Networking and Security Research Center

Technical Report

NAS-TR-0151-2011

Towards System-Wide, Deployment-Specific MAC Policy Generation for Proactive Integrity Mediation

Sandra Rueda and Divya Muthukumaran and Hayawardh Vijayakumar and Trent Jaeger and Swarat Chaudhuri

29 September 2011

The Pennsylvania State University
344 IST Building
University Park, PA 16802 USA
http://nsrc.cse.psu.edu

Towards System-Wide, Deployment-Specific MAC Policy Generation for Proactive Integrity Mediation

Sandra Rueda
Universidad de Los Andes
ruedarod@cse.psu.edu

Divya Muthukumaran
Pennsylvania State University
muthukum@cse.psu.edu

Hayawardh Vijayakumar
Pennsylvania State University
huv101@cse.psu.edu

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

Swarat Chaudhuri
Rice University
swarat@cs.rice.edu

Abstract

Preventing attacks proactively in modern distributed systems is a major challenge. The addition of mandatory access control (MAC) enforcement in commodity software was supposed to prevent such attacks by limiting the number of processes accessible to adversaries and confining those still accessible. Unfortunately, the task of security professionals is still reactive, fixing vulnerabilities as adversaries identify them. We claim that in order to configure systems to defend themselves from attacks proactively, MAC enforcement must be customized to the target deployment. However, OS distributors currently focus on designing generic MAC policies for all their customers, leaving system administrators with the difficult task of composing and customizing these policies for their deployment manually. In this paper, we propose the Proactive Integrity Methodology, a mostly-automated method that computes system-wide MAC policies to prevent attacks proactively for distributed system deployments. We constructed a tool that implements this methodology to generate system-wide, Decentralized Information Flow Control (DIFC) MAC policies that require near-minimal effort to make an information flow safe system for Linux web application deployments in tens of seconds. System administrators can use such a tool to generate deployment-specific MAC policies for their distributed systems and verify whether OS distributions satisfy those policies, enabling proactive configuration to prevent attacks.

1 Introduction

Preventing attacks proactively in modern distributed systems is a major challenge. Modern distributed systems often consist of multiple interconnected hosts running several application processes on complex platforms of system software. Many application and system processes are accessible to legitimate and adversarial users alike (e.g., via the network), providing entry points for attack. In web application deployments, web server processes and network-facing daemons are all accessible to adversaries. But, this is just the start of the challenge, as applications and system processes may also propagate data to several other processes and even to other hosts. For example, web servers propagate requests to any number of web application programs, any of which can be used to launch local exploits against the system software if compromised (e.g., to install a rootkit) or can be used to propagate untrusted data to others if spoofed (e.g., SQL injection).

To control the propagation of attacks within and across hosts in a distributed system, mandatory access control (MAC) enforcement [18, 3, 30] has been integrated into a variety of commodity software, including operating systems [64, 58, 63], virtual machine monitors [13, 50, 29] (VMMs), user-level programs [60, 38, 65], and integrated with network access control [34, 22, 41]. OS MAC enforcement aims to prevent attacks by reducing the number of processes that are accessible to adversaries and confining those accessible processes to restrict the propagation of untrusted data. However, MAC enforcement at the operating system is not enough to prevent attacks because: (1) programs are often entrusted with data of multiple

security requirements (e.g., belonging to multiple users); (2) operating systems are too complex to satisfy the requirements of the *reference monitor concept* [3]; and (3) MAC enforcement in distributed systems must be integrated among multiple hosts. The MAC enforcement in several user-level programs, VMMs, firewalls, and network communication mechanisms has been developed to supplement OS MAC enforcement to fill these gaps, providing the ability to enforce mandatory access control system-wide.

However, security practitioners do not appear to be reaping the benefits of all these MAC enforcement mechanisms, as the task of security professionals is still reactive, fixing vulnerabilities as adversaries identify them. We claim that the problem is that the current approaches for configuring MAC enforcement do not account for the specific requirements of system deployments. As defining MAC enforcement policies is a complex undertaking, OS distributors have taken responsibility for configuring these policies for their system distributions, including user-level and VMM policies. However, OS distributors must define MAC policies that permit the operations required in a wide variety of customer deployments. Distributed systems integrate several generic, independently-developed MAC policies together, creating many unpredictable paths for propagating attacks through systems. Automated methods to configure systems to prevent such attacks either only consider one component's MAC policy [36, 59, 23, 53, 11] or build system-wide attack graphs from heuristic information rather than MAC policies [44, 56, 39], leading to incomplete analyses.

Instead, what we want is an automated approach to generate system-wide MAC policies that protect system deployments proactively. A system administrator should be able select the software for their deployment (i.e., including preconfigured MAC policies) and configure network policies (i.e., connecting the components together), enabling an automated method to determine whether the combination proactively defends all threats and, if so, generate a system-wide MAC policy that enforces those defenses. This may sound like a daunting task, but the information necessary to solve such a problem is now available, partly due to the efforts of OS distributors to configure default MAC policies for these various enforcement mechanisms. Our key insight is that the distribution MAC policies actually define the expected functionality of a system, whereas the deployment specifications identify the security requirements and threats. We use the security requirements and threats to compute where defenses are necessary to protect the distributed system's integrity while preserving necessary function.

