Analysis of Virtual Machine System Policies

Sandra Rueda, Hayawardh Vijayakumar, Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
The Pennsylvania State University
University Park, PA, 16802
{ruedarod,huv101,tjaeger}@cse.psu.edu

ABSTRACT

The recent emergence of mandatory access (MAC) enforcement for virtual machine monitors (VMMs) presents an opportunity to enforce a security goal over all its virtual machines (VMs). However, these VMs also have MAC enforcement, so to determine whether the overall system (VM-system) is secure requires an evaluation of whether this combination of MAC policies, as a whole, complies with a given security goal. Previous MAC policy analyses either consider a single policy at a time or do not represent the interaction between different policy layers (VMM and VM). We observe that we can analyze the VMM policy and the labels used for communications between VMs to create an inter-VM flow graph that we use to identify *safe*, *unsafe*, and *ambiguous* VM interactions. A VM with only safe interactions is compliant with the goal, a VM with any unsafe interaction violates the goal. For a VM with ambiguous interactions we analyze its local MAC policy to determine whether it is compliant or not with the goal. We used this observation to develop an analytical model of a VM-system, and evaluate if it is compliant with a security goal. We implemented the model and an evaluation tool in Prolog. We evaluate our implementation by checking whether a VM-system running XSM/Flask policy at the VMM layer and SELinux policies at the VM layer satisfies a given integrity goal. This work is the first step toward developing layered, multi-policy analyses.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and protection; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Security

SACMAT'09, June 3–5, 2009, Stresa, Italy.

Keywords

Policy Analysis, Virtual Machines, Xen Security Modules (XSM), SELinux

1. INTRODUCTION

The recent development of reference monitors for virtual machine monitors (VMMs) provides a mechanism (e.g., Xen Security Modules (XSM) [5]) to enforce mandatory access control (MAC) policies on operations that virtual machines (VMs) execute on VMM resources. We refer to a VMM and its constituent VMs as a *VM-system*. In a VM-system, a VMM allocates memory, communication channels, and other resources among VMs. A VMM reference monitor aims to control whether one VM can access another's memory or communicate with another VM directly. In addition, VM-systems not only enforce a VMM policy, they may also host virtual machines that run operating systems that implement their own MAC controls (e.g., SELinux [17]).

A problem is to determine whether the VM-system enforces a security policy that satisfies a security goal (e.g., secrecy and integrity restrictions). This task is difficult because: (1) the VMM and some VMs are trusted with the enforcement of the VM-system security policy; (2) such MAC policies can be complex; and (3) the individual VMM and VM MAC policies are developed independently. First, VM-systems often contain privileged VMs that manage VMM resources, and several other VMs may be entrusted with the prevention of leakage or protection from untrusted data. As a result, it is necessary to consider these VM MAC policies along with the VMM policy in determining the meaning of the policy that governs the VM-system as a whole. Second, both VMM and VM MAC policies may consist of many rules. For Xen, the introduction of the Xen Security Modules (XSM) framework enables the enforcement of comprehensive control over VMM resources within the hypervisor. The XSM/Flask [5] policy model for XSM is based on SELinux, so VMM policies will be comprehensive, but non-trivial. Combined with the trusted VM MAC policies, many thousands of policy rules will have to be considered to determine whether a security goal is enforced correctly. Third, these MAC policies are not developed with a system-wide goal in mind, so it will be necessary to determine such security goals, and relate (map) each relevant MAC policy to the goal.

Previous analysis of MAC policies either consider a single policy at a time and/or do not represent the interaction between policy enforcers at different system layers. First, many policy analyses have been constructed to evaluate se-

curity goals for a single MAC policy [9, 21, 27, 22]. These analyses convert MAC policies into an *information flow* [7] graphs that represent which MAC labels can operate on (i.e., read or write) other MAC labels. A naive approach would be to compose these policies into a single information flow graph and evaluate the resultant graph. However, such a graph would be prohibitive in size, preventing effective analysis. The current SELinux reference policy [26] has around 2200 labels and enables 200000 information flows among these labels (the although the actual number depends on system configuration). Second, those policy analyses that consider multiple policies [12, 14, 2] do not consider that some policies may control operations at different software layers. For example, a VMM policy restricts usage of VMM resources, but the VM policy controls the usage of OS resources. We have previously examined layering between the application and the OS [19], but this is the first work we are aware of that examines the relationship between VMM and VM policies.

In this paper, we develop an information flow-based model to represent inter-VM flows that the VMM policy enables and inter-VM interactions the local VM policies enable (i.e., via *VM-visible labels*). We define an approach for constructing such graphs automatically by identifying the information flow mapping that is required between VM and VMM labels. Using our previously-defined compliance analysis [10], we show that performing an inter-VM analysis and VM-local analyses for certain VMs is sufficient to prove compliance for the composite of these policies. We have implemented our approach in a Prolog-based tool. We demonstrate use of this tool on VM systems consisting of XSM/Flask policies in the VMM and SELinux policies in the VMs to show that these policies satisfy an integrity policy goal.

In the next section we present background on compliance analysis of individual mandatory access control (MAC) policies. In section 3 we elaborate on the problem that emerges because of the composition of VMM and VM policies on a VM-system. In section 4 we present an analytical model to represent a VM-system. We provide an implementation of the model, and a case study in section 5. Finally, we present our conclusions and future work.

2. BACKGROUND

In this section we review the policy compliance problem for individual MAC policies and show how this problem is analyzed [10, 19]. Later, we build on this problem to analyze VM-systems that consist of multiple MAC policies.

In a MAC system all resources, subjects and objects, are represented with labels, and there is a policy that defines access control rules in terms of those labels, i.e. a subject with a *subject label* is allowed to perform an *operation* upon an object with an *object label*. To evaluate the security enforced by MAC policies, we represent policies as information flow graphs.

DEFINITION 2.1 (INFORMATION FLOW GRAPH). *An information flow graph is a directed graph $G = (V, E)$ where V is the set of vertices, and E the set of edges in the graph. Each $v \in V$ is labeled with a label from $\mathcal{L}$, the set of labels assigned to subjects and objects in a MAC policy. An edge $(u, v) \in E$ if (1) u has* `write` *access to v, or (2) v has* `read` *access for u. We use the functions V(G), and E(G) to get the vertices and edges respectively.*

Several systems that implement mandatory access controls (MAC) are currently available [25, 28, 17]. We present SELinux as an example to illustrate the *Information Flow Graph* concept, and the properties we can evaluate on that graph. SELinux is an extension of the Type Enforcement model [4], it uses *types* for labels. Administrators assign permissions using *allow* rules where the first argument is the subject type (label), the second argument is the object type (label) that includes the datatype (e.g., file), and the third argument is the permissions (operation set) for that object label. For instance, `allow admin_t etc_t:file {read write}`, enables administrators to access files in the directory `/etc`, with read and write permissions, assuming administrators are assigned the label `admin_t` and files in the directory `/etc` are assigned the label `etc_t`.

