monitor between those systems. The major insights are that attestation can serve as a basis for extending trust to remote reference monitors and that it is actually possible to obtain effective reference monitor guarantees from a distributed reference monitor. This work provides a basic mechanism and guarantees for building a distributed reference monitor for a BOINC system. In addition, the architecture also enables exploration of MAC, secure communication, and attestation policies and the construction of reference monitors from a set of open source components. As the community gains experience with MAC bridging and new architectural features become available (e.g., TPMs [46], Intel LT [16], and Intel VT [17]), the quantity of code in the bridged TCB can be further reduced. Our bridging architecture enables security policy to be layered based on its complexity, from coarse-grained hypervisor-level policy up to sophisticated application-level policy.

References

[1] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the Workshop on Grid Computing*, November 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1972.

[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 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[5] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[6] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS wide area security architecture. In *Proceedings of the USENIX Security Symposium*, January 1998.

[7] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical report, MITRE MTR-2997, March 1976.

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust-management system, version 2. IETF RFC 2704, September 1999.

[9] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1989.

[10] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[11] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMWare, Inc., October 1998. US Patent No. 6397242.

[12] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, , and T. Ylonen. Spki certificate theory. IETF RFC 2693, September 1999.

[13] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings og the National Computer Conference*, pages 329–334, 1979.

[14] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3), 2001.

[15] D. Harkins and D. Carrel. The internet key exchange (IKE). IETF RFC 2409, November 1998.

[16] Intel Corporation. LaGrande technology architectural overview. Technical Report 252491-001, September 2003.

[17] Intel Corporation. Intel virtualization technology specification for the IA-32 Intel architecture. Technical Report C97063-002, April 2005.

[18] Trent R. Jaeger, Serge Hallyn, and Joy Latten. Leveraging IPSec for mandatory access control of linux network communications. Technical Report RC23642 (W0506-109), IBM, June 2005.

[19] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.

[20] M. H. Kang, J. N. Froscher, and I. S. Moskowitz. An Architecture for Multilevel Secure Interoperability. *Proceedings of the 13th Annual Computer Security Applications Conference, San Diego, CA*, 1997.

[21] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network Pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, 1996.

[22] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[23] S. Kent. Security options for the internet protocol. IETF RFC 1108.

[24] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). IETF RFC 2406, November 1998.

[25] S. Kent and R. Atkinson. Security architecure for the internet protocol. IETF RFC 2401, November 1998.

[26] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[27] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris. Global-scale service deployment in the XenoServer platform. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[28] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[29] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, February 2003.

[30] Ninghui Li and John C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 89–103, June 2003.

[31] Hiroshi Maruyama, Frank Seliger, Nataraj Nagaratnam, Tim Ebringer, Seiji Munetoh, Sachiko Yoshihama, and Taiga Nakamura. Trusted platform on demand. Technical Report RT0564, IBM, February 2004.

[32] M. McGrath. Virtual machine computing in an engineering environment. *IBM Systems Journal*, 11(2), 1972.

[33] David Molnar. The SETI@Home problem. ACM Crossroads 7.1, Available at: `http://www.acm.org/crossroads/columns/onpatrol/september2000.html`, September 2000.

[34] Jim Reeds. Secure IX network. In *Cryptography and Distributed Computing*, Series in Discrete Mathematics and Theoretical Computer Science. AMS/ACM, 1991.

[35] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

[36] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[37] R. Schell, T. Tao, and M. Heckman. Designing the GEMSOS security kernel for security and performance. In *Proceedings of the National Computer Security Conference*, 1985.

[38] M. D. Schroeder, D. D. Clark, J. H. Saltzer, and D. Wells. Final report of the MULTICS kernel design project. Technical Report MIT-LCS-TR-196, MIT, March 1978.

[39] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, October 2005.

[40] SETI@Home: Search for extra-terrestrial intelligence. `http://setiathome.ssl.berkeley.edu/`, July 2005.

[41] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[42] Stephen Smalley and Peter Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[43] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.

[44] Sean W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, October 2002.

[45] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the USENIX Security Symposium*, 1999.

[46] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands, October 2003. Version 1.2, Revision 62.

[47] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[48] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *Proceedings of Symposium on High Performance Distributed Computing (HDPC)*, June 2003.

[49] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. The linux security module framework. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[50] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the USENIX Security Symposium*, 2002.

[51] Heng Yin and Haining Wang. Building an application-aware ipsec policy system. In *Proceedings of the USENIX Security Symposium*, 2005.

[52] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the USENIX Security Symposium*, August 2002.