

# Shamon: A System for Distributed Mandatory Access Control

Jonathan M. McCune\*  
Carnegie Mellon University  
jonmccune@cmu.edu

Trent Jaeger  
Pennsylvania State University  
tjaeger@cse.psu.edu

Stefan Berger Ram3n C3ceres Reiner Sailer  
IBM T. J. Watson Research Center  
{stefanb,caceres,sailer}@us.ibm.com

## Abstract

We define and demonstrate an approach to securing distributed computation based on a shared reference monitor (*Shamon*) that enforces mandatory access control (MAC) policies across a distributed set of machines. The *Shamon* enables local reference monitor guarantees to be attained for a set of reference monitors on these machines. We implement a prototype system on the Xen hypervisor with a trusted MAC virtual machine 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). We show that, through our architecture, distributed computations can be protected and controlled coherently across all the machines involved in the computation.

## 1. Introduction

Recent advances are bringing flexible mandatory access control (MAC) to commercial systems, such as Linux [34] and FreeBSD [37], but it does not appear to be straightforward to extend these systems to a distributed security architecture. Previous distributed security architectures, such as those based on Taos [1, 6], Kerberos [21, 27], trust management [10, 13, 23, 24], and grid computing [14, 38] have had successes, but are limited by the lack of distributed trust and by enforcement complexity. They lack a basis for establishing that all the machines in the distributed environment have trustworthy enforcement mechanisms and are configured to enforce the proper MAC policy. These architectures also control resources at a fine granularity, such as individual files, which results in complex enforcement mechanisms and MAC policy specifications. The emerging MAC enforcement mechanisms, such as SELinux, do not address overall system trust and have significant complexity, so it seems likely that extending these architectures directly will result in the same problems. We aim to define a distributed systems security architecture that provides trust in enforcement and limits the complexity of enforcement.

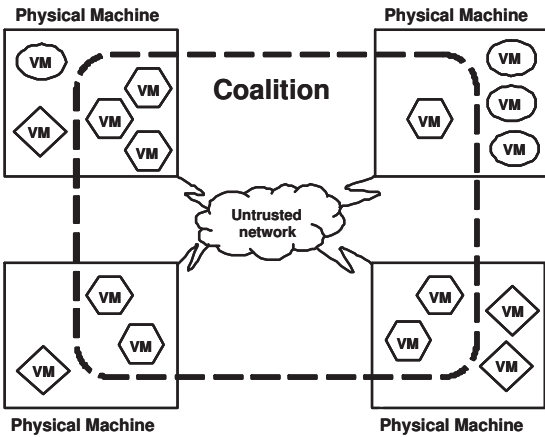
Figure 1 illustrates our high-level goal. A distributed application consists of a *coalition* of virtual machines

(VMs) 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 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. We call the result of this sharing of reference monitors, whose mutual trust can be verified, a *Shamon*. As VMs are added or migrate to new machines, the *Shamon* is verified to ensure its trustworthiness.

In this paper, we introduce a *Shamon* 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 (VMM) as the base code and relying on minimal operating system controls. The Xen hypervisor system is our VMM [5], 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 at present runs a complete operating system (Linux). However, we find that MAC enforcement only requires that the trusted VM control network communication. We use only the SELinux controls for IPsec and packet processing (seven hooks) to perform MAC enforcement in the trusted VM. 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 established using remote attestation [26, 35]. 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

\*This work was done during an internship at IBM Research.



**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.

(i.e., the VM that provides application services) through attestation as well.

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]. 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 enforces the isolation of the BOINC VMs from others and protects other coalitions from the BOINC VMs. We describe how the *Shamon* approach enables verification of trust and MAC-enforced isolation.

The rest of this paper is organized as follows. Section 2 surveys related work and provides background motivation for the problem of building a *Shamon*. Section 3 presents the architecture of our *Shamon*, and Section 4 describes our prototype implementation. Section 5 presents an experimental evaluation of the security features of our prototype implementation, while Section 6 discusses some out-

standing issues and areas for future work. Finally, Section 7 offers our conclusions.

## 2. Background and Related Work

We examine the two main issues in building a secure distributed system: complexity and trust. Previous systems meet one or the other of these requirements, but not both.

**Complexity.** We define MAC enforcement complexity in terms of the number of unique operations in the system that require mediation and the number of statements necessary to describe the MAC policy. In most systems, MAC enforcement is done by an operating system, but the fine granularity of system objects and the variety of applications that need to be controlled result in complex MAC enforcement. Extending this approach directly to distributed systems is not practical.