To create the graph we use SLAT's [9], and PAL's [21] definition of information flow. First, we classify all permissions in two categories: `read_like` and `write_like`. `read_like` permissions enable subjects to get information about objects, `write_like` permissions enable subjects to modify objects. Second, we say there is an information flow from a resource `r1` with type `t1` to a resource `r2` with type `t2` if the policy has (1) an access rule `allow t1 t2:class perm` and `perm` is classified as `write_like`, or (2) an access rule `allow t2 t1:class perm` and `perm` is classified as `read_like`.

We use the information flow graph to check security properties. For instance, we can determine all the types that are allowed to write to a target type, or all the types a target type is allowed to write to. We can also evaluate the Biba integrity [3] of a particular type or set of types (e.g., a proposed trusted computing base (TCB)). These analyses can be generalized to evaluate security goals. Some security goals can be represented as information flow graphs. For example, confidentiality and integrity have long been known to be information flow properties.

DEFINITION 2.2 (INFORMATION FLOW GOALS). *We specify an* integrity information flow goal *as a security lattice $\mathcal{L}_i = (L_i, \sqsubseteq_i)$. The lattice $\mathcal{L}_i$ has a top $\top$ and a bottom $\bot$ elements that represent the highest and lowest integrity level. Given $a, b \in L_i$, $\texttt{a} \sqsubseteq_i \texttt{b}$ indicates that a can flow to b, but not the other way around.*

We specify a confidentiality information flow goal *as a security lattice $\mathcal{L}_c = (L_c, \sqsubseteq_c)$. The lattice $\mathcal{L}_c$ has a top $\top$ and a bottom $\bot$ elements that represent the lowest and highest secrecy levels. Given $a, b \in L_c$, $\texttt{a} \sqsubseteq_c \texttt{b}$ indicates that a can flow to b, but not the other way around.*

DEFINITION 2.3 (POLICY COMPLIANCE). *We write $u \hookrightarrow_G v$ if there is a path between vertices u and v in the graph G, and $i \sqsubseteq_{\mathcal{L}} j$ if i can flow to j in the lattice $\mathcal{L}$. An information flow graph $G = (V, E)$ is compliant with an information flow goal $\mathcal{L} = (L, \sqsubseteq)$ if exists a mapping $h : V \to L$ such that for all $u, v \in V$, $h(u), h(v) \in L$, if there is a path $u \hookrightarrow_G v$, then $h(u) \sqsubseteq_{\mathcal{L}} h(v)$.*

Figure 1 shows an information flow graph G and an integrity information flow goal I for a generic application. The information flow graph complies with the information flow goal under the mapping function *int* because for every flow $src \hookrightarrow_G tgt$, $int(src) \sqsubseteq_I int(tgt)$ is enabled too. For example, $etc\_t \hookrightarrow_G app\_t$ and $int(etc\_t) \sqsubseteq_I int(app\_t)$.

In this paper, we extend our initial approach to evaluate compliance of VM-systems. We decided to extend our

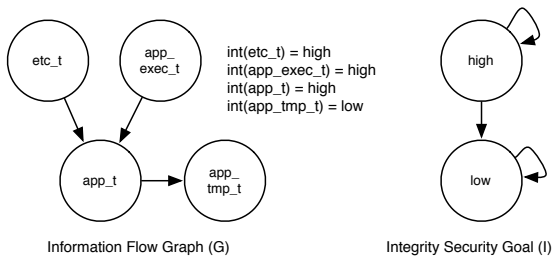

Figure 1: Information flow graph and integrity information flow goal for a generic application. The graph is compliant with the goal under the mapping function *int_level*.

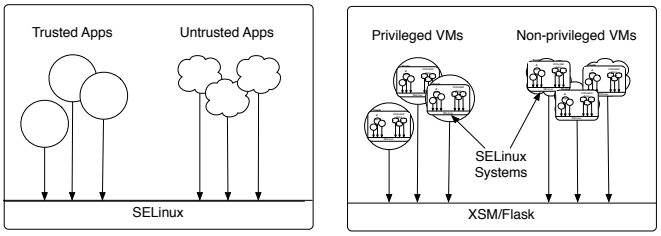

Figure 2: From single MAC policy systems to VM systems. A MAC policy system governs how processes use OS resources, and a VMM policy governs how VM use VMM resources.

approach because it has two key advantages compared to other approaches: (1) we can use the security goal as an intermediate representation, therefore we can relate several policies with different semantics and granularities, and (2) we can automatically deduce some security goals, therefore reducing the burden of goal specification on administrators.

3. PROBLEM DEFINITION

3.1 VM-System Architecture

In this paper, we consider a single physical machine that runs a VMM directly over hardware (Type 1 VMM [8]). Type 1 VMMs, in the interest of remaining bug-free, aim for minimal function. We consider the case where they offload responsibilities of administration (e.g., creation and deletion of VMs) and device driver implementation to other VMs, and only provide an interface to support these capabilities [18, 13]. For example, the VMM may offer the abstraction of shared memory and virtual interrupts to allow device drivers to communicate with the VMs using them.

In such a system, there is a policy in the VMM, and policies in each of the VMs. Figure 2 shows similarities between a single MAC policy and a VMM policy in a VM-system. The VMM policy governs the flows between VMs using the abstractions provided by the VMM. Thus, a VMM policy labels VMs and other VMM resources and describes the access rights of VM labels to VMM resource labels. The VM policy is an OS MAC policy, which governs the flows among process using the abstractions provided by the OS, as described in Section 2 for SELinux.

We distinguish three types of VMs in our systems. First, because we want to keep the VMM as small as possible, VM-systems have *privileged VMs* that manage access to physical resources (e.g., disk and memory) for every VM. All VMs communicate with the privileged VMs to obtain and use the system's physical resources. Second, *service VMs* provide general system functions (e.g., VM loading and integrity measurement [1]) for other VMs. Third, *user VMs* run applications by using resources from privileged VMs and services from service VMs.

