RC23865 (W0602-038) February 2, 2006
Computer Science

IBM Research Report

DeuTeRiuM - A System for Distributed Mandatory Access Control

Jonathan M. McCune[1], Stefan Berger[2], Ramón Cáceres[2], Trent Jaeger[3], Reiner Sailer[2]

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

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

[3]Pennsylvania State University
University Park, PA 16802

DeuTeRiuM – A System for Distributed Mandatory Access Control

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, trusted reference monitor* (DTRM) that enforces mandatory access control (MAC) policies 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) virtual machines to simplify reference monitor design and the MAC policies enforced; and (3) IPsec with MAC labels to ensure the protection and authorization of commands across machines. 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 whose reference monitor design requires only 13 authorization checks, only 5 of which apply to normal processing (others are for policy setup). This prototype enforces MAC between machines using IPsec extensions that label secure communication channels. We show that, through our architecture, distributed computations can be protected and controlled coherently across all the machines involved in the computation.

1 Introduction

Mandatory access control is becoming a common feature in commercial operating systems, such as Linux, IBM AIX, and FreeBSD, and has been present in Trusted Solaris for several years. *Mandatory access control* (MAC) mandates that security policies be defined and managed by the system rather than the individual users or their programs. A system enforces mandatory access control using a *reference monitor* [4]. Since a reference monitor must be capable of protecting the system from all programs and of enforcing MAC for the entire system, reference monitor implementations must be tamperproof and must provide complete mediation of all security-sensitive operations. MAC is becoming popular because of its ability to confine processes (e.g., decomposing *root*) and its potential to enforce security guarantees (e.g., secrecy and integrity). For example, MAC can ensure that no information flows leak secret information to subjects not authorized for that information. Also, MAC can ensure that low-integrity data is not input to high-integrity processes.

Naturally, we want such security guarantees to span multiple systems. For example, NetTop separates computing of different secrecy levels into their own isolated virtual machines (VMs) running on Linux [27]. NetTop uses Linux MAC enforcement of the SELinux [34, 35] reference monitor to provide isolation of the virtualization function from traditional system services. Further, SELinux must control network communication by NetTop VMs to ensure isolation of VMs of the same level. However, a compromise of SELinux security state could result in mislabeling of data and communications that would violate the NetTop policy.

[*]`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

NetTop and the other OS-based MAC approaches enable MAC enforcement across a set of machines, but they have two limitations that we seek to address: (1) the complexity of the reference monitor, the code that must be trusted to enforce distributed MAC, is high and (2) trust in the necessary enforcement function of the reference monitor across each machine in the distributed system is not established. First, these approaches depend on complete mediation of operating system resources (e.g., files, sockets, IPC) which involves hundreds of reference monitor calls. For example, the SELinux reference monitor for Linux implements over 150 distinct kernel requests to authorize access. While NetTop is built on a virtual machine system, it still depends on the host OS and the correctness of its reference monitor. The virtual machine monitor (VMM) provides isolation, but it does not enforce a security policy. Second, while current approaches enable the transmission of MAC context (e.g., security labels) to reference monitors on other machines, they assume trust in each individual machine's ability to enforce such requirements. There is no guarantee that each machine is really running a trusted computing base that enforces the intended MAC policy. This limits MAC enforcement to environments where homogeneous control of systems is possible, and even then, a misconfiguration or compromise may go undetected.

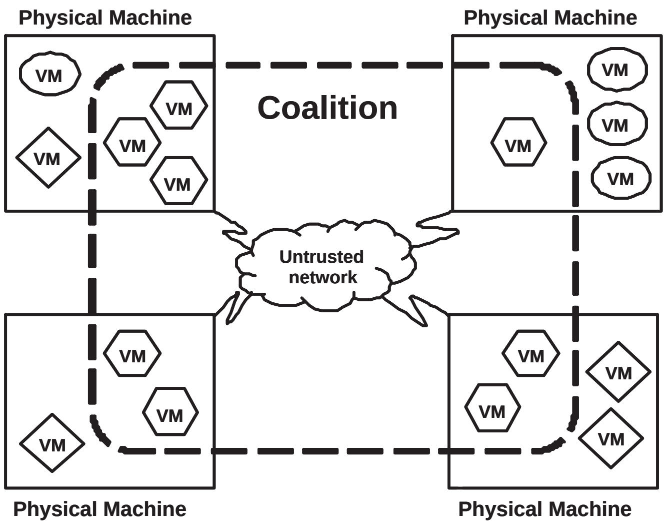

Figure 1: *Example of a distributed coalition. Virtual machine (VM) instances sharing common Mandatory Access Control (MAC) labels on multiple physical hypervisor systems are all members of the same coalition.*

Figure 1 illustrates our high-level goal. A distributed application consists of a *coalition* of virtual machines that execute across a distributed system of physical machines. Each of the coalition VMs may reside on a different physical machine and multiple coalitions may execute on each physical machine. The physical machines each have a reference monitor capable of enforcing MAC policies over all its VMs. However, to the individual coalitions, the combination of reference monitors forms a coherent, uniform unit that protects the coalition from other coalitions and limits the actions of the coalition VMs. As VMs are added or migrate to new machines, the coherent, uniform reference monitoring unit is verified to ensure its trustworthiness. We call this combination of reference monitors, whose mutual trust can be verified, a *distributed, trusted reference monitor*, or DTRM.