In this paper, we develop the Proactive Integrity Methodology, a mostly-automated approach for generating system-wide MAC policies for distributed system deployments. We solve two major challenges in creating such a methodology. First, we need to produce comprehensive information flow models of distributed system deployments automatically. As discussed above, distributed systems often consist of several components, many with MAC enforcement that was developed independently, but these must be composed into a single model to detect whether any unauthorized information flows (i.e., possible attacks) are allowed. Recently, analyses have been developed to find such information flow errors in MAC policies [45, 61, 23, 53, 11] and programs [36], but these methods require manual input to identify security constraints and connect multiple policies, making them impractical for use in distributed system deployments. We show that comprehensive information flow models can be composed automatically for deployments from available security policies and basic deployment knowledge.

Second, often the functional requirements of systems result in information flow integrity errors, creating a difficult manual task of resolving such errors. Recently, researchers have developed methods to automatically compute near-optimal resolutions of information flow errors [25, 45], which we apply to generate system-wide MAC policies using the Decentralized Information Flow Control (DIFC) model [27]. However, these proposed resolution methods have only been applied to single policies with simplistic resolution requirements, so a number of challenges result, such as determining a practical definition of "near-minimal" for distributed systems and combining the solutions to several individual resolution problems.

This research makes the following contributions.

- We develop a method to produce a *system-wide information flow integrity model* to evaluate whether

the data flows authorized in a distributed system prevent illegal information flows.

- We develop a graph-cut method to *resolve information flow errors* for these information flow models.
- We built a tool *produce DIFC-Flume MAC policies* [27] mostly-automatically, demonstrated on Xen hypervisor hosts [7] with application VMs using SELinux distributions [42].

Using this tool, we can determine whether a legal MAC policy exists, generate system-wide MAC policies based on deployment conditions, and identify the integrity decisions that each program is expected to make to protect the distributed system from attack proactively.

The remainder of the paper is structured as follows. In Section 2, we motivate the problem of generating system-wide MAC policies for distributed systems and define the problem as one of finding a MAC policy that satisfies functional and security constraints of a deployment. In Section 3, we outline the Proactive Integrity Methodology and detail the designs for the technical tasks in the methodology. In Section 4, we evaluate the effectiveness of an implementation of the methodology and the efficiency of its computing tasks. Finally, we outline prior work in Section 5 and conclude in Section 6.

2 Problem Definition

2.1 Example System

Figure 1 shows a simplified web application deployment, our sample distributed system. Our web applications are so-called LAMP systems, consisting of (Apache) web servers, (MySQL) databases, and various (Perl/PHP/Python) web application programs running on one or more (Linux) operating systems. In Figure 1, a web server application and DB each run in separate guest VMs. The purpose of this system is for the web application to respond to requests from authorized clients, where such requests may require querying or updating the database. The guest VMs are deployed on a host system consisting of a virtual machine monitor (VMM) and its privileged VM, such as the Xen hypervisor [7] and its domain 0 host, respectively. Such systems may also depend on the integrity of network servers to prevent network attacks, such as DNS hijacking. Such a deployment would be analogous to many server systems as well as some utility computing environments, such as a IaaS clouds (public or private).

While the web server provides access to the web application for external clients, it also provides a means for adversaries to launch attacks against the web application and the system at large (e.g., [52]). Even if the web server protects itself from compromise, it forwards data to other web application components (e.g., PHP scripts and databases) that may compromise those components (e.g., through input validation and SQL injections). Compromised web application components may then launch local exploits against a variety of system processes. Finally, a compromised system hosting a web server or application component can then launch attacks against the host's virtualization infrastructure and other hosts. Of course, web applications are just one of the many networked applications and services to worry about.

Traditionally, it has been the responsibility of system administrators to configure security policies for their system deployments, but the complexity of systems and of the security policies themselves has made this task impractical. To ease the system administrators' job, several OS distributions are now deployed with their own mandatory access control (MAC) policies [42, 40, 63, 35], including MAC enforcement in user-level programs [62, 46, 38, 65, 10] and VMMs [50, 13, 48], which administrators now depend upon for the security of their deployments. However, while some attacks are being prevented, system administrators and OS distributors are still reacting to attacks, responding to adversary exploits as they are released.

We claim that the reason for this situation is twofold. First, the distribution MAC policies are not specialized to prevent attacks in a deployment, because distribution MAC policies permit the functionality required of many deployments. While OS distributors often consider distinct deployments in the design of distribution MAC policies, these MAC policies must permit the operations necessary for any reasonable customer deployment to function properly. Readers may remember that when SELinux was introduced, many found it to be unusable because the policy was too restrictive to run their desired programs effectively

(or even boot the system). As a result, distribution MAC policies often permit far more permissions than are required in a specific deployment (see Section 4), leading to attacks that are unexpected when composed into distributed systems.