A VM-system provides mechanisms for VMs to communicate. For example, the Xen VMM system (Section 5.1) enables VMs to send messages using grant tables and event channels. Also, VMs may communicate by sharing memory. Processes running inside VMs do not use these mechanisms directly, however. The VM operating systems map inter-machine communication (i.e., networking) to these VMM mechanisms. Note that the VM operating system also uses VMM mechanisms to obtain physical resources from the privileged VMMs, but this is invisible to the VM processes and its MAC enforcement (i.e., the reference monitor in the VM).

In a VM-system, both the VMM and a VM's operating system may authorize the same communication. First, all inter-VM communications are authorized by the VMM using its labels. For example, the Xen VMM authorizes a VM's access to sending a message via a grant table by determining whether the VM's label in the VMM policy can send to grant tables of a particular label. In addition, when the VM's operating system is aware of a communication, it may also authorize it. For example, many operating systems can authorize network communication. In this case, the VM operating system determines whether a process can send a message via a network channel (e.g., IPsec). Thus, we define two types of inter-VM flows in a VM-system:

Type 1 Flow: An object that enables VM-to-VM communication that is assigned a VMM label only. Such flows are either invisible to the OS (e.g., access to physical resources via privileged VMs) or given a default OS label (e.g., unlabeled networking). Such flows are assigned the VMM label, so the security of the data communicated is based on the security level of this label.

Type 2 Flow: An object that enables VM-to-VM communication that is assigned both a VMM and VM label. Such flows are assigned a composite label `VMM_label.VM_label`. The security of this label is determined by the level of the VM label, but that level must be consistent with the VMM label's security level.

3.2 VMs with Multiple Security Levels

VMs may be authorized to access data at multiple security levels. For example, the privileged VMs provide physical resources for all VMs, so they are trusted with all security levels of data in the system. Service VMs may also be trusted with data of multiple security levels, as they may serve multiple VMs. Even some user VMs may be entrusted with handling multiple security levels of data. This is particularly true for integrity, where a VM may be trusted to perform high integrity functions, even when it receives low integrity requests.

VMs entrusted with multiple security levels must ensure that illegal information flows do not result, this task includes

conveying labels to other VMs to ensure that they enforce security properly. As an example, consider Figure 3. Suppose we have VMs $VM1$, $VM2$ and a privileged VM, and processes labeled A, B, C, and D, and assume that process B sends data to process C. The VM is assigned a label in the VMM policy and the processes are assigned labels in the VM policy. Suppose that the processes have different labels that also imply different secrecy levels, $L1$ and $L2$ in $VM1$ and $L2$ and $L3$ in $VM2$. Assume that $L1 \sqsubseteq L2 \sqsubseteq L3$.

When B sends a message to C, we need to determine the security level of the data that may be communicated to assess the security of the VM-system. For example, since $VM1$ has both $L1$ and $L2$ processes, it may send either $L1$ or $L2$ data. Hence, we say that $VM1$'s VMM label implies a *range of security levels*. This may result in ambiguity in analysis, because $VM2$ is not allowed to access $L1$ data. However, when B of level $L2$ sends a communication to C of level $L2$, this should be allowed. The problem is that the level of the process labels is lost — we do not know that the data is labeled $L2$.

In order for such a communication to be authorized, the VMM and two VMs must correctly authorize the flow. The VMM policy may allow data from $VM1$ label to flow to $VM2$, but this would involve some risk for the VMM as some unauthorized flows could result. The VMM policy is depending on the VM policies in $VM1$ and $VM2$ to correctly enforce the flow. Fortunately, VM policies can enforce inter-VM flows. For example, netlabel [16] and Labeled IPsec [11] enable two VMs (systems) to agree on a label before communicating data between the VMs. For example, the two processes B and C could cause their respective VMs to negotiate a network flow at a label X. We call such VM policy labels used to label inter-VM flows, *VM-visible labels* (see Definition 4.3). Since B and C are both running at level $L2$, the flow is then from $L2$ to $L2$, so this is legal.

Unfortunately, VMs may not declare all inter-VM flows using VM-visible labels. As described above, VMs access the privileged VM in a manner that is invisible to the MAC policy. The mechanism to virtualize the VM creates a problem, because the OS does not tell the privileged VM the security level of the data being conveyed. Further, inter-VM communications may not be labeled: the two VMs do not specify a label for the communication channel. In both of these cases, we must assume that any data from the VM may be communicated, so these communication channels can have data spanning the full range of security levels of VM.

Based on the notion of multiple security levels in VMs, we refine our definitions of the types of inter-VM flows.

Type 1 Flow: Invisible or unlabeled inter-VM flows are assigned a VMM label only. The security level of this flow is the range of security levels for the sending VM in the VMM policy.

Type 2 Flow: These flows are associated with VM-visible labels. The security level of this flow is the security level of its VM-visible label, as long as that level is within the range of security levels for the VM based on the VMM policy.

3.3 Summary

The problem that we address in this paper is as follows. For a VM-system with a VMM MAC policy and multiple VMs with their own MAC policies that may span a range of security levels, we aim to build an information flow-based analysis approach that uses Type 1 and Type 2 Flows to determine whether all inter-VM flows are compliant with an information flow security goal. Because VMs may support multiple security levels, we cannot determine whether an intra-VM flow may not comply, but we can use the compliance analysis of Definition 2.3 to evaluate this. Further, we will not need to verify VM compliance for VMs that only support one security level, as compliance on inter-VM flows verifies their overall compliance. This reduces the number of VMs that must be evaluated.

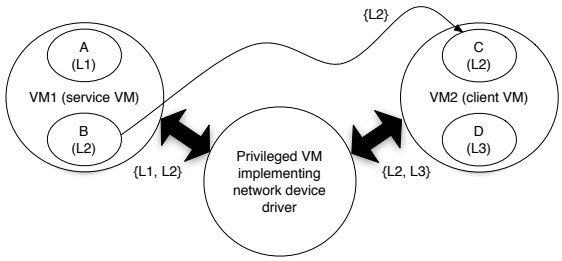

Figure 3: Information flows in a 3-VM system with with applications A, B in $VM1$ and C, D in $VM2$. B is sending data to C. Labels of applications are within parentheses. Labels of flow are marked on the arrow.

3.4 Related Work

Our work provides a model to analyze *multiple, layered policies in a VM-system*. We perform a global analysis on VM interactions, followed by local analyses of VM policies where required. Though previous work deals with multiple policies, they do not address policy layering in VM systems, and the reduction of analyses resulting from it. We present below previous approaches dealing with multiple policies.