In this paper, we introduce a DTRM approach for MAC enforcement across distributed systems that requires a very small amount of reference monitoring function on each machine, thus enabling trust in this function to be verifiable over the entire distributed system. MAC enforcement is simplified by using a small virtual machine monitor as the base code and relying on minimal operating system controls. The Xen hypervisor system is our VMM, and we only depend on it for inter-VM controls which are available through only two Xen mechanisms: grant tables (shared memory), and event channels (synchronous channels). Xen provides system services (such as hardware and guest virtual machine controls) through a single trusted VM that runs a complete operating system (Linux) at present. However, we find that MAC enforcement only requires that the trusted VM control network communication. Only the SELinux controls for IPsec and packet processing (seven hooks) are needed for MAC enforcement. As a result, the enforcement of only 13 total authorizations (combined from Xen and SELinux) are needed from the reference monitors.

Trust in the MAC enforcement capabilities of a remote system is developed using remote attestation [28, 36]. We use remote attestation to enable each machine to verify the following properties of the reference monitoring infrastructure: tamperproofing (i.e., code and communication integrity), mediation (e.g., effective MAC enforcement mechanisms), and the satisfaction of security goals (e.g., isolation from other workloads) in a distributed environment. We can extend this trust up to the target VM (i.e., the VM that provides application services) through attestation as well. While this may sound conceptually similar to previous work on VM attestation in Terra [12], this work differs in that Terra only attests VM code and static data integrity, does not offer MAC enforcement, and does not attest to any enforcement properties.

The contributions in this work are:

1. a system built from open-source software components that enables enforcement of MAC policies across a set of machines;
2. complete MAC reference monitoring from two software layers, (1) the Xen hypervisor that controls inter-VM resource accesses, and (2) SELinux and IPsec network controls; and
3. the use of attestation to build trust in the reference monitoring across all machines in a distributed system.

We demonstrate this implementation by applying it to a BOINC distributed computing application [2]. The BOINC infrastructure enables distributed computations by a group of clients coordinated by a server, such as the SETI@home volunteer distributed computing effort [3]. On our system, we run a BOINC server and its clients in VMs. The reference monitors of each of the machines hosting the BOINC VMs perform a mutual verification of acceptable reference monitoring software and MAC policy. Then, each of these reference monitors enforce the isolation of the BOINC VMs from others and protect other coalitions from the BOINC VMs. We describe how the DTRM approach enables verification of trust and MAC-enforced isolation. We note that other MAC policies may be enforced on the BOINC system, depending on the trust in the user-level VMs, which can also be established via remote attestation.

The rest of this paper is organized as follows. Section 2 provides background motivation for the problem of building a distributed, trusted reference monitor. Section 3 presents the architecture of our DTRM, 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

Other researchers have developed systems that meet some of the necessary requirements for a distributed MAC system. However, we find that existing systems are each insufficient along at least one of three axes: software complexity, policy complexity, and trust establishment.

Software Complexity As Figure 2 illustrates, a prohibitively large number of operating system hooks are required to mediate security-sensitive operations in operating system MAC systems. This level of code complexity greatly reduces the usefulness of MAC policy and attestation mechanisms, as the policies must be hardware- and operating system-dependent to be effective. For example, the reference monitor must mediate all possible access paths to all system objects (e.g., files, sockets, IPC, shared memory, packets, etc.) defined by the operating system. The result is a large reference monitor code base. Also, since operating system unify different physical objects into files, the subtle semantic distinctions between different objects (e.g., socket, device, and regular files) complicate reference monitoring.

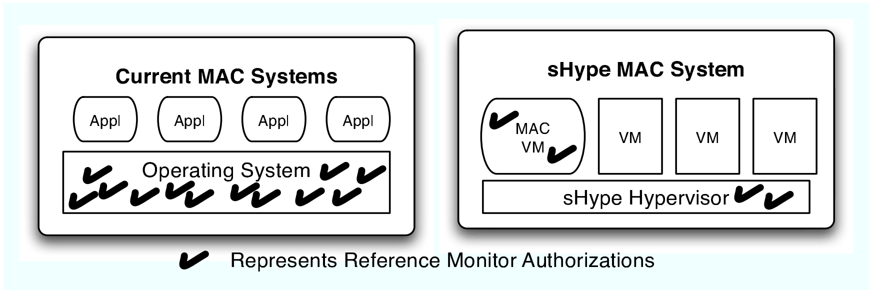

Figure 2: *Current Mandatory Access Control systems use a prohibitively large number of operating system hooks (on the order of hundreds). MAC policies for these systems depend on details of the particular system, making enforcement across a distributed system difficult. By comparison, our system leverages virtualization so that MAC policies can be largely system-independent, resulting in significantly fewer required mediation points. sHype is a hypervisor security architecture developed by IBM Research for different virtual machine monitors [31].*

An example of an operating system with MAC capabilities that suffers from these shortcomings is Trusted Solaris [37]. Trusted Solaris consists of extensions to the Solaris operating system to provide multi-layer security (MLS) labels. Subjects and objects (e.g., processes, files) are labeled with secrecy levels, and communication is also labeled using IP Security Options. Since the operating system is the lowest level of MAC, a prohibitive number of instrumentation points (hooks) are required to enforce MAC policy. This complexity precludes formal analysis, and thus the robustness of the resulting system cannot be guaranteed.

We consider an alternative: the use of a VMM layer which supports coarse-grained MAC enforcement. VMMs exist today which are several orders of magnitude smaller than commodity operating systems. As shown in Figure 2, we find that some operating system-style MAC controls are required to control inter-VM communication, but these are minimal (e.g., seven authorization hooks).

Policy Complexity A further advantage gained from the use of virtualization technology is reduced policy complexity. A VMM-level MAC policy is naturally more coarse-grained than that of an operating system. For example, in the sHype hypervisor[1] security architecture [31], a VMM-level MAC system considers VMs as subjects and only inter-VM communications as objects. Also, problems where unnecessary sharing is present in current systems can be remedied by using separate VMs. As a result, simpler policies with real separation may enable formal verification of security goals from MAC policies, which has been difficult for OS-level policies. We note that 8 of the 13 authorization hooks for the VMM-level MAC system are for policy administration, for which few subjects are authorized.