Current operating systems capable of enforcing MAC, such as Trusted Solaris [36], SELinux [34], and FreeBSD [37], leverage the finest granularity of control offered by the operating system, where individual labels are associated with processes (for subjects) and files (for objects). While such control enables us to reason about the security of our systems in the most flexible manner, it does not appear that such fine-grained control will scale to distributed systems or that it is necessary for such systems. First, fine-grained controls require more complex reference monitor designs, such as the Linux Security Modules (LSM) framework [39], resulting in both large MAC policies (e.g., the 30,000 policy statements in an SELinux *strict* policy) and the challenge of mapping system objects to their labels. Second, 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.

Therefore, we propose to move from the fine-grained controls of operating systems to an architecture that controls communication between applications. The proposed approach uses a virtual machine architecture to control communication. While this architecture provides isolation between applications running in separate virtual machines (VMs), our approach enables flexible control of communication at the virtual machine-level, such that any inter-VM communication can be flexibly allowed or denied. Such a mechanism enables the composition of VM coalitions where member VMs communicate within the coalition and have limited communication with external VMs. We find that enforcement is possible with few enforcement points (5 hooks for enforcement) where we can specify MAC policies (e.g., Type Enforcement [11] or Multi-Level Security (MLS) [7]) at the VM level.

Virtual machines are not a new technology and have long been used for security, but we make several improvements to current systems. First, compared to VM isolation technologies [29], we enable not only VM isolation, but also flexible, but controlled, communication across systems. Second, compared to VM systems with integrity verification for trust (e.g., Terra [15]), we define a complete MAC enforcement mechanism and basic MAC policies for distributed systems. Terra only provides a placeholder for controlling access. Third, we define VM access control at the lowest levels of the system. This contrasts with NetTop [25] which uses a virtual machine monitor (VMWare) and operating system with MAC support (SELinux) to enable what were traditionally physically separate computer terminals on the desks of government employees to be consolidated onto a single system. The NetTop architecture relies extensively on the security controls of the host OS, which we have already shown to suffer from excessive complexity. Finally, our approach takes a pragmatic approach to the security ideals of the VAX VMM [18] and KVM/370 [32] systems, where we provide high performance and flexible function with similar security controls. While we do not provide covert channel controls, we can prevent the execution of conflicting VMs on the same physical platform where this is a concern.

**Trust establishment.** Something that has been lacking in distributed systems, historically, has been a practical basis for trust in the distributed enforcement mechanism. For building a coalition, we must establish trust in the MAC enforcement of each member of the coalition, and we must verify that the MAC policy being enforced on each machine supporting the coalition is consistent with the coalition’s security goals (e.g., secrecy and integrity).

Previous distributed security architectures depend implicitly on a trusted computing base without any practical basis for this trust. For example, trust management systems [10, 13, 23, 24] compute authorizations, but we have no basis to trust that these functions are performed correctly.

Within a single administrative domain, trust is often assumed because all the systems are under the same administrative control. However, it is possible that some of the machines in the domain have been compromised or misconfigured. Typically, no effort is made to verify the correctness of MAC enforcement beyond software updates.

The bootstrapping of trust in a distributed security architecture is described for the Taos system [22]. In this system, a preliminary form of secure boot [4] is proposed where at each step in the boot sequence the current system verifies the integrity of the next step prior to starting its execution. We agree with the requirement of building trust bottom-up, but this work lacked a mechanism to prove trust to remote parties in the distributed application. Further, we also focus on achieving security guarantees via MAC policies, where Taos supported discretionary delegation.

We leverage remote attestation as a basis for building trust in distributed enforcement. While much prior work has been done on remote attestation [26, 35], complexity of software and policy have rendered attestations less meaningful than desired on existing systems. For example, Terra [15] is a VMM-based architecture for providing isolation and includes attestation support. Today, the Xen hypervisor system [5] with Trusted Platform Module (TPM) support [9], enables the solution we present here to enforce mandatory security policies between VMs and to establish trust into the VM management environment, both of which are not addressed by Terra.

The challenges in this work are to determine how to establish trust in a set of machines that participate in a coalition. In particular, we must be able to attest to the enforcement mechanisms of each machine and the consistency of MAC policy enforcement throughout the coalition. This will ensure that each system has a trusted mechanism to enforce MAC requirements, that the MAC requirements are met at each site in the coalition, and that there is a consistent labeling of objects across the coalition systems.