The Flexible Authorization Manager Framework (FAM) [12] enables specification and enforcing of multiple access control models within a single system. The framework has an authorization manager that governs access control decisions and a language to define the policy that the authorization manager will enforce. This approach is different from ours, while FAM focuses on policy specification and enforcement, we aim to provide a framework to analyze simple and composite policies.

Another framework [14] proposes a formalism to specify, compare and integrate access control policies. The formalism represents policies as graphs and uses the theory of graph transformation to integrate policies into one graph and analyze them. By taking advantage of the policy layering in a VM-system (VMM policy control relationships between policies while VM policies control objects inside VMs), we do not need to integrate all the policies into a single graph, therefore we aim to reduce the size of the actual representation while keeping the information we need to evaluate information flow security goals.

The MACS [2] framework enables administrators to specify multiple access control policies within the same system. The authors define policy composition as the coexistence of several access control models such that each of them handles a different partition of the set of objects to be protected. This approach is not appropriate for our analysis because

we need to consider the objects that the involved policies (VMM and VMs) share, these objects are the enablers of interaction between policies that otherwise would be isolated from each other.

4. VM SYSTEM POLICY MODEL

This section presents the model we developed to represent VM-systems and enable administrators to evaluate VM-system's properties. It represents a composition of single MAC policies, one at the VMM layer and several at the VM layer, and the interactions among VMs. To do so, we extend our model to analyze single MAC policies (Section 2).

A first approach to represent a VM-system as a whole would be to create an information flow graph with each of the individual flow graphs of all VMs and add edges between nodes of different VMs and applications for *Type 1 and Type 2* flows. Though this representation is expressive, it is redundant and the information flow graph size increases the requirements to store and handle the model.

We observe that evaluating properties of a VM-system as a whole is the same as evaluating those properties on their inter-VM interactions and then evaluating each VM. We use this observation to develop our model.

4.1 VM-System Model

Below, we present our VM-system policy model. We assume that the only way data can flow in and out from a single VM is through the abstractions provided by the VMM, and the network.

A virtual machine vm_i in a VM system has two associated elements: (1) a label, and (2) a local MAC policy. The VMM policy assigns the VM a label and specifies access control rules on VMM resources based on that label. The local policy is the VM's OS MAC policy. Also, a VM's label is assigned integrity and secrecy level ranges. A VM range indicates the degree to which the VM is trusted to enforce system requirements, protecting higher integrity and higher secrecy data from lower secrecy and lower integrity processing. A range also defines the lowest and highest levels that may be associated to elements a VM contains and may send and receive in interactions with other VMs. Some VMs may be single level, the low and high values in its range are equal e.g. `low-low`.

Definition 4.1 (Integrity/Secrecy ranges for VMs). *Virtual machines in a VM-system are assigned integrity and secrecy ranges. The functions* `integrity` *and* `confidentiality` *map a VM to its integrity and confidentiality ranges respectively. The functions* `lint` *and* `hint` *return the low and high levels in an integrity range respectively. The functions* `lconf` *and* `hconf` *return the low and high levels in the confidentiality range respectively.*

We use the function `lint` to define the label of any information that a VM may send by default, and `hconf` to define the label of any information a VM may receive by default. Because the VM label represents the label of any information that the VM may send, the VMM must interpret this label as having the highest secrecy of any label in the VM. If a VM label's secrecy were not the highest secrecy in its range, then an unlabeled flow could leak secret information. Similarly, the VM label's integrity must be the lowest integrity in the VM's range to prevent a receiver from being tricked into accepting data at the wrong integrity level.

As previously stated, there are two types of flows between VMs. We use the next definitions to represent those concepts.

Definition 4.2 (Type 1: Default Flows). *A default flow represents a communication channel between VMs enabled by a VMM policy.*

Definition 4.3 (VM-visible labels). *A VM-visible label is a label assigned by two applications running on different VMs to a channel they will use to communicate. We add the label of the hosting VM to differentiate between a label $vm_i.l$ supported by a VM vm_i and a label $vm_j.l$ supported by a VM vm_j.*

Definition 4.4 (Type 2: VM-visible Flows). *A VM-visible flow represents a communication channel between two applications running in different VMs with an associated VM-visible label.*

An information flow graph $G = (V, E)$ that represents a VM-system includes the information flows among VMs and VM-visible labels in the system via Default and VM-visible flows (Type 1 and Type 2 Flows, as defined in Section 3.2).

Definition 4.5 (VM Information Flow Graph). *A VM information flow graph $G = (V, E)$ is an information flow graph such that V contains: (1) the labels assigned to the VMs by the VMM policy and (2) the VM-visible labels. E contains: (1) the flows allowed by the VMM policy and (2) the flows enabled by VM-visible labels.*

4.2 Compliance

In this section, we outline an algorithm to evaluate compliance for VM-systems based on the analytical model we presented in Section 4.1. First, we build the VM information flow graph (Definition 4.5) of the VM-system. Second, we build a security goal for the VM-system and resolve the mapping between the security levels in the security goal and the labels of the information flow graph. Although this task requires manual work, we discuss ways to reduce manual effort. Third, we evaluate compliance of the inter-VM flows of the VM-system. A complete, successful evaluation will show that each flow in a VM-system is *safe* (see Definition 4.6, below). Once we show that inter-VM flows are safe, then it is only necessary to evaluate local VM compliance using the approach outlined in Definition 2.3.

Step 1. Build VM information flow graph. In this step we describe how to build a VM information flow graph (Definition 4.5). First, we create the set of vertices: we add a vertex for: (1) every label the VMM policy associates with hosted VMs, and also (2) for every VM-visible label. Second, we create the edges that represent information flows: we add an edge for: (1) every information flow defined by the VMM policy (Type 1 Flow), and (2) every VM-visible type that enables a labeled communication channel (Type 2 Flow).

Figure 4 shows a scheme of the policies associated with a VM-system. This VM-system has four VMs: `dom0_t`, `domp_t`, `doms_t`, and `domu_t`. The solid line arrows between VMs represent inter-VM information flows enabled by the VMM policy. The dashed ellipses represent local VM policies and the dashed arrows VM-visible flows.