NetTop [27] uses a virtual machine monitor (VMWare) and operating system with MAC support (SELinux)

[1]The difference between a Virtual Machine Monitor and a Hypervisor is a hotly debated topic. We use the terms interchangeably, referring to a thin software layer that runs directly on top of the hardware and presents a virtualized image of that hardware to conventional operating systems running above.

to enable what were traditionally physically separate computer terminals on the desks of government employees to be consolidated onto a single system. However, NetTop does not consider a VMM with support for MAC. The NetTop architecture relies extensively on operating system security controls, which we have already shown to suffer from excessive complexity. We conclude that NetTop-style systems may have value if all the systems are under the same administrative domain, but they are not viable in heterogeneous distributed systems.

Trust Establishment To achieve a secure distributed system, mechanisms are required for establishing trust into a remote system. While much prior work has been done on attestation, complexity of software and policy have rendered attestations less meaningful than desired on existing systems. The goal here is to ensure that each reference monitor is of high integrity, is capable of enforcing the desired MAC policy, and is really using that policy. The use of the VMM-based approach means that attestation of the VMM itself and any privileged supervisor VMs is necessary to verify reference monitor integrity and enforcement (e.g., authorization hooks). These components should be smaller and more stable than operating system-only MAC systems. Consider the the complexity of MAC policies in current OS systems and the large number of system drivers. The simplifications offered by the VMM MAC approach reduce the complexity of MAC policy through simpler subject (e.g., VM) and object (e.g., inter-VM communications) labeling, and through simpler subject rights. Also, the MAC VM can be stabilized by focusing on inter-VM control function and a small number of drivers.

Terra [12] introduces an architecture which uses VMM technology for isolation and includes attestation support. Today, the Xen hypervisor system [5] with Trusted Platform Module (TPM) support, which we leverage in our solution, achieves similar properties to those of Terra. However, Terra itself is an incomplete solution, because no controls exist for enforcing a MAC policy.

We have designed a distributed trusted reference monitor (DTRM) that facilitates all three of the requirements described above. The DTRM approach builds trust in layers, starting from the bottom, such as from trusted hardware like the Trusted Computing Group's Trusted Platform Module (TPM). After the BIOS and boot firmware, the bottom-most software layer is a VMM which is capable of enforcing a coarse-grained (hence low complexity) MAC policy regarding information flows between isolated VMs. The VMM codebase is substantially smaller than that of a host OS (tens of thousands of lines of code, as opposed to millions, using Xen and Linux as examples), predicating formal verification for assurance. The MAC VM and MAC policy attestation complete the establishment of DTRM trust, and we aim to show that the complexity required of these components can still be a significant improvement over host OS MAC. Note that we have not formally verified the implementation that we describe later in the paper, but that our architecture lends itself to making the most security-critical components as small as possible, thereby helping to alleviate security-relevant dependencies on components of excessive complexity.

The resulting system is shown conceptually in Figure 3; the entire distributed system functions as if there is one reference monitor, which enforces the necessary policy on all members of the distributed system. In order to build a reference monitor across machines, we must enable verification of its tamperproof protections and its mediation abilities, and that verification of correctness of its implementation and MAC policies is practical. We discuss these goals in light of our approach and implementation in Section 6.

3 System Architecture

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

The goal of our architecture is to enable the creation of distributed coalitions of VMs, as shown earlier in Figure 1. Sailer et al. define a *coalition* as a set of one or more *user VMs* that share a common policy

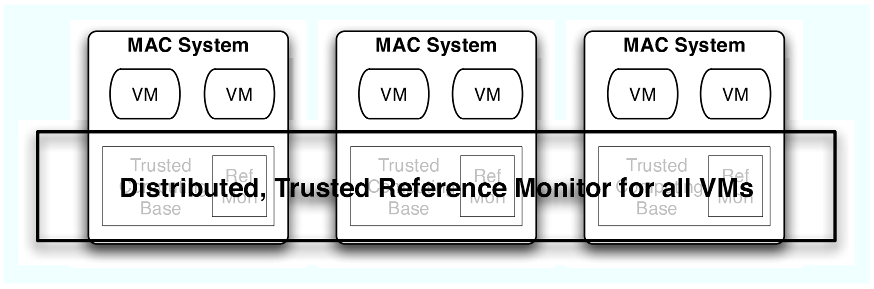

Figure 3: *The DTRM approach results in a conceptually singular reference monitor which is shared across all machines in the distributed system. Individual machines have assurance that other machines are enforcing the desired MAC policy.*

and are running on a single hypervisor system with MAC [31]. We extend the definition of a coalition to include VMs on physically separate hypervisor systems which share a common MAC policy. The resulting distributed coalition has a MAC policy enforced by a *distributed, trusted reference monitor* (DTRM). We begin by providing a high-level overview of our architecture. We then present the specifics of the hypervisor MAC architecture we use (Section 3.2). Next, the process of extending the DTRM is presented, thereby establishing a *bridge* between two systems (Section 3.3). We also discuss some limitations of our design (Section 3.4).

3.1 Architecture Overview

As depicted in Figure 4, our architecture consists of the following concepts: (1) *user VMs* implement application function; (2) *coalitions* consist of a set of user VMs implementing a distributed application; (3) *distributed, trusted reference monitors* (DTRM) consist of a combination of reference monitors for all machines running user VMs in a single coalition; (3) *common MAC policies* define MAC policies for a single DTRM; (4) *hypervisors* are VMMs that run on a single physical machine and enforce the common MAC policy for local communications on that machine; (5) *MAC VMs* enforce the common MAC policy on inter-VM communications across machines; and (6) *secure, MAC-labeled tunnels* provide integrity protected communication which is also labeled for common MAC policy enforcement.