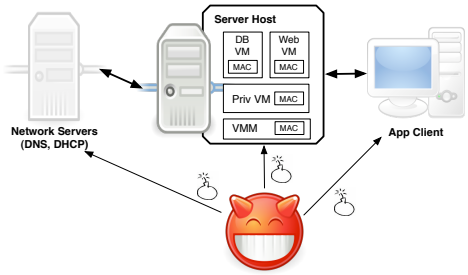

Figure 1: End-to-end web application system

Second, in designing distribution MAC policies, OS distributors often must assume that programs securely handle any untrusted inputs that they may receive. However, there is often a mismatch between the program interfaces (e.g., system calls) that may receive untrusted input in a deployment and the program interfaces that are designed to defend against untrusted input. For example, logrotate is a utility to perform log rotation that has long existed on Unix systems. While it was well known that the log files themselves might contain untrusted input, the fact that the log filenames are also attacker-controlled was overlooked until recently (CVE-2011-1155), allowing an attacker to deny service by inserting special characters in such filenames.

Instead, what a system administrator would like is a system-wide MAC policy that prevents attacks proactively for their deployment. When a system administrator chooses the deployment software (including their default MAC policies for those components that perform MAC enforcement) and configures the software and network policy, she would like a MAC policy that identifies how all threats created by such a deployment are defended, preferrably in terms of classical information flow integrity [9, 8]. We envision that this can be accomplished in two steps. First, OS distributors design their MAC policies to account for program interfaces that must defend the programs from attacks in a variety of deployments. Second, system administrators generate system-wide MAC policies for their specific deployments that are consistent with the program defenses to proactively defend their distributed systems from attack. Interestingly, both of these tasks correspond to the same conceptual problem, automatically generating system-wide MAC policies with near-minimal number of resolutions for information flow errors, as described below.

2.2 Policy Generation Problem

The general problem is to take the existing *MAC policies* of the software components in a system deployment and a *goal policy* that describes the legal and necessary operations in that deployment to generate a *system-wide MAC policy* that minimizes the *mediation* that components must perform to satisfy the goal policy. To model this problem, we suggest using the information flow approach used in several program and system policy analyses [36, 23, 53, 26, 45, 11, 61], where: (1) component MAC policies are converted to data flow graphs representing the authorized information flows in each deployment and (2) the legal operations in goal policies are specified in terms of lattice integrity policies [9], consisting of *integrity levels* and the authorized information flows among them. Source and sink nodes in the data flow graph are assigned to integrity levels in the lattice policy, enabling identification of information flow errors: nodes that receive input data of integrity levels lower than the node's integrity level.

The aim is to generate a system-wide MAC policy that contains no information flow errors, thus preventing all operations that do not satisfy the lattice policy. This can be achieved by either: (1) removing data flow edges in the graph until no information flow errors remain or (2) adding *mediation* that changes the level of data output by a mediating node to resolve information flow errors. Removing data flow edges may not be permissible because those edges may imply operations necessary to provide system function. Adding mediation is not free however, as mediation must be implemented by new processes (e.g., guards) or adding (or acknowledging) mediation abilities in existing programs (e.g., transformation procedures [12]). Thus, our aim is to minimize mediation while satisfying information flow integrity.

This policy generation problem is computationally complex because both information flow integrity (negative) constraints and functional (positive) constraints must be satisfied simultaneously. There are typi-

cally two approaches taken to solving such problems: (1) solve for the set of constraints using off-the-shelf SMT solvers [15, 24] or (2) find a satisficing, although potentially sub-optimal, solution using a greedy method. While it may be possible to encode our problem for an SMT solver, a key challenge is that the functional constraints of a system deployment are difficult to predict. Instead, we develop a greedy method that produces near-minimal mediation to satisfy information flow integrity given a particular function specification. Users can choose the expected functional requirements (e.g., using other analyses, see Section 4) to explore their impact on mediation. We note that commodity MAC policies represent functional requirements, often the upper bound of system function, as they were created to satisfy all legitimate deployments.

Developing a greedy solution has several challenges that we address in this paper. First, the multiple, independent MAC policies in a system must be composed into a single, coherent model. For example, the integrity of the example web application depends on the integrity of the MAC policies of the web server, database, and privileged VMs, as well as the VMM MAC policy and the MAC policies of some network servers. Second, often information flow requirements are not explicitly specified in commodity system deployments, requiring researchers to specify constraints manually. However, manual specification of system-wide information flow integrity is not practical for system administrators. Third, once a solution is found must be converted into a system-wide MAC policy. However, commodity MAC policies only express authorized operations, not mediation, so we cannot simply convert the result back to a commodity MAC policy without losing information.