Step 2. Define security goal and mapping function. In this step, we define an information flow goal $\mathcal{L}$

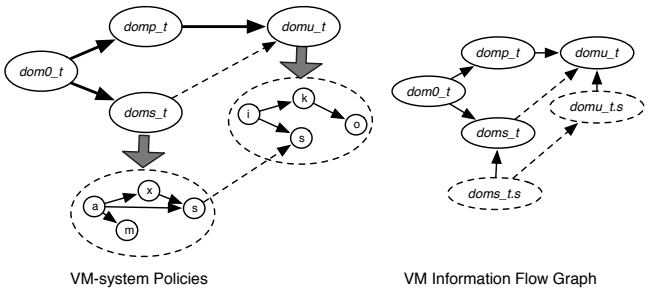

Figure 4: VM-System Policy and Information Flow Graph. `dom0_t`, `domp_t`, `doms_t`, and `domu_t` are VMs. The solid line arrows between VMs represent information flows enabled by the VMM policy. The ellipses represent local VM policies. The dashed arrows represent VM-visible flows. The information flow graph shows our VM-system representation.

that determines the authorized flows between the security levels in the VM-system, and we map the labels in the VM information flow graph to the security levels.

Although these tasks require manual specification, we have mechanisms to automatically deduce some goals and mappings. For instance, we can compute an initial integrity goal based on the relationships among virtual machines, VMs that run VMM back-end services require the highest integrity level, and VMs that run servers require higher integrity than client VMs. These relationships do not change with VM migration since the purpose of the VM does not change, for instance, a client VM will still be a client VM after migration. Application data may also have integrity and secrecy requirements, we expect them to be specified as part of the application policy.

To compute the mapping function we use the identity function as a starting point. We use Type 1 flows to build the goal and use the declared Type 2 flows to help administrators identify the labels that must be mapped. Finally, we analyze the internal MAC policies to help administrators identify the applications that can communicate to other systems via non-labeled channels so they can be resolved. Additional mechanisms to automate the generation of goals and mapping functions are part of our future work.

Step 3. Verify Compliance for VM information flow graph. We notice that evaluating compliance of a VM-system as a whole is equivalent to evaluate compliance of the VM information flow graph (Definition 4.5) and compliance of each one of the local VM MAC policies.

Definition 4.6 (Information Flow Compliance). *Given a lattice $\mathcal{L} = (L, \sqsubseteq)$ and a flow $(u, v) \in E$ and mapping functions* `integrity` *and* `secrecy`*, the flow is compliant if it is SAFE, non-compliant if it is UNSAFE, and has unknown compliance if it is AMBIGUOUS, as defined below (in terms of integrity, confidentiality is analogous).*

let `iu=integrity(u)`, `iv=integrity(v)`*:*
$[$ `lint(iu)` $\sqsubseteq$ `hint(iv)` $] \rightarrow SAFE(u, v)$
$[$ `hint(iu)` $\not\sqsubseteq$ `lint(iv)` $] \rightarrow UNSAFE(u, v)$
$[$ (`lint(iu)` $\sqsubseteq$ `lint(iv)` $\wedge$ `hint(iv)` $\sqsubseteq$ `lint(iu)`) $\vee$
(`hint(iu)` $\sqsubseteq$ `lint(iv)` $\wedge$ `hint(iv)` $\sqsubseteq$ `hint(iu)`) $] \rightarrow$
$AMBIGUOUS(u, v)$

A flow is $SAFE$ if the lowest integrity of the source label *can flow to* the highest integrity of the target label. According to Biba [3], all possible combinations of these levels would be secure. A flow is $UNSAFE$ if even the highest integrity that the source label may generate *cannot flow to* the lowest integrity the target label can accept, so all possible flows are insecure (i.e., violate Biba integrity). A flow may be $AMBIGUOUS$ if the level ranges associated to the source and target labels overlap, so the flows between these labels may or may not violate Biba integrity. Further analysis will be necessary to determine whether the flow is $SAFE$ or not.

Step 4. Find Information Flow-Safe VMs. After evaluating *Information Flow Compliance*, VMs that have only $SAFE$ flows may be isolated from the inter-VM flow analysis. As such VMs are only involved in $SAFE$ interactions with other VMs, then they cannot cause a security violation based on their interactions with other VMs.

Definition 4.7 (Information Flow-Safe VM). *Given a VM Information Flow Graph $G = (V, E)$, define $Flows(vm)$ for a VM vm, as all flows $(u, v) \in E$ where $u = vm$ or $v = vm$. A VM vm is said to be* information flow-safe *if $\forall (u, v) \in Flows(vm) \rightarrow [SAFE(u, v)]$. An information-flow safe VM cannot cause an information flow error in any other VM in G, so the VM-system can be assessed independently from vm.*

The result is that an information flow-safe VM can be isolated from the VM information flow graph. However, if the VM handles a range of security levels it must have its local VM MAC policy evaluated for compliance. We can perform this evaluation independently from that for the VM-system.

Step 5. Disambiguate Flows. A VM may have flows that are $AMBIGUOUS$. To disambiguate such flows, it is necessary to evaluate the VM's MAC policy to determine the actual flows that can result. We must look at the MAC policies at both the output and input VMs. For each output (input) label, we must determine the possible security levels for data that may be sent (received) using that label. For a VMM label, the processes with the ability to send to (receive from) an `unlabeled_t` channel in the VM's SELinux policy can send (receive) data at that label. Thus, it is the security level of these processes that determines the possible security levels of data that may be communicated using this flow.

Based on the actual range of the process, we enumerate the possible flows and determine if they are all $SAFE$. For example, if an output label u is assumed to span the security level range $L1$ to $L3$, but only processes at level $L3$ can send `unlabeled_t` data, then the flow need only be assessed for $L3$ output. We must classify each flow as either $SAFE$ or $UNSAFE$. A VM that has only $SAFE$ flows is information flow-safe, and all VM-system VMs must be information flow-safe in a compliant system.

Step 6. Verify compliance for local VM MAC policies. For VMs that process data at a range of security levels, we will need to verify compliance of the VM's MAC policy, even if it only has $SAFE$ flows. This is because there may be a non-compliant flow within the VM MAC policy that

violates the requirements for the inter-VM flows. For example, a VM may be authorized to receive low integrity data, but if it fails to sanitize the data before using it in a high integrity process that would violate compliance. Such non-compliance is a problem with the VM MAC policy, as the flow knowingly accepted low integrity data, but the VM policy allowed flows within the VM that do not comply with the security goal.

In the following paragraphs, we prove that evaluating a VM-system as a whole is equivalent to evaluating the parts and their interactions.

Theorem 1. A VM-system is compliant with a security goal if: (1) all the inter-VM flows are information flow compliant with the security goal and (2) all the VMs are compliant with the security goal.