User VMs represent application processing units. Typically, a user VM will belong to one coalition and inherit its label from that coalition. For example, a set of user VMs that may communicate among themselves, but are isolated from all other user VMs, would form a coalition. Each user VM runs under the same MAC label, and all 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 [6].

Special user VMs may be trusted to belong to multiple coalitions, such as the MAC VM that is accessible to all coalitions. These have a distinct label that conveys rights in the common MAC policy to access multiple coalitions.

A coalition's reference monitor is a distributed, trusted reference monitor (DTRM). It consists of the union of the reference monitors for the physical machines upon which coalition's user VMs run.

The common MAC policy is the union of the MAC policies of the reference monitors in a coalition's DTRM. The common MAC policy must ensure MAC properties (e.g., isolation) of its coalition in the context of the other user VMs from other coalitions that may also be present on the DTRM's physical machines.

The hypervisor and MAC VM comprise the reference monitoring components on a single physical

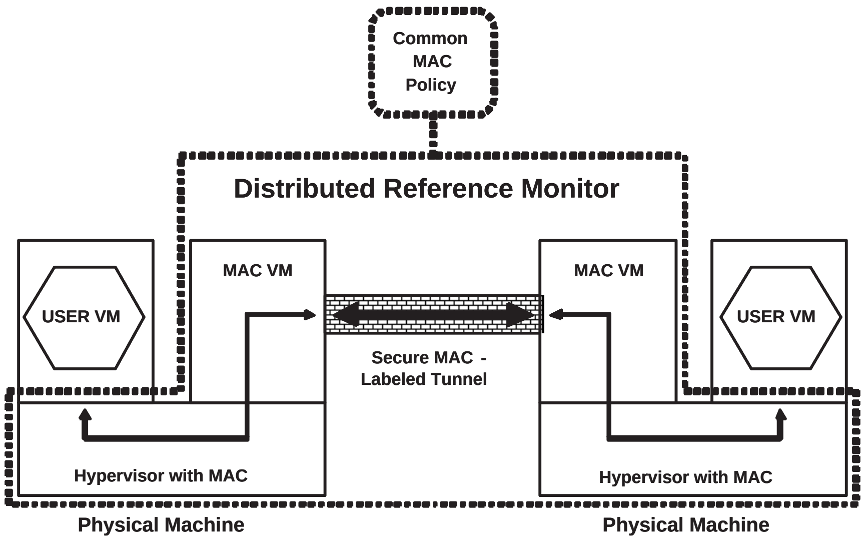

Figure 4: *Example of a distributed, trusted reference monitor (DTRM).*

machine. The hypervisor controls user VM function local to that machine, and the MAC VM enforces inter-machine communications. That is, we define that inter-machine communications are implemented only in the MAC VM. Both enforce controls based on the common MAC policy.

Inter-machine communication is implemented via secure, MAC-labeled communication tunnels. The DTRM constructs secure communication tunnels between physical machines 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 DTRM can control which user VMs can use which tunnels.

The combination of above concepts forms a distributed, trusted reference monitor system. The architecture must enable composing and extending DTRMs as new machines join, an act that we call *bridging*. The key step is the establishment of trust in the resultant DTRM.

3.2 sHype MAC Hypervisor

The foundation of a DTRM is sHype, a hypervisor security architecture developed by IBM Research for different virtual machine monitors [31]. Building on emerging and broadly available hardware and software support for virtualization, sHype provides simple, system-independent and robust security policies and enforcement guarantees within the boundaries of a single VMM. sHype deploys mandatory access control policies enforced independently of the controlled virtual machines. It offers two policy components: a Simple Type Enforcement policy (STE) that controls the sharing between different VMs, and a Chinese Wall policy (CHWALL) that controls which VMs can run simultaneously on the same system.

The STE policy component controls sharing between virtual machines by controlling access of virtual machines to inter-VM communication and to any virtual resources such as virtual block devices and virtual network devices, through which VMs can share information indirectly. Conceptually, the STE policy creates coalitions of VMs and assigns VM and resource memberships to coalitions. Treating both VMs and virtualized hardware resources equally as generic resources, access control decisions using STE are based on common coalition membership. This is stated formally below, where res stands for $resource$:

(1) $access\_allowed(res1, res2) = \exists coalition : member(res1, coalition) \wedge member(res2, coalition)$

(2) $member(res, coalition) = coalition \in ste\_label(res)$

The CHWALL policy component controls which VMs can share a physical machine at any time. It is designed to approximate an air-gap between conflicting workloads so that such workloads, running in different virtual machines but at different times, cannot affect each other even in the absence of strong resource control or in the presence of residue covert channels. The CHWALL policy defines *conflict_sets* that consist of a set of CHWALL types (usually a CHWALL type refers to a certain data set used by a workload). Every VM is assigned a CHWALL label consisting of the set of CHWALL types for this VM. Let *running_types* be the managed set of CHWALL types of all labels of currently running VMs. Then starting a VM A is decided as follows:

(3) $start\_allowed(A) = \forall type \in chwall\_label(A), \forall conflict\_setsCS :$
$type \in CS \rightarrow \forall cstype \in CS, cstype \neq type : cstype \notin running\_types$

The sHype CHWALL component implements the simple security property of the formal Chinese Wall security model defined by Brewer and Nash [8]. The *-property can be approximated by properly configuring the STE policy component by aligning the STE and CHWALL type definitions. For example, a financial analyst being prevented from knowing conflicting information of different companies in the Brewer/Nash model corresponds to the hypervisor in sHype being prevented from multiplexing access to a single shared hardware device between conflicting workload types.