3 Proactive Integrity Methodology

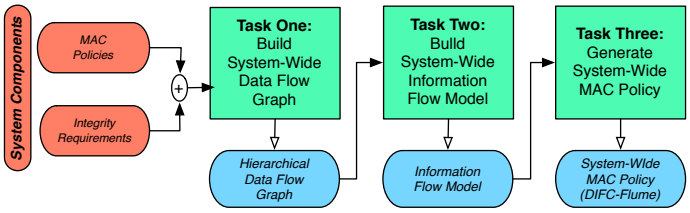

Figure 2: Proactive Integrity Methodology Tasks

Figure 2 shows an overview of our Proactive Integrity Methodology. This methodology generates system-wide MAC policies for distributed systems composed from a set of independent software components by solving three tasks. The first task automatically builds a *system-wide, data flow graph* using the components' internal and external MAC policies, as described in Section 3.2. This hierarchical data flow graph represents all the possible communication paths in the system. The second task annotates the data flow graph generated in the first task with security constraints necessary to create a *system-wide, information flow integrity model*, as described in Section 3.3. In particular, this task generates an integrity lattice policy for the distributed system and maps the integrity levels in that policy to the data flow graph. The third task applies a graph-cut method to generate a *system-wide MAC policy* that complies with the integrity lattice policy of the information flow integrity model, identifying where exceptional mediation is necessary to prevent information flow errors automatically, as described in Section 3.4. Using this solution, this task generates a system-wide MAC policy in the Decentralized Information Flow Control (DIFC) model of the Flume system [27].

The MAC enforcing software components in the distributed system are the input to this methodology, where each component is defined by the following fields:

- **Name**: Unique name of this component in the distributed system
- **Parent**: Unique name of the parent of this component
- **MAC Policies**: Consisting of internal and external MAC policies, where: (1) an *internal MAC policy* governs operations for subjects and objects internal to the component and (2) an *external MAC policy* governs how subjects and objects internal to this component communicate with external components
- **Integrity Requirements**: Consisting of the component role, target application, and integrity dependencies, where: (1) the *component role* identifies the component's role in protecting system-wide

integrity; (2) the *target application* identifies MAC policy subjects and objects that must be integrity-protected; and (3) *integrity dependence* defines relations between a MAC policy subject or object and another MAC policy subject or external component that must protect its integrity

Each component in a distributed system can be uniquely identified by its *name*, so other components can refer to it. Currently, we use IP addresses for names of operating systems (and VMs) and VMMs are associated with their physical machines. Each component has a single *parent* component, except for the root component, that defines its placement in the hierarchical data flow graph (see Section 3.2). The component *MAC policies* define all the authorized operations that component permits its subjects to perform on its objects (i.e., internal MAC policy) or upon other components in the distributed system at large (i.e., external MAC policy). For an OS distribution, the distribution MAC policy is an internal MAC policy and the network policy is an external MAC policy[1]. Finally, a component's *integrity requirements* identify its own integrity-critical data and the relative integrity of that data. The target applications of the component identify the subjects and objects in the component's MAC policy that require integrity protection. The component's role and integrity dependence requirements specify relative integrity relationships. We augment this specification with a built-in knowledge base to infer each system's integrity lattice policy, as described in Section 3.3.

3.1 Assumptions

The key assumption in this work is that the programs, operating systems, and VMMs that enforce MAC policies or provide mediation do so correctly. This is a significant assumption given the size and complexity of such software, but this work's focus is on how one provides and verifies the *placement* of defenses that would proactively protect the integrity of entire distributed systems. Specifically, we assume that the programs, operating systems, and VMMs satisfy the *reference monitor concept* [3], which requires that a reference validation mechanism (i.e., MAC enforcement) must "must always be invoked" upon a security-sensitive operation, "must be tamperproof," and must be "small enough to be subject to analysis and tests, the completeness of which can be assured," which implies correctness. The reference monitor concept is certainly the goal of several commodity reference validation mechanisms, although they do not meet the letter of these requirements. Also, our method can help improve tamperproofing by protecting the resources critical to MAC enforcing components.

Also, our methodology assumes an interaction between OS distributors and programmers, where OS distributors use our method to find where mediation is necessary in programs and programmers will supply effective mediation code. At present, such a dialogue is ad hoc, but we envision that such a method may provide concrete motivation for programmers to improve mediation. Our method does not specifically depend on whether it is the programmer or the OS distributor who drives the process of adding mediation to address deployment threats, however.

Finally, current commodity MAC policies cannot express mediation, so we instead leverage "practical" integrity models that define MAC policies that express how programs use mediation to resolve information flow errors [54, 28, 57, 27, 67]. In particular, our method generates MAC policies using the DIFC-Flume model [27]. Thus, we assume that a commodity system that is capable of enforcing the DIFC-Flume policy model. Since the Flume system was implemented as an extension to a commodity system (Linux), we believe that such an assumption is acceptable.

3.2 Constructing Hierarchical Data Flow Graphs

In the first task, we develop a method that uses the MAC policies for all the MAC-enforcing components in a distributed system to construct a hierarchical data flow graph that represents all the authorized operations in the distribution system.

[1] We also use external MAC policies to describe which OS processes may use which VMM hypercalls in a virtualized system.

A *hierarchical state machine* K is a tuple $(K_1, ...K_n)$ of *modules*, where each module K_i has the following components:

- A finite set V_i of *nodes*.
- A finite set B_i of *boxes*.
- A subset I_i of V_i, called *entry nodes*.
- A subset O_i of V_i, called *exit nodes*.
- An *indexing function* $Y_i : B_i \rightarrow \{i+1, ..., n\}$ that maps each box of the i-th module to an index greater than i. That is, if $Y_i(b) = j$ for box b of module K_i, then b can be viewed as a reference to the definition of module K_j.
- If b is a box of the module K_i with $j = \tilde{Y}_i(b)$, then pairs of the form (b, u) with $u \in I_j$ are the *calls* of K_i and pairs of the form (b, v) with $v \in O_j$ are the *returns* of K_i.
- An *edge relation* E_i consisting of pairs (u, v), where the source u is either a node or a return of K_i and v is either a node or a call of K_i.