Proof Sketch for Theorem 1: (By contradiction) Assume that a VM-system is compliant and either there exists a VM that is not compliant or there exists an inter-VM flow that is not information flow compliant. If a VM is not compliant then an information flow that is not authorized by the security goal is present in a VM, which violates compliance. If an inter-VM flow is not information flow compliant, then there exists an information flow that is not authorized by the security goal.

5. IMPLEMENTATION AND EVALUATION

In this section we present a framework that implements the analytical model we presented in section 4, and enable administrators to test compliance and other properties on the model. We encoded the model in Prolog, using the XSB Prolog implementation [6]. XSB has multiple advantages; it uses tabled resolution to improve performance, the encoding of the operators defined in the model is trivial in most cases, Prolog is ideal for implementing search algorithms, and to extend the implemented interface is easier than it would be with any other language, although it does require skills to program in prolog [21, 10, 6]. To evaluate our approach and its implementation, we check whether a VM-system running SELinux in the VMs, and XSM/Flask on the Xen hypervisor, meets a specific security goal. We first introduce Xen and XSM/Flask, and then we present a case study, its analysis and results.

5.1 Xen, XSM and XSM/Flask

XSM implements mandatory access controls (MAC) for the Xen hypervisor (VMM). Xen is a type I virtual machine monitor (VMM) that runs directly on the hardware. Operating systems running on Xen are paravirtualized to trap the sensitive operations into the hypervisor, so the hypervisor can enforce controls on these sensitive operations [18]. Xen presents these sensitive operations to the OS via a hypercall interface. This is similar to how the operating system presents operations to processes through system calls.

A VM is called a *domain* in Xen. *dom0* is a privileged VM for administration and hosts device drivers, though there is provision for driver domains. The XSM design is derived from Linux Security Modules (LSM). The LSM [23] approach inserts hooks in the Linux Kernel, in points that involve access to security relevant objects. XSM provides a set of hooks in the Xen kernel, to mediate access to security relevant Xen resources. Current security modules that can be linked at boot time with an XSM system are: Dummy (XSM default), ACM/sHype (IBM) [20], and Flask (NSA) [5]. An XSM/Flask module provides Xen the same kind of functionality that SELinux gives to Linux.

The resources that Xen provides are analogous to Linux: domains (processes), event channels (signals) and grant tables (shared memory). Sensitive operations on these are intercepted by XSM hooks and controlled by Flask policy. For example, sensitive operations on domains include creation and deletion. In this environment, Type 1 flows are caused through event channels and grant tables, which are the VMM abstractions used by VMs to communicate.

The XSM/Flask [5, 24] module uses the existing SELinux [17] policy language to specify the rules that define an XSM policy. To be able to analyze an XSM/Flask policy we studied its semantics and defined the mapping of XSM/Flask permissions to the `read_like` and `write_like` semantics. See [29] for more details.

5.2 Case Study - Integrity

This section presents a simple VM-system and its analysis.

EXAMPLE 5.1. *We have a VM-system with a privileged VM* `dom0_t`*, a service VM* `doms_t`*, and two user VMs* `domu_t` *and* `domv_t`*.* `dom0_t` *has access to all VMM resources, it runs back-end services that enable access from other VMs to VMM resources.* `doms_t` *runs a server that clients in* `domu_t` *and* `domv_t` *use. The server running on* `doms_t` *negotiates channels to communicate with (a) the client running on* `domu_t` *with label* `c2_t`*, and (b) the client running on* `domv_t` *with label* `c1_t`*.*

Table 1 shows excerpts of the XSM/Flask policy, the SELinux, and IPsec policies in the VM-system. The XSM/Flask policy defines flows among VMs. The table shows some of the permissions that `dom0_t` subjects have on `domu_t` resources, i.e., that belong to the domain `DomU`. The SELinux policy shows permissions assigned to an application running with label `server_t` on its own resources and resources labeled with `c1_t` (the label assigned to `domv_t` in the VM-system). The IPsec policy defines label and security features to assign to network connections between the indicated source and target points (`sr` and `tg` respectively). Note that the only VM-visible labels currently are defined in the IPsec policy, but that need not be the case for our model in general.

Figure 5 shows the graph with the information flows enabled by these policies (graph to the left). Logically, `doms_t` communicates with `domu_t` and `domv_t`, but such communications are implemented by `dom0_t`, so the actual system information flows are different. We will build the actual information flow graph below (Step 1). This figure also shows the information flows authorized by the security goal (graph to the right). This graph identifies the system's integrity levels and the authorized flows among them. We will need to map the system's labels to those levels for analysis (Step 2). Then, we will perform VM-system analysis in Steps 3-5 and verify that the VM policies are consistent with the VM-system in Step 6.

Step 1. Build VM Information Flow Graph. We build the graph with data from the XSM/Flask policy and the network policy. Since `dom0_t` implements network communication, all VM-visible labels connect a VM with `dom0_t` at that label. We also assume that each VM can send communications using its VMM label (e.g., Type 1 Flows). We also need to analyze the XSM/Flask policy to identify other

XSM/Flask Policy:

```
allow dom0_t domu_t:domain {create max_vcpus hypercall setdomainmaxmem setvcpucontext scheduler};
allow dom0_t domu_t:shadow {enable};
allow dom0_t domu_t:mmu {map_read map_write pinpage};
allow dom0_t domu_t:grant {query setup};
```

SELinux Policy:

```
allow server_t c1_t:tcp_socket relabelto accept read write connect;
allow server_t c1_t:association recvfrom sendto;
allow server_t server_t:file read write append create getattr setattr;
allow c1_t ipsec_spd_t:association polmatch;
```

IPsec Policy: (IPsec labeled)

```
spdadd <tg> <sr> any -ctx 1 1 "system_u:object_r:ipsec_spd_t:s0" -P out ipsec esp/tunnel/ <sr>-<tg> /req;
spdadd <tg> <sr> any -ctx 1 1  "system_u:object_r:ipsec_spd_t:s0" -P in ipsec esp/tunnel/ <tg>-<sr> /req;
```

Table 1: Excerpts of XSM/Flask, SELinux, and IPsec policies in a VM-system.