3.3 Setting up a Bridge

When a user VM of a system joins a coalition, its reference monitor (components of the VMM and MAC VM on the joining system) bridges with the coalition's DTRM. In our implementation, a reference monitor that is already a coalition member serves as a representative for the coalition. 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 joining reference monitor (JRM) and the DTRM mutually verify that their tamper-responding and mediating abilities are sufficient for the bridging; (3) the new user VM is initialized; and (4) the secure, MAC-labeled network communication of the bridge is enabled. Each of the four stages of the bridging process are now described in detail.

Stage 1: Establish Common MAC Policy A new reference monitor joining the coalition will affect MAC policy in two ways: (1) the joining reference monitor will add the coalition label and its rights to its local MAC policy and (2) the DTRM common MAC policy will become the union of the JRM and former DTRM MAC policies. First, the JRM must verify that the resultant coalition policy is compatible with the current policy (e.g., does not violate isolation guarantees of the coalition). Second, the resultant DTRM policy now includes that of the JRM to ensure that overall DTRM security goals can be enforced.

We present two different ways that the joining reference monitor (JRM) can obtain a coalition's common MAC policy. First, the JRM may have its own MAC policy and a means for translating coalition MAC policy to its labels. This is necessary because the semantics of a particular label (e.g., *green*) in the JRM's existing configuration may map to those of another label (e.g., *blue*) in the distributed coalition. In a coalition that uses a single label, the label name may be translated to one the JRM understands. Using simple name translation, coalitions may 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.

A second option is to have the distributed reference monitor push a configuration to the JRM and have the JRM 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 first approach, so the MAC policy is fixed at the hypervisor level and coalition policies are mapped to it.

Stage 2: Confirm Tamper-Responding and Mediating Abilities An attestation policy is used to mutually verify JRM and DTRM tamper-responding and mediation abilities. 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, and the security goals that will be enforced via the MAC policy. Our prototype, which we describe in Section 4, attests to the Xen hypervisor code, MAC VM code, and the MAC policy.

Stage 3: Initialize User VM The code to be executed inside the user VM is assigned a MAC label based on attestation of the code. In the BOINC example, the user VM is already present and labeled (e.g., *green*), so initialization is trivial. The BOINC server may want attestation that the BOINC client was started as expected. In that case, attestation may be applied at the user VM level to prove to the BOINC server which code was used. An additional optimization, which we discuss in Section 6, is to have the BOINC server provide the code for the entire user VM (i.e., the OS image as well as the BOINC client software).

Stage 4: Secure, Labeled Communication 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 keyed-hash 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, a *green* user VM may have access to *green* tunnels and only to *green* tunnels, so an isolated coalition can be constructed. Our prototype uses the MAC-labeled Linux IPsec implementation in the MAC VM to construct and control access to tunnels for user VMs.

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. If such protection is needed, attestation needs to obtain appropriate guarantees (e.g., from a TPM in a location that assures such protections).

Initialization While discovering that an initial value is wrong is easy (e.g., Tripwire [23]), 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 [9]. Attestation enables verification of the initial state of code and static data, but not for mutable data.

Runtime Tamper-Responsiveness TPM-based attestation mechanisms (e.g., the Integrity Measurement Architecture (IMA) of Sailer et al. [30]) measure inputs at load-time. Thus, runtime tampering may go undetected. Other techniques, such as Copilot [18] and BIND [33], aim to provide some runtime guarantees in addition to load-time guarantees, 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. However, the code loaded and attested can safely be related to known vulnerabilities. It is here that minimizing code and policy complexity can pay off. The lower the complexity, the stronger the guarantees that can be provided via attestation. Ideally, everything would be of sufficiently low complexity to enable formal verification for assurance.

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 sHype hypervisor MAC policy enables by use of conflict sets of the Chinese Wall policy to formally define which coalitions cannot run at the same time on the same hypervisor system [31].

4 Implementation

We implemented a DTRM for volunteer distributed computation according to the design presented in the previous section. This section describes our implementation in detail. It starts with a description of the hardware and software configuration of our prototype. It continues with descriptions of how we implemented secure, MAC-labeled tunnels for network communication; type mapping and MAC enforcement for the reference monitor; and integrity measurement for attestation.

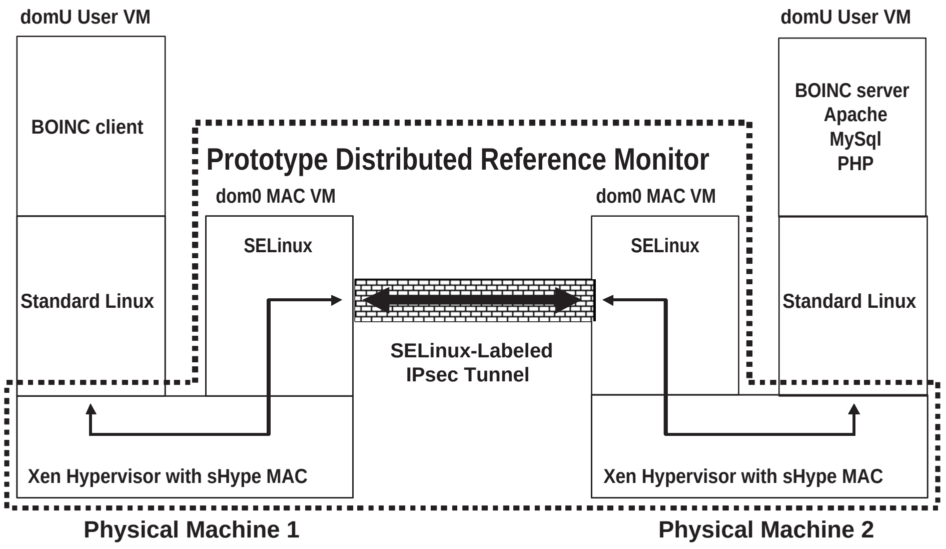

Figure 5: *Bridging the Distributed, Trusted Reference Monitor (DTRM) in our distributed computing prototype.*