Figure 3: Encapsulated hierarchical graph definition for Hierarchical State Machines [2] (HSMs)

3.2.1 Hierarchical Data Flow Graphs

Prior work has defined methods to convert individual programs [36, 16] and system MAC policies to data flow graphs to find information flow errors [45, 60, 53, 23, 11]. In a MAC policy, authorized operations can be classified as read-only, write-only, and read-write, between subject labels and object labels in the policy[2]. From this classification, a directed data flow graph can be constructed where the nodes are the subject and object labels and the edges are the flows resulting from the authorized operations.

Distributed systems often consist of multiple components that enforce independent MAC policies, so there is a need to build a unified data flow graph from multiple MAC policies that represents all authorized data flows and enable optimizations to reduce unnecessary graph complexity. We find that distributed systems consist of a set of encapsulated software components that mediate all the operations of their subjects, and that these components are arranged hierarchically, where processes run on operating systems, which may run on VMMs, which may be a host in the network. Figure 3 shows the definition for the encapsulated hierarchical model we use, which is the graph model used in *hierarchical state machines* [2] (HSMs). Algorithms have been designed that solve a variety of computing problems using this model, such as reachability, equivalence, and inclusion [1].

The key concept in an HSM graph is a *module*, which represents a single subgraph of nodes and edges within an HSM graph. Each component's internal MAC policy is represented by a module. Subjects and objects that do not enforce a MAC policy are represented as nodes in a module. The other important feature of the HSM model is its flexible model for expressing data flows between modules. *Boxes* describe independent data flows from a parent node to a child node through calls and returns. Using boxes, an external MAC policy can be converted to calls and returns to boxes in the graph precisely.

3.2.2 Constructing HSM Graphs

The overall approach is to create an HSM module for each internal MAC policy, which covers all of the information flows authorized by those policies. Then, we connect the HSM modules using the external MAC policies, which govern interaction between components. We demonstrate the generation of inter-module connections using the network policy shown in Table 1, but other policies, such as the informal policy governing access to security-sensitive VMM hypercalls, also enable connections.

Create the Root HSM Module. The root module consists of data flows among the subjects and objects in the internal MAC policy of the root component. For a distributed system, the root policy is a network policy, and for a single host the root policy is a VMM or OS MAC policy. For a network policy, the root module includes a node for each IP address and a directed edge (u, v) if node u is authorized to send to node v. For a MAC policy, a node is a subject or object label in the policy, and a directed edge (u, v) is added for each subject u that can write to object v or each object u that can be read by subject v. In both cases, the root module represents the authorized data flows among components.

Create a Child HSM Module. If one of the nodes in the root HSM module enforces its own MAC policy[3] (VM or host), then a child HSM module will replace this node. The internal nodes and edges of

[2] In this paper, we use *label* for the MAC policies (e.g., SELinux and DIFC-Flume) and *level* for the lattice nodes, as in the information flow definition of Section 3.3.1.

[3] A subnet policy is also possible below a root network policy.

VM Policy:

```
allow httpd_t http_server_packet_t:packet send;
allow httpd_t http_server_packet_t:packet recv;
```

Network Policy: (iptables)

```
iptables -t mangle -A INPUT -p tcp --dport 80
  -d 130.203.32.56 -j SECMARK
  --selctx system_u:object_r:http_server_packet_t:s0
iptables -t mangle -A OUTPUT -p tcp --sport 80
  -s 130.203.32.56 -j SECMARK
  --selctx system_u:object_r:http_server_packet_t:s0
```

Table 1: SELinux and iptables Secmark policy excerpts. The SELinux policy specifies the subjects that have access to network communication channels and to what channels, while the network policy specifies the authorized communication channels.

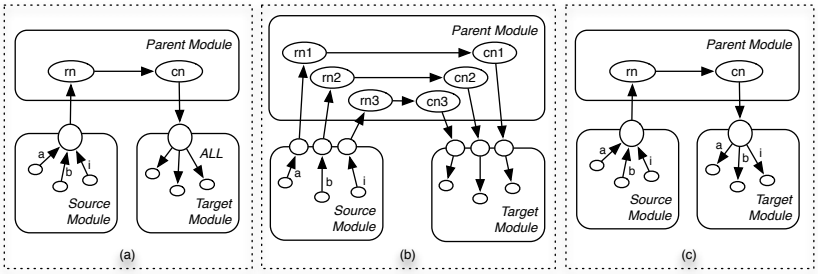

Figure 4: Representing information flows between nodes in the HSM model by: (a) merging all security requirements; (b) keeping the security contexts separate via separate flows; and (c) depending on nodes to demultiplex security contexts.

a child HSM module will be created in the same manner as for the root module, but additional nodes and edges are necessary to connect this child to its parent.