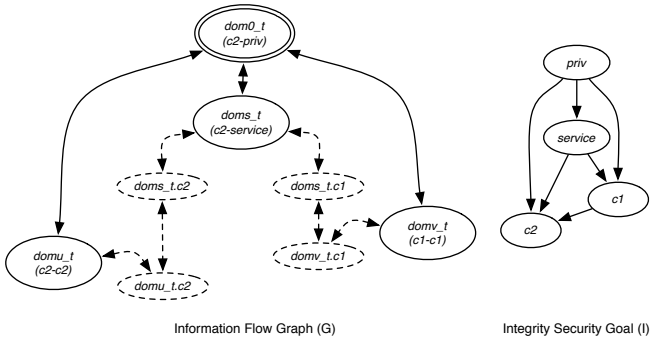

Figure 5: VM Information Flow Graph that represents the VM-system described in Example 5.1 and the security goal to evaluate integrity compliance. The solid and dashed arrows in the VM information flow graph represent Type 1 and Type 2 inter-VM flows respectively.

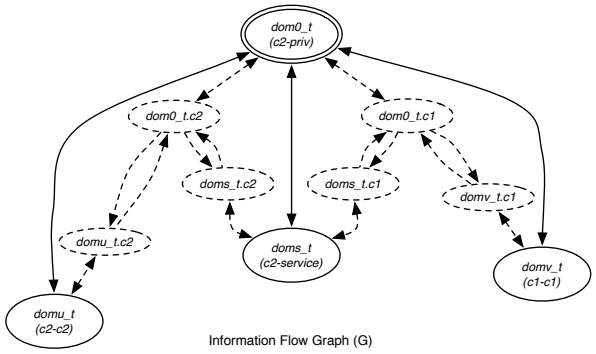

Figure 6: Actual Inter-VM Flows. The back-end servers that run in Dom0 implement network communication between processes that belong to different VMs. In the figure we split the flow between `domu_t.c2` and `doms_t.c2` into several flows (indicated by the dashed arrows) to represent this fact.

VMM resources that enable information flows between VMs, such as shared memory.

Figure 6 shows the graph. The solid arrows represent Type 1 inter-VM flows (enabled by the VMM policy). All VMs can communicate with `dom0_t` because it implements all back-end services, including networking which is the only means of communication between VMs in this example. The dashed arrows represent Type 2 inter-VM flows (enabled by the VM-visible labels). The labeled IPsec policy specifies: (1) the network connection between `doms_t` and `dom0_t`, for the server to receive messages at `doms_t.c1_t` and `doms_t.c2_t`; (2) between `domu_t` and dom0_t for this client to send and receive messages at `domu_t.c2_t`; and (3) between `domv_t` and `dom0_t` for this client to send and receive messages at `doms_t.c1_t`. In practice, the last two cases do not strictly require VM-visible labels (as the VM label is at the same integrity level), but this is shown for clarity.

Definition 5.1 (Supporting VM). *Supporting VMs are trusted to perform two security functions: (1) all the information flows from the supporting VM to any of its clients occur at the security level range of the client and (2) a supporting VM enforces noninterference on each input received from a client, so the supporting VM enables no information flows among client data.*

`dom0_t` is a *supporting VM*, multiple back-end services running in `dom0` have access to VMM resources and provide other VMs access to such resources. We represent the supporting VM with double line in Figure 6. In current implementations VMs access VMM resources only via `dom0`. However, Dom0 disaggregation [15] distributes these tasks among multiple VMs. In that case, our model would have multiple supporting VMs.

Step 2. Define Security Goal and Mapping Function. For this example, we describe the manual approach to specify goals and mapping functions. We define Prolog predicates `int_glevels`, `int_gedges` and `integrity` to specify an integrity security goal and map the labels in the policy to the levels in the goal. Even where specified manually, we could use a graphical interface to make the specification stage easier for administrators. The first two predicates define the integrity levels and the relation *can flow to* between the levels, respectively. The predicate `integrity` specifies the mapping function, it requires a resource label and the lowest and a highest security levels associated to the resource. Goal I in Figure 5 shows a graphical representation of the lattice defined by the predicates `int_glevels` and `int_gedges` below. The element $(priv, service)$ indicates that privileged domains, such as dom0, are higher integrity

Id	Channel	Enabled Flow	Result
1	dom0_t → doms_t	c2-priv → c2-serv	ambiguous
2	dom0_t → domv_t	c1* → c1	safe
3	dom0_t → domu_t	c2* → c2	safe
4	dom0_t ← doms_t	c2-priv ← c2-serv	ambiguous
5	dom0_t ← domv_t	c1* ← c1	safe
6	dom0_t ← domu_t	c2* ← c2	safe
7	domu_t.c2 → dom0_t.c2 → doms_t.c2	c2 → c2 → c2	safe
8	domu_t.c2 ← dom0_t.c2 ← doms_t.c2	c2 ← c2 ← c2	safe
9	domv_t.c1 → dom0_t.c1 → doms_t.c1	c1 → c1 → c1	safe
10	domv_t.c1 ← dom0_t.c1 ← doms_t.c1	c1 ← c1 ← c1	safe

Table 2: Evaluation of the Information flows represented in Figure 6. Flows from and to `dom0_t` are safe because Dom0 is a supporting VM (marked with *). Although the range for `Dom0` is `c2-priv` the back-end servers can establish connections at the level of the clients (in Flows 1-6). Flows 7-10 use VM-visible labels at a single security level to ensure safety.

than their client VMs. Likewise, $(service, c1)$ indicates that servers are higher integrity than their clients.

```
int_gllevels([priv,service,c1,c2]).
int_gedges([(priv,service),(priv,c1),(priv,c2),
(service,c1),(service,c2),(c1,c2)]).
integrity(dom0_t,c2,priv).
integrity(doms_t,c2,service).
integrity(domv_t,c1,c1).
integrity(domu_t,c2,c2).
```

The integrity ranges of these VMs indicate the security levels that the VMs are trusted to handle and the labels they may use to communicate. For example, `dom0_t` must be able to communicate to any VM in the VM system at the proper integrity level, therefore it has the range `c2-priv`.

Step 3. Verify compliance of the VM information flow graph. We check compliance of the VM information flow graph against the integrity goal. First, we classify flows into $SAFE$, $UNSAFE$, and $AMBIGUOUS$. If all the flows are $SAFE$ then we go to the next step. If we detect $UNSAFE$ flows, we report them and the system as noncompliant. $AMBIGUOUS$ flows require further analysis, described below. Table 5.2 shows a summary of the analysis for Example 5.1. The flows to and from `domu_t` and `domv_t` go via `dom0_t`, they are $SAFE$ because dom0 is a supporting VM (see Definition 5.1) and these VMs use only one integrity level and the supporting VM will reply in the client's integrity level. The flows involving the `doms_t` VM are $AMBIGUOUS$, as we cannot tell what integrity level is actually used by `doms_t` (further analysis in Step 5).