4.1 Machine Configuration

We configured two hypervisor systems running Xen [5] with sHype [31]. One system runs a dedicated BOINC [2] server inside a non-privileged user VM. The other system runs one or more BOINC clients, each in its own user VM. The supervisor domain in each Xen system runs Fedora Core 4 with SELinux [35] configured in *strict* mode. These supervisor domains serve as the MAC VMs and perform the necessary policy translations from SELinux labels on an IPsec tunnel [21, 22] to local sHype labels. Our implementation is based on a Simple Type Enforcement (STE) policy, where Xen VMs can share resources and data only if they have been assigned a common STE-type. Figure 5 shows the structure of our prototype.

We used two machines in our experiments, shype1 and shype2. Each has a 2.4 GHz Pentium IV with 1 GB of RAM and a 512KB cache. We used shype1 as the BOINC client and shype2 as the BOINC server. We will provide more details as appropriate, referencing the machines by name for convenience.

Device Driver and MAC VMs on Xen

We built and maintain our DTRM prototype 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 one device driver VM – dom0.

Our configuration 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. As we will show, the SELinux policy needed on dom0 is significantly smaller than an SELinux policy for a typical Linux distribution, as it deals primarily with networking controls.

User VMs on Xen The domU on shype2 runs Fedora Core 4 and consists of installations of Apache, MySQL, PHP, and the BOINC server software. The BOINC server issues compute jobs to clients, collects and tabulates results, and makes status information available via the website it hosts.

The domU on shype1 runs Fedora Core 4 and the BOINC client software. The BOINC client accepts compute jobs from the BOINC server, runs them, and returns the results.

4.2 Labeled IPsec Tunnels

We use labeled IPsec connections operating in tunnel-mode [22] as the secure communication mechanism between the dom0s (MAC VMs) on shype1 and shype2. We first describe the role of the labeled tunnels in the distributed MAC system, and then describe their implementation. We describe the processing of packets arriving at a dom0 from a remote system and destined for a local domU; processing is symmetric in the opposite direction, when packets arrive at a dom0 from a local domU and destined for a remote system.

Packets arrive in dom0 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 silently 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 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 5, reference monitor functionality exists in both the endpoint of the IPsec tunnel (*OS type check* in dom0) and in the hypervisor (*hypervisor type check* in 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. [17]. 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) [13] 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 key authorization controls which IPsec policies, and hence labels for resultant IPsec security associations, may be authorized to a subject (i.e., a user VM). There are six other hooks used: four authorize allocation and deallocation of IPsec policies and security associations, and two filter incoming and outgoing packets.

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 the DTRM of 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 [40].

4.3 Type Mapping and Enforcement

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 that help to ensure that machines in a distributed coalition enforce semantically equivalent sHype policies. To achieve this goal, we must translate between SELinux subject types and sHype types.

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 (in the SELinux sense) for the types of domUs, 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 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. Device drivers for physical network interfaces reside entirely in dom0, so that packets to and from physical networks always leave and enter the platform via dom0.

Our authorization function (see pseudocode in Figure 6, and a summary of total code changes in Figure 7), 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.

One of our design goals is to minimize per-packet overhead. We decided to use the iif and oif to keep track of packet origin and destination while packets traverse the IP routing logic inside the kernel. By default in xen-unstable, the iif and oif are not maintained all the way through to our authorization function, though space for the variables exists in the flowi. We added one line of code to net/ipv4/xfrm4_policy.c in the dom0 kernel to maintain the iif (e.g., vif1.0) on outgoing flows (packets travelling from domU to the IPsec tunnel) so that the iif can be used in the authorization logic for outgoing flows. Note that this iif value may be incorrect if a packet traverses the routing logic multiple times (through, e.g., multiple kernel-only virtual interfaces within dom0, which are not visible outside of dom0). Since our prototype system does not use multiple layers of routing logic, the iif value is always correct.

Though we would like to make a corresponding change for packets travelling from the IPsec tunnel to some domU in order to authorize packet delivery to that domU, the oif is not known until after the packets have been routed. We would like to keep the label of the tunnel on which a packet arrived with that packet until it reaches our authorization function. However, we decided against adding additional memory requirements for all packets. Maintaining the `iif` on packets arriving on the physical interface for the IPsec tunnel will not be helpful because traffic for more than one `domU` may travel through a particular physical interface. Hence, we currently use the destination IP address of a tunneled packet to lookup the destination interface of a `domU` from a small struct that we added to the kernel, described below. Note that the integrity of the destination IP address is maintained by the IPsec tunnel.