To determine the connections, we use the external MAC policies, such as the labeled network policy of the VM or host. For a particular network policy rule, an entry (exit) node is created in the child module and the child's internal nodes representing its subjects with access to that network communication are connected to that entry (exit) node. In the parent module, a box is created that is connected to the entry (exit) node. Nodes in the parent module that want to communicate with the child must perform a call (box and entry node) and return (box and exit node).

Table 1 shows excerpts of Secmark network [34] and SELinux MAC [42] policies associated to a web server VM. The Secmark network policy would cause entry and exit nodes to be created for communications to and from 130.203.32.56, respectively. The SELinux policy authorizes the internal node for the `httpd_t` subject to read from that entry node and write to that exit node. A call and return are created in the parent module for the box associated with this module to send and receive communications for host 130.203.32.56. If the client also has a module, then the returns and calls would be linked together in the parent module. There are some options for doing this, which we discuss below.

Connecting through Parent Modules. A question is how to connect an entry (exit) node in one module to a call (return) for another module through a parent. The naive way is to use the edge between hosts authorized by the parent's network policy, as shown in Figure 4(a). However, as we can see, this would mix all of the output data in the single edge, so all the host's transmitted data would reach all the destination processes, rather than providing the data with the expected integrity to each. The HSM model enables these flows to be separated by having a distinct flow from host's call or return to the destination's call or return in Figure 4(b).

The latter approach preserves the security context for the distinct communications, but the downside is that it may create many edges. When we consider a VM system, such as Xen, which has all networking redirected to a privileged VM, many edges with the same destinations will be created to keep the security information distinct. Our aim is to multiplex data of multiple integrity levels[4] where nodes are simply transporting information with distinct security contexts and demultiplex the data at nodes entrusted to do such

[4]The data flow graph is used to propagate data and their integrity levels, which is made clear in Section 3.3.

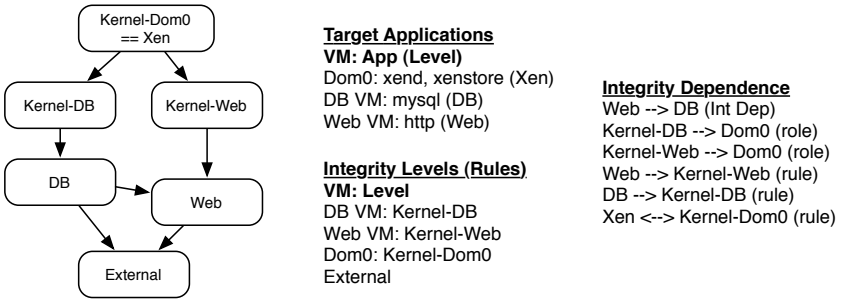

Figure 5: Integrity Requirements and Resulting Integrity Lattice for Web Application Example

enforcement (e.g., OS kernel). Figure 4(c) shows that a single path can be created between the two modules for transmitting data with multiple integrity levels, but the flows from the receiver's entry node are annotated with the demultiplexed levels before being forwarded to internal module nodes. Such a mechanism corresponds to network demultiplexing in the kernel and/or privileged VM.

3.3 Constructing Information Flow Integrity Models

In the second task, we develop a method that uses the data flow graph generated in the first task and the components' integrity requirements to construct an information flow integrity model of the distributed system sufficient to detect information flow errors.

3.3.1 Information Flow Model

In this work, we use the information flow model[5] defined below:

1. A *directed data flow graph* $G = (V, E)$ consisting of a set of nodes V connected by edges E, as defined by the HSM data flow graph in Figure 3.
2. A *lattice* $\mathcal{L}$= $\{L, \preceq\}$. For any two levels $l_i, l_j \in L$, $l_i \preceq l_j$ means that l_i 'can flow to' l_j.
3. There is a *level mapping function* $M : V \rightarrow \mathcal{P}^L$ where $\mathcal{P}^L$ is the power set of L (i.e., each node is mapped either to a set of levels in L or to $\emptyset$).
4. The lattice imposes security constraints on the information flows enabled by the data flow graph. Each pair $u, v \in V s.t. [u \hookrightarrow_G v \wedge (\exists l_u \in M(u), l_v \in M(v).\ l_u \not\preceq_{\mathcal{L}} l_v)]$, where $\hookrightarrow_G$ means there is a path from u to v in G, represents an *information flow error*.

It has been shown that programs can be automatically translated into such a model [37], and we show that a MAC policy can also be translated into the model automatically.

The main challenges are to determine the integrity lattice and the level mapping function for a distributed system. In single policy analyses, such information must be specified manually. While it may be possible to identify integrity requirements within a single MAC policy, we envision that OS distributors, and especially system administrators, will need assistance in determining the integrity lattice policy and level mapping function implied by the composition of independent MAC policies in a distributed system.

3.3.2 Inferring Lattices and Level Mappings

Pre-defined rules use components' *integrity requirements* (see Section 3) to create integrity levels, define the level mapping function to map integrity levels to nodes in the data flow graph, and identify the information flows that should be authorized by the system's integrity lattice policy. Using the method described below, the integrity lattice for the web application example system is shown in Figure 5.