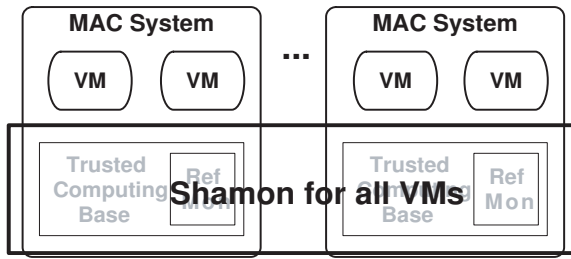
### 3. System Architecture

In this section, we describe the system architecture for a *Shamon* and examine its ability to achieve the guarantees of a host reference monitor across a distributed environment. We begin by providing a high-level overview of our architecture (Section 3.1). Then, the process of extending the *Shamon* is presented, thereby establishing a *bridge* between two systems (Section 3.2).

#### 3.1. Architecture Overview

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 and are running on a single hypervisor system with MAC [30]. 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 *Shamon*.

We have designed a *Shamon* that builds trust in layers, bottom-up, starting 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), bringing us closer to practical formal verification for assurance. 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,



**Figure 2.** The Shamon 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.

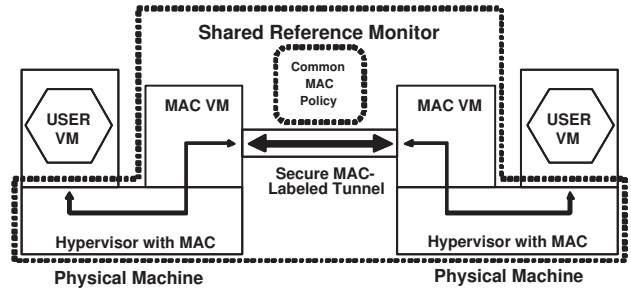
thereby helping to alleviate security-relevant dependencies on components of excessive complexity. A MAC VM and MAC policy attestation complete the establishment of *Shamon* trust. The complexity required of these components can be a significant improvement over host OS-only MAC.

The resulting system is shown conceptually in Figure 2; 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 the correctness of its implementation and MAC policies is practical.

Figure 3 summarizes the primary concepts in our architecture: (1) *hypervisors* are VMMs that run on a single physical machine and enforce the common MAC policy for VM-to-VM communications on that machine; (2) *MAC VMs* enforce the common MAC policy on inter-VM communication across machines; and (3) *secure, MAC-labeled tunnels* provide integrity protected communication which is also labeled for MAC policy enforcement; (4) *user VMs* implement application function; (5) *coalitions* consist of a set of user VMs implementing a distributed application; (6) a *Shamon* consisting of the combination of reference monitors for all machines running user VMs in a single coalition; and (7) *common MAC policies* define MAC policies for a single *Shamon*.

**Hypervisors and MAC VMs.** The hypervisor and MAC VM comprise the reference monitoring components on a single physical machine. The hypervisor controls user VM communication local to that machine, and the MAC VM controls inter-machine communications.

**MAC-labeled tunnels.** Inter-machine communication is implemented via secure, MAC-labeled communication tunnels. The *Shamon* 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 *Shamon* can



**Figure 3.** Example of a Shamon.

control which user VMs can use which tunnels.

**User VMs and coalitions.** 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 where interaction is controlled by the Bell-LaPadula policy [8].

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.

**Shamon.** A coalition’s reference monitor is a *Shamon*. It consists of the union of the reference monitors for the physical machines upon which coalition’s user VMs run (see Figure 2).

**Common MAC policies.** The common MAC policy of a coalition is the union of the MAC policies of the reference monitors in a coalition’s *Shamon*. 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 *Shamon*’s physical machines.

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

### 3.2. 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 *Shamon*. 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 and the *Shamon* mutually verify that their policy-enforcement (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, the joining reference monitor (JRM), will affect MAC policy in two ways: (1) the JRM will add the coalition label and its rights to its local MAC policy, and (2) the *Shamon* common MAC policy will become the union of the JRM's and former *Shamon*'s MAC policies. First, the JRM must verify that the resultant coalition policy is compatible with its current policy (e.g., does not violate isolation guarantees of its other local coalitions). Second, the resultant *Shamon* policy now includes that of the JRM to ensure that overall coalition security goals can be enforced.

We present two different ways that the 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 pre-define 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, the labels and flows implied by the MAC policy are defined by the coalition's *Shamon*. A problem here is that two coalitions may use the same label (e.g., *blue*) to mean different things. The coalitions will have to determine which labels are internal to the coalition (i.e., isolated) and which may have information flows (i.e., the labels are global or known to other specific coalitions). Our prototype uses the first approach, so the MAC policy is fixed at the hypervisor level and coalition policies are mapped to it.

Further, the coalitions must ensure that objects are labeled consistently across the coalition. If objects are labeled *blue* on one system, but objects with same security semantics are labeled *green* on another system, then problems can ensue. At present, we download user VMs and objects for the coalition at join time, so labeling is determined consistently by the coalition.