We added two data structures (linked lists of small `struct`s) to the `dom0` kernel to maintain the 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).

```
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 6: *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.*

File	LOC	Purpose
dynsa.c	814	get_sid_from_flowi(), /proc entries, linked lists
xfrm.c	10	calls get_sid_from_flowi()
xfrm.h	1	get_sid_from_flowi() prototype
privcmd.c	35	hcall_ssid_from_domid()
privcmd.h	1	hcall_ssid_from_domid() prototype
xfrm4_policy.c	1	maintain iif
xfrm6_policy.c	1	maintain iif
vif-route	8	update /proc entries on domain create / destroy

Figure 7: *Size of code changes for our DTRM implementation.* `dynsa.c` *is the only file that we created; all other entries in the table refer to modifications to existing files. LOC stands for Lines of Code.*

4.4 Integrity Measurement

We establish trust into the individual systems that form a distributed MAC system by determining that each system is running software that forms an acceptable reference monitor enforcing the required security properties, that each system has been configured with a MAC policy whereby the common MAC policy protects the coalition, and that the software and policy have not been tampered with. To this end we use remote attestation based on the Trusted Platform Module (TPM).

The most important requirement is to establish trust into the parts of each system that make up the DTRM. Recall from Figure 5 that the DTRM comprises the Xen hypervisor and MAC VM (i.e., dom0) on all systems that join a coalition. We attest to the integrity of these components by inspecting measurements of the system BIOS and boot loader, the Xen hypervisor image and its MAC policy, as well as dom0's SELinux kernel image, its initial RAM disk and its MAC policy. Currently, we do not attest to the IPsec configuration programs racoon and setkey because of an SELinux limitation having to do with stacking multiple LSM modules in the same kernel. However, we expect that this limitation will be lifted in future Linux versions, and we plan to extend attestation to application-level software in dom0 at that time.

We also provide a way to establish trust into the user VMs running on top of the DTRM (i.e., domUs). We attest to the integrity of a domU by inspecting measurements of its Linux kernel image and its initial RAM disk, as well as application binaries loaded in that virtual machine. For the BOINC client and server, this involves measurement of their binaries.

We use a virtual TPM (vTPM) facility, which is already a part of xen-unstable, to report measurements of software loaded into domUs. This facility is necessary to make TPM functionality available to all virtual machines running on a platform. It creates multiple vTPM instances that each emulates the full functionality of a hardware TPM, and multiplexes requests as needed to the single physical TPM on the platform. Each domU is associated with a vTPM instance that is automatically created and connected to the domU when that virtual machine is created.

We have achieved this by dividing the number of TPM Platform Configuration Registers (PCR) into two regions. The lower 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. This connection of independent measurement lists is valid since the party that connects the hardware TPM with the software TPM measurements is fully attested by the hardware TPM.

5 Experiments

We ran a number of experiments to verify the workload isolation and software integrity properties of our distributed MAC system. In all these experiments we used the prototype system shown in Figure 5 and described in Section 4.

5.1 Isolation

To verify isolation, we first constructed appropriate sHype, SELinux and IPSec policies on `shype1` and `shype2`. To the sHype and SELinux policies we added types named for colors, e.g ., `red_t`, `green_t`, and `blue_t`. In the sHype policy, we gave `dom0`s access to all sHype types since each `dom0` plays the role of a MAC virtual machine in our system. Recall that MAC virtual machines assist the hypervisor in enforcing MAC policy and form part of the truted computing base. Also in the sHype policy, we assigned the same sHype type to the client and server `domU`s, e.g., `green_t`, since they form part of the same distributed coalition of virtual machines. To our policy translation tables we added mappings between corresponding sHype and SELinux types, e.g., `green_t` in sHype mapped to `green_t` in SELinux.

As a final step in the policy configuration, we created labeled IPsec policies based on the IP addresses of `shype1` and `shype2`, and on the IP addresses of the client and server `domU`s. The `domU`s, being full-featured virtual machines, have their own IP addresses separate from the IP addresses of `shype1` and `shype2`. So for example, we added an entry to the IPsec Security Policy Database on both `shype1` and `shype2` that instructed the system to allow communication between the client `domU` and the server `domU` via a dynamically established IPSec tunnel between `shype1` and `shype2` that is labeled `green_t`. The SELinux policy has authorization rules that allow `green_t` subjects to send and receive using `green_t` security associations.

Next, we confirmed that `shype1` and `shype2` could not communicate unless the proper IPsec, SELinux and sHype policies were in place at both endpoints. We verified that the `dom0`s on `shype1` and `shype2` would not establish an IPsec tunnel between them until the necessary entries had been added to the IPsec Security Policy Database and the SELinux policy at each endpoint. More specifically, we ran `ping` in both directions between `dom0` on `shype1` and `dom0` on `shype2`, and used `tcpdump` to confirm that no traffic flowed in either direction. We also verified that neither system would forward packets between the IPsec tunnel endpoint in `dom0` and the local `domU` until the necessary entries had been added to the the sHype policy in the Xen hypervisor, the SELinux policy in `dom0`, and the type mapping tables in `dom0`. In this case, we ran `ping` in both directions between the `dom0` in one platform and the `domU` on the remote platform, and between the `domU`s in both plaforms. We again used `tcpdump` to confirm that no traffic to or from a `domU` would be forwarded by a `dom0`.

In summary, only when all the appropriate policies where in place would packets flow between the two `domU`s. In that case the BOINC server successfully sent compute jobs to the BOINC client, who ran the jobs and successfully sent the results back to the server.

5.2 Integrity

To verify the trustworthiness of the hypervisor environments including the `dom0` integrity, we first built a database of software components. For each component, the database contains its measurement (i.e., hash), and whether it's trusted or untrusted. We added database entries for the key trusted components mentioned in the discussion of attestation in the previous section. For example, for the Xen hypervisor we measured its loadable image and its security policy. For each `dom0` we measured its SELinux kernel image, its initial RAM disk, and its MAC policy.

Next, we set up two pairwise attestation sessions. In each session, one system periodically challenges the other system for measurements of the software it has loaded into the hypervisor environment that is relevant for the trustworthiness of the DTRM. We had `dom0` on `shype1` challenge `dom0` on `shype2`, and vice versa. The challenged system returns a quote signed by the TPM of the current values of PCR registers as well as the list of measurements taken by the Integrity Measurement Architecture [14]. The challenging system compares the returned measurements to its database and attestation succeeds if only known components are measured, which can be found in the created databases and which are tagged trustworthy in the database.

Finally, we confirmed that `shype1` and `shype2` could not communicate if any aspect of attestation failed. We verified that the `dom0`s on `shype1` and `shype2` would not establish an IPsec tunnel between them unless the attestation sessions between them showed that they where running the expected software.