Integrity Levels. The components' user-supplied *target applications* identify subjects and objects in a MAC policy that require integrity protection and the name of the level. A new integrity level is created for each target application (and component). For example, integrity levels are created for the Apache web server

[5]We use this information flow model to reason about integrity only.

(level Web in Figure 5), MySQL database in the web application example (DB), and Xen services in Dom0 (Xen). Our methodology also includes pre-defined *mapping rules* that automatically identify integrity levels that are present in all deployments, such as kernel levels (e.g., Kernel-Web, Kernel-DB, and Kernel-Dom0) and assign subjects and objects to those levels. Since Xen services can write kernel objects, they are also assigned the Kernel-Dom0 integrity level. The resultant level mapping function assigns the new integrity levels to the nodes corresponding to the subjects and objects.

Integrity Dependence. Once the integrity levels are identified, it is necessary to determine relative integrity of the data of each level. We use a combination of pre-defined *integrity dependence rules* and user-supplied *component role* and *integrity dependence* statements to determine the dependence among integrity levels shown in Figure 5. First, pre-defined dependence rules leverage well-known integrity relationships. For example, the hierarchical ordering of the components in the hierarchical data flow graph already impose an integrity dependence. For example, VMMs must have higher integrity than VMs. User-defined component roles define trusted services, such as privileged VMs, which are higher integrity than guest VMs (e.g., Kernel-DB and Kernel-Web depend on Kernel-Dom0). Similarly, every application depends on its OS's trusted computing base. Therefore, Kernel-DB, Kernel-Web and Kernel-Dom0 have the highest integrity in their respective applications. Finally, users may specify additional deployment-specific integrity dependence statements in the component definition. In our example, the Apache web server depends on the MySQL database (to isolate web application data), so an integrity dependence is created between the levels assigned to them. If levels are not assigned to both nodes, then an information flow edge is created from the protector to the dependent node.

Integrity Lattice. The integrity lattice consists of the integrity levels and the authorized information flows among them (see Section 3.3.1 for a formal definition). The authorized information flows identify cases where the data of one integrity level may flow to data of another level, defined by the integrity dependence relations. Otherwise, integrity levels are assumed to be incomparable, which preserves a conservative interpretation of the security (i.e., only information flows that are explicitly approved are allowed). An External level is always included as System-Low, representing data that enters the distributed system that is unlabeled.

3.4 Resolving Information Flow Errors

In the third task, we develop a graph-cut method to generate system-wide MAC policies that comply with the integrity lattice policy of the information flow model using near-minimal number of mediators.

3.4.1 System-Wide MAC Policies

All system-wide MAC policies generated by our methodology must comply with the integrity lattice of the information flow model. Recall that a MAC policy is represented by the data flow graph of the information flow model, so such compliance is obtained only if the data flow graph only permits information flows that are authorized in the lattice policy or has mediation that prevents information flow errors. In our methodology, we do not change the possible data flows, but instead generate near-minimal mediation necessary to eliminate all information flow errors automatically.

We define a system-wide mediation for the lattice $\mathcal{L}$ for data flow graph G as follows: for a path $u \hookrightarrow_G v$ where $l_u \not\preceq_{\mathcal{L}} l_v$ is an information flow error, if a node n on that path is a mediator, we perform level manipulation such that data on any outgoing edge from the node n has some level l' such that $l' \preceq l_v$ (i.e., the integrity level of the data leaving the node n is at least l_v). This resolves the information-flow error.

The resultant system-wide MAC policy consisting of data flows and mediators is converted to the Decentralized Information Flow Control (DIFC) defined for the Flume system [27]. We show in Appendix A that the Flume model is equivalent to the information flow model in Section 3.3.1. In short, levels in our model correspond to *labels* in the DIFC-Flume model, where DIFC-Flume labels represent a set of satisfied integrity requirements (i.e., the set of levels dominated by or equal a particular level in our model). Mediation in our model corresponds to *dual capabilities* in the DIFC-Flume model, where a dual capability

```
1  MEDIATIONRESOLUTION(G, L, M, MaxRaise) {
2    LS ← TopologicalSort(L)
3    for (l ∈ LS)
4      do Sources ← {l_i ∈ L | l_i ⋠ l}
5         G' ← GraphCopy(G)
6         AddSuperSource(G', Sources,' SuperSource', M)
7         AddSink(G', l, M)
8         SR ← SliceReachable(G',' SuperSource', l)
9         AdjustWeights(SR, l, MaxRaise)
10        Mediators ← Mediators ∪
11          (l, MinimumCut(SR,' SuperSource', l))
12        }
```

Figure 6: Mediation Resolution Algorithm

permits a subject to add (*endorse*) or remove (*downgrade*) an integrity level. We note that automatically generating system-wide MAC policies in DIFC-Flume for commodity distributed systems is a significant result by itself, as to date Flume policies have been developed manually.

3.4.2 Computing Mediation Placements

Researchers had the insight that placing a mediator that resolves an illegal information flow between two levels l_i and l_j is tantamount to generating a vertex cut[6] of the information flow graph with the nodes mapped to l_i as the sources and the nodes mapped to l_j as the sinks [25, 45, 26]. This property is called *Cut-Mediation Equivalence*. We extend this solution to general lattices, as required for this information flow model. We also customize the solution to account for components that have limited mediation abilities (i.e., may be able to mediate some, but not all errors).