**Stage 2: Confirm tamper-responding and mediating abilities.** An attestation policy is used to mutually verify JRM and *Shamon* tamper-responding and mediation abil-

ities. 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 (e.g., *green*). In the context of the BOINC example, the BOINC server may want an 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 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).

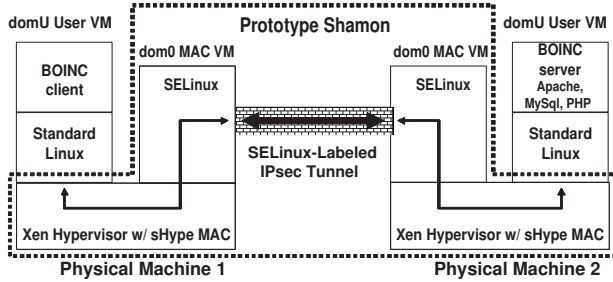
**Stage 4: Secure, labeled communication.** We construct a secure, MAC-labeled tunnel for the bridge in the MAC VM. The secure communication policy is selected when the user VM attempts to communicate with a coalition member and determines the secrecy and integrity requirements of the communication (e.g., 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.

## 4. Implementation

We implemented a *Shamon* 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.

### 4.1. Machine Configuration

We configured two hypervisor systems running Xen [5] with sHype [30], *shype1* and *shype2*. *shype1* runs one or more BOINC clients, each in its own user VM. *shype2* runs a dedicated BOINC [2] server inside a non-privileged user VM. The supervisor domain in each Xen system runs Fedora Core 4 with SELinux [34] 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 [19, 20] to local



**Figure 4.** Bridging the shared reference monitor (*Shamon*) in our distributed computing prototype.

sHype labels, and vice versa. 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 4 shows the structure of our prototype.

**sHype MAC in Xen.** The foundation of a *Shamon* is sHype, a hypervisor security architecture for different virtual machine monitors [30]. 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 of resources (e.g., network, block devices) between different VMs, and a Chinese Wall [12] policy (CHWALL) that controls which VMs can run simultaneously on the same system. We did not use CHWALL policy in our experiments, but our *Shamon* architecture supports its use.

The STE policy component controls sharing between virtual machines by controlling access of virtual machines to VM-to-VM communication, and to any virtual resources 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.

**Device driver and MAC VMs on Xen.** We built and maintain our *Shamon* 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) pol-

icy. `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, 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

The labeled IPsec tunnel(s) between machines in a distributed coalition provide authenticated, encrypted communication while conveying MAC type information. We use labeled IPsec [17] operating in tunnel-mode [20] as the secure communication mechanism between the `dom0`s (MAC VMs) on `shype1` and `shype2`. 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 in Figure 4, reference monitor functionality exists in both the endpoint of the IPsec tunnel in `dom0` and in the hypervisor with sHype.

The type check in `dom0` occurs automatically as part of the normal operating behavior of our IPsec configuration. The IPsec tunnels that we employ use tunnel-mode extensions to labeled IPsec [17]. These researchers added support for SELinux subject labels to be included in the negotiation process when IPsec connections are established. IPsec policies are authorized for subjects, which are user VMs in our system. User VMs are labeled based on the STE labels assigned to the VMs by sHype. Note that we depend on six `dom0` kernel mediation points for correct IPsec policy enforcement: four authorize allocation and deallocation of IPsec policies and security associations, and two filter incoming and outgoing packets.

The functionality of labeled IPsec [17] 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 *Shamon* of a distributed coalition must be preconfigured with all possible SELinux subject types that may be needed 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 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 labeled IPsec extensions 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, numbering approximately 850 lines of commented C code, returns an SELinux security identifier (SID) when given a `flowi` and direction. A `flowi` is a kernel struct which maintains state for a generic Internet flow, including the input interface (IIF), output interface (OIF), and source and destination IP addresses.

We added two data structures (linked lists of `structs`) 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 `domUs`. 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 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 *Shamon*. Recall from Figure 4 that the *Shamon* 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.

We also use the Integrity Measurement Architecture (IMA) [31] to establish trust into the user VMs running on top of the *Shamon* (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 [9], 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 emulate 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.

## 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 4 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 trusted 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 instructs the system to allow communication between the BOINC client *domU* and the BOINC 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. 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 sHype policy in the Xen hypervisor, the SELinux policy in *dom0*, and the type mapping tables in *dom0*.