We also had `domU` on `shype1` challenge `domU` on `shype2`, and vice versa. This attestation pair establishes security properties by mutually attesting the BOINC client to the BOINC server and vice versa. These properties are essential for the distributed BOINC client-server application to ensure the trustworthiness of the BOINC computation result. For each `domU` we measured its Linux kernel image, its initial RAM disk, and the images and configuration information of applications such as the BOINC client. We also added to the database an entry for a test application that we labeled *untrusted*.

We also verified that the `domU`s on `shype1` and `shype2` would not communicate unless the attestation sessions between them showed correct results. In particular, we tested the effectiveness of our periodic challenges by running our untrusted test application alongside the BOINC client software after communication had been successfully established. The next time the server `domU` challenged the client `domU`, the returned measurements included one for the untrusted application, which caused the server `domU` to shut down network communication with the client `domU`.

6 Discussion and Future Work

In this section, we review the achievements of the prototype relative to the construction of a reference monitor across machines, and briefly mention lessons we learned during its construction.

Retrospective We discuss three requirements for a DTRM listed at the end of Section 2 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. *Verifiable enforcement:* Our prototype uses 13 total authorizations in Xen and SELinux to enforce MAC policies, and the MAC policies themselves only apply to user VMs for 5 of the authorizations. However, the coalition examined is fairly simple. Nonetheless, we are optimistic that verification of the reference monitor and MAC policies at this level of abstraction may prove practical for a number of interesting systems. The main challenge is reducing the MAC VM or enabling verification of reference monitor in spite of significant function in the MAC VM, such as network processing, as discussed further below.

Reducing Attestation Overhead by Shipping Code The biggest challenge to effectively using attestation is interpreting the measurement values returned to the party requesting the attestation from the attested system. This is typically done by matching the returned measurement(s) against a database containing measurements of known-good software. To reduce the requisite maintenance of this database, the DTRM may “ship” code to a new VM when it joins a coalition. Under these conditions, rather than checking a measurement against a set of acceptable measurements, the DTRM knows *exactly* what measurement to expect. In this case, the reference monitor can construct the new user VM and immediately assign it a MAC label. Consider the BOINC example used in this paper. The user VM can already be present and labeled (e.g., *green*) when the user joins it to an existing coalition. Then, the VM receives the *green* BOINC code from the server and attests its integrity to prove to the BOINC server that this code was used.

Minimizing the DTRM 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 enforces MAC policy at two layers, the hypervisor and MAC VM. 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 DTRM enforces coarse-grained policies. We envision that *intra*-VM security controls, such as traditional discretionary access control on file and user granularity or sophisticated mandatory controls such as implemented by SELinux, will work within the *inter*-VM security controls offered by the DTRM into the application environment. These *intra*-VM controls can benefit directly from the DTRM mandatory controls through a hypervisor interface that allows VMs to interact in a controlled way with the hypervisor mandatory access control policy. This structure is advantageous since the most security-critical components are also the most robust. The layered service interface between the user VMs and the hypervisor follows the model of a layered Trusted Computing Base. This may enable 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

In Section 2 we discussed other systems which are related to distributing mandatory access control and remote attestation. We now review additional alternatives to securing distributed systems.

Virtual Private Networks (VPNs) allow roaming individuals to connect to a geographically constrained network as though they were located within those constraints. Today, IPsec [22] 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 a high-assurance environment [19]. The authors combine single-level systems to multi-level distributed environments by using the network pump [20] to safely connect systems of different security levels. Reeds [29] describes the networking of similar machines that are mutually trusted by administration. He looks ahead to 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 have addressed in this work.

Distributed system access control was investigated in the context of the Taos operating system [24]. They defines a general approach for administering authorizations in a distributed system based on discretionary management of access and delegation statements. Some aspects of this work are requirements of ours, such as building trust bottom-up, but we also focus on achieving security guarantees via MAC policies, where Taos supported discretionary delegation.

Trust management systems also aim to describe authorization policies across a distributed system. Such approaches compute authorizations from specifications in certificates that enable more general delegations [7, 10, 25, 26]. These approaches are also discretionary, but they connect user identities to programs. We rely on code identity (i.e., digests) for labeling. Connecting users to subject labels will be done at user VM initialization time, but the exact mechanism is to be determined.

Seshadri et al. propose Pioneer [32], 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 load-time attestation. This work is complementary to our distributed MAC system, which could leverage run-time attestation as well as TPM-based attestation.

Globus is an open architecture for grid computing [11]. 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 [39]. 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). Using our bridged coalitions, Globus could share hardware with other applications.

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 and creating a Distributed, Trusted Reference Monitor (DTRM). 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 mechanism and guarantees for building a distributed reference monitor to support distributed applications such as BOINC. 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 [38], Intel LT [15] and VT [16], and AMD Pacifica [1]), the quantity of code in the bridged TCB can be further reduced. Our bridging architecture enables security policies to be layered based on their complexity, from coarse-grained hypervisor-level policy up to sophisticated application-level policy.

References

[1] AMD. AMD64 virtualization codenamed 'Pacifica' technology, secure virtual machine architecture reference manual. Technical Report Publication no. 33047, revision 3.01, May 2005.

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

[12] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

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

[14] IBM. Integrity measurement architecture for linux. `http://www.sourceforge.net/projects/linux-ima`.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Robert Meushaw and Donald Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 9(4):1–8, 2000.

[28] Microsoft Corporation. Next generation secure computing base. `http://www.microsoft.com/resources/ngscb/`, May 2005.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] Sun Microsystems. Trusted Solaris 8 Operating System. `http://www.sun.com/software/solaris/trustedsolaris/`, February 2006.

[38] 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.

[39] 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.

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