Flows from `domu_t` to `doms_t` via VM-visible labels are $SAFE$ because the data is conveyed from `domu_t` to `dom0_t` to `doms_t` using VM-visible labels at level $c2$ (defined by IPsec policy), so all intermediate flows are $SAFE$. The approach is similar for the flows from `domv_t`.

Step 4. Find Information Flow-Safe VMs. `domu_t` and `domv_t` are information flow safe, i.e. they are source and target for only $SAFE$ flows.

Step 5. Disambiguate Flows. For the $AMBIGUOUS$ flows, we analyze the local MAC policies of the involved VMs to determine the actual levels of the data sent via these flows. First, we determine the security levels of the processes, in the VMs, that can send data with the levels in the range associated to the flow. We identify two cases: (1) processes that may use `dom0_t` as a supporting VM and (2) processes that may send data labeled with `doms_t`. In the first case, the guarantees assumed by the supporting VM ensure that these communications are $SAFE$. In the second case, we identify the processes in `doms_t` authorized to send data labeled with `doms_t`. In SELinux, those processes are the ones with `create` or `bind` permissions on `tcp_sockets` with type `unlabeled` because an `unlabeled` communication defaults to the label of the VM. The services that run in `dom0_t` should have level `priv`, which is the higher integrity level, so the communication to `doms_t` is $SAFE$ (Table 5.2, row 1). The flow from `doms_t` to `dom0_t` is only $SAFE$ if the `dom0_t` processes use `service` level for receiving the communication and have satisfactory filtering interfaces for this data [22]. Since `dom0_t` provides services to less trusted VMs, it should have such filtering interfaces to protect its integrity.

Step 6. Verify local compliance for every VM. Since `domv_t` and `domu_t` are single level, we can exclude them from local compliance evaluation. The VMs that need to be evaluated are `dom0_t` and `doms_t`. In this case `dom0_t` is marked as a supporting VM, meaning it is capable of connecting to multiple clients at various integrity levels while enforcing noninterference between their information flows. A justification of supporting VM properties is future work. For `doms_t`, we follow the procedure described in Section 2 to evaluate its VM policy. In brief, we create an information flow graph based on the `doms_t` VM policy, and we test the graph against the system integrity goal.

5.3 Space Requirements

To evaluate the advantages of our approach to test VM-system compliance we estimate space requirements for the naive model, i.e., integrate the policies from all the VM-system components into a single information flow graph, and for our model. The information flow graph representation of the available XSM/Flask policy has around 15 vertices and 100 edges. The information flow graph of the SELinux reference policy has more than 2200 vertices and 200000 edges. The number of applications enabled by the SELinux policy to access network resources is around 420, this is the upper bound (per VM) of the number of possible information flows between VMs.

Therefore, the size of the naive model would be the size of the XSM/Flask policy representation, plus n times the size of the SELinux policy representation, where n is the number of VMs in the VM-system. In comparison, the size of our model, in the worst case, is the size of the XSM/Flask policy representation plus (n * 420) edges. In some cases we will need to independently test SELinux policies for VMs but we do not need to test all of these policies at the same time.

6. CONCLUSIONS AND FUTURE WORK

We presented a framework to analyze multiple, layered MAC policies being enforced in a VM-system. We observe that evaluating a VM-system as a whole is the same as independently evaluating (1) the interactions and (2) each VM policy. Therefore, we only combine the policies where inter-

action occurs. BY doing so we reduce the size of the problems to evaluate. We designed an analytical model based on this observation and implemented it in Prolog. We used the tool to check integrity compliance of a VM-system enforcing an XSM/Flask policy at the VMM layer, and multiple SELinux policies at the VM layer.

We plan to extend the model to represent more complex VM-systems. In the current model we assume a closed system, there is no communication with systems running outside of the VMM being analyzed. We will study the problem of integrating other VM-systems. Also, our model assumes that a single authority defines the security goals of the system, in a composite system that assumption must be eliminated and the model must handle (rectify) multiple goals provided by independent authorities.

7. REFERENCES

[1] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[2] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A System to Specify and Manage Multipolicy Access Control Models. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[3] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE, April 1977.

[4] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[5] G. Coker. Xen Security Modules (XSM). `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`.

[6] Computer Science Department of the Stony Brook University. XSB: Logic Programming and Deductive Database system for Unix and Windows.

[7] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, 1976.

[8] R. P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the AFIPS National Computer Conference*, 1973.

[9] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying Information Flow Goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[10] B. Hicks, S. Rueda, L. StClair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. In *Proceedings of (SACMAT)*, Antipolis, France, June 2007.

[11] T. Jaeger, K. Butler, D. H. King, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for Mandatory Access Control Across Systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, Aug. 2006.

[12] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. In *SIGMOD*, 1997.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the USENIX Technical Conference*, 2003.

[14] M. Koch, L. Mancini, and F. Parisi-Presicce. On the Specification and Evolution of Access Control Policies. In *Proceedings of SACMAT*, 2001.

[15] D. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the Virtual Execution Environments*, 2008.

[16] NetLabel - Explicit labeled networking for Linux. `http://www.nsa.gov/selinux`.

[17] Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[18] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7), July 1974.

[19] S. Rueda, D. King, and T. Jaeger. Verifying Compliance of Trusted Programs. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[20] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of ACSAC*, Washington, DC, USA, 2005. IEEE Computer Society.

[21] B. Sarna-Starosta and S. Stoller. Policy Analysis for Security-Enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004.

[22] U. Shankar, T. Jaeger, and R. Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium*, February 2006.

[23] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical Report 01-043, NAI Labs, 2001.

[24] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the USENIX Security Symposium*, 1999.

[25] Sun Microsystems. Trusted solaris operating environment - a technical overview. `http://www.sun.com`.

[26] Tresys. Policy management server.

[27] Tresys. SETools - Policy Analysis Tools for SELinux. Available at http://oss.tresys.com/projects/setools.

[28] C. Vance, T. Miller, and R. Dekelbaum. Security-Enhanced Darwin: Porting SELinux to Mac OS X. In *Proceedings of the Third Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, March 2007.

[29] H. Vijayakumar, S. Rueda, and T. Jaeger. Semantics of XSM/Flask Policies. Technical Report NAS-TR-0108-2009, SIIS Lab. Pennsylvania State University., 2009.