In summary, only when all the appropriate policies are in place can packets flow between the two *domU*s. In that case, the BOINC server successfully sends compute jobs to the BOINC client, who runs the jobs and successfully sends 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 *Shamon*. 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 [16, 31]. The challenging system compares the returned measurements to its database of known trustworthy components. Attestation succeeds if the measured components are all found 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 were running the expected software.

Further, we 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 added to the database an entry for a test application that we labeled *untrusted*.

We 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

In this section, we review the achievements as well as the limitations of the prototype relative to the construction of a reference monitor across machines.

**Distributed tamper-proofness.** Our prototype requires a VM to successfully attest its ability to uphold the secu-

rity 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.

**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.

**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 we 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.

**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 *Shamon* enforces coarse-grained *inter-VM* policies. *Intra-VM* controls can benefit directly from the *Shamon* 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.

**Mitigating Covert Channels.** The individual reference monitors will not have complete formal assurance, so some information flows, such as covert channels, may not be enforced. The sHype hypervisor MAC policy enables the 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 [30].

**Runtime tamper-responsiveness.** TPM-based attestation mechanisms (e.g., IMA [31]) measure inputs at load-time. Thus, runtime tampering may go undetected. 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. Other techniques, such as Copilot [28] 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.

**Hardware attacks.** This architecture does not protect the system against cracking 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).

## 7. Conclusions and Future Work

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 *Shamon*. 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 *Shamon*. This work provides a mechanism and guarantees for building a distributed reference monitor to support distributed applications. 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. 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.

Future work includes reducing the size of the MAC VM and exploring additional policy options. 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 only the critical components for MAC operation. This specialized code may be designed to enforce more expressive policies, such as Chinese Wall policies, which expands the applicability of *Shamon*.

## References

- [1] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1993.
- [2] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the Workshop on Grid Computing*, Nov. 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] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1997.
- [5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2003.
- [6] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS wide area security architecture. In *Proceedings of the USENIX Security Symposium*, Jan. 1998.
- [7] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report

- ESD-TR-75-306, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1976.
- [8] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical report, MITRE MTR-2997, March 1976.
- [9] S. Berger, R. Cáceres, K. Goldman, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the USENIX Security Symposium*, July 2006.
- [10] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust-management system, version 2. IETF RFC 2704, Sept. 1999.
- [11] W. E. Boebert and R. Y. Kain. A practical alternative to heirarchical integrity policies. In *Proceedings of the National Computer Security Conference*, 1985.
- [12] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1989.
- [13] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, Sept. 1999.
- [14] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3), 2001.
- [15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2003.
- [16] IBM. Integrity measurement architecture for linux. <http://www.sourceforge.net/projects/linux-ima>.
- [17] T. R. Jaeger, S. Hallyn, and J. Latten. Leveraging IPSec for mandatory access control of linux network communications. Technical Report RC23642 (W0506-109), IBM, June 2005.
- [18] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, 1991.
- [19] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). IETF RFC 2406, Nov. 1998.
- [20] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF RFC 2401, Nov. 1998.
- [21] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). Internet Draft, Sept. 1992.
- [22] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.
- [23] N. Li, B. N. Grosz, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, Feb. 2003.
- [24] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 89–103, June 2003.
- [25] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 9(4):1–8, 2000.
- [26] Microsoft Corporation. Next generation secure computing base. <http://www.microsoft.com/resources/ngscb/>, May 2005.
- [27] Open Software Foundation. *Introduction to OSF DCE*. Prentice Hall, Englewood Cliffs, NJ, 1993.
- [28] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A coprocessor-based kernel runtime integrity monitor. In *Proceedings of the USENIX Security Symposium*, 2004.
- [29] T. T. Russell and M. Schaefer. Toward a high B level security architecture for the IBM ES/3090 processor resource / systems manager (PR/SM). In *Proceedings of the National Computer Security Conference*, Oct. 1989.
- [30] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen openource hypervisor. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2005.
- [31] 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.
- [32] M. Schaefer, B. Gold, R. Linde, and J. Scheid. Program confinement in KVM/370. In *Proceedings of the ACM National Conference*, Oct. 1977.
- [33] E. Shi, A. Perrig, and L. V. Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.
- [34] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.
- [35] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the European Symposium on Research in Computer Security*, Oct. 2002.
- [36] Sun Microsystems. Trusted Solaris 8 Operating System. <http://www.sun.com/software/solaris/trusted-solaris/>, Feb. 2006.
- [37] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [38] 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*, June 2003.
- [39] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the linux kernel. In *Proceedings of the USENIX Security Symposium*, 2002.
- [40] H. Yin and H. Wang. Building an application-aware IPsec policy system. In *Proceedings of the USENIX Security Symposium*, 2005.