Given a graph G=(V, E) and a lattice $\mathcal{L}$=$\{L, \preceq\}$, there is a *cut problem* when there is an information flow error between two nodes with the level mapping function M. A *cut problem set* consists of the information flow errors, as defined in Section 3.3.1. Researchers have shown that the *multicut problem*, the problem of find the minimal cut set for a cut problem set, is NP-Hard [14]. They proposed a simple greedy solution to the problem that returns the *union* of the solutions for each individual cut problem.

To improve on the simple greedy approach, we use the insight that classical integrity encourages processes to upgrade input integrity to their level [12]. We note that each node has a limit regarding how high it may upgrade that we call its *maxraiselevel*. A node's *maxraiselevel* is limited to its own level, if mapped, or that of its closest ancestor that is mapped in the hierarchical graph. The graph-cut procedure produces a *raiselevel* l' for mediators which is at most *maxraiselevel*. We use these raiselevels to define a property called *Mediation Dominance* [5]. This property states that solving a cut problem in graph G for level l_1 may solve any overlapping problem in the same graph G for a level l_2 if $l_1 \preceq l_2$. The intuition behind this is that since l_1 is higher integrity than l_2, the semantics of mediation implies that any mediators that can mediate for l_1 can automatically mediate for l_2. Therefore, by solving the cut problems for l_1 before l_2, we get two advantages: (1) we may solve a smaller problem for l_2 (compared to solving the problem for l_2 independent of l_1) since *mediation dominance* enables us to remove the mediators computed for l_1 and any flows they fostered before solving the problem for l_2 and (2) if there is an overlap in the graph between the different cut problems of *comparable* lattice levels, then the size of ordered cut solution can be smaller than the naive solution, a *union* of the individual solutions.

We propose an algorithm, MEDIATION RESOLUTION, that solves cut problems of graph G for the lattice $\mathcal{L}$, whose pseudocode is shown in Figure 6. The algorithm receives a data flow graph G, a lattice $\mathcal{L}$, the mapping M of nodes to levels, and the *maxraiselevels* for the mapped nodes $MaxRaise$. Line 2 first sorts

[6]In graph theory, given a graph G=(V,E), a vertex cut of this graph with respect to a source and a sink is a set of vertices whose removal will divide the graph into two parts, one containing the source and the other containing the sink, such that the sink can no longer be reached from the source.

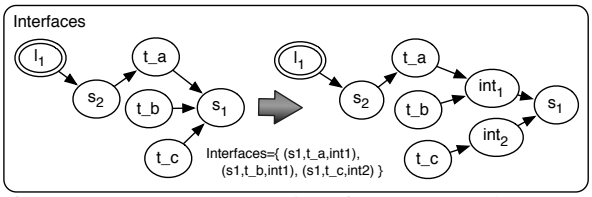

Figure 7: Representation of interfaces. Program s_1 has two interfaces, int_1 and int_2, to access objects of labels t_a, t_b, and t_c. Level l_1 propagates to s_1 via its interface int_1 only.

the levels in the lattice to create its topological sort LS. Line 4 determines the set of labels that conform the cut problem, (i.e., the labels that cannot flow to the sink that is being analyzed), line 5 creates a copy of the graph for the current cut-problem, and line 6 creates a SuperSource connected to the nodes mapped to levels identified in line 4. Line 7 adds a new node to represent the sinklabel l and adds an edge (n, l) for every node n that is mapped to label l. Line 8 creates a slice of the graph with only the nodes that can reach the sink. Line 9 adjusts weights to account for the fact that not all nodes can be mediators for a sink. Finally, line 10 and 11 compute the minimum cut for that particular cut problem and adds the new set of mediators to the global set of mediators.

The running time of the algorithm is dominated by the time of computing the MinimumCut for each sink, (for every label in the lattice $\mathcal{L} = \{L, \preceq\}$). The running time of the MinimumCut algorithm is $O(n^3)$, thus, the running time of the MEDIATIONRESOLUTION algorithm is $O(|L|.n^3)$. The algorithm and a detailed explanation are provided in a tech report [5].

3.4.3 Mediation Costs

OS MAC policies are defined in terms of programs, but practical integrity requires defenses be placed at particular program interfaces (i.e., system calls) where mediation is necessary. For example, the Flume system permits the program to choose which system calls may receive data of lower integrity than the process by using its capabilities. Thus, in order to evaluate the cost of mediation, it is necessary to determine the cost of defending program interfaces that may receive untrusted inputs.

There are two options for representing the cost of mediation for program interfaces. First, each node may be associated with a mediation cost, indicating the number of interfaces to mediate for that node[7]. This approach assumes that all low integrity inputs will always reach all the program interfaces, regardless of which MAC labels are accessed by which interfaces.

Second, a node can be created for each interface in the program, and an edge is added from a MAC label node to an interface node only when it is known that subjects or objects with that MAC label are read by that interface. Determining which interfaces receive which data is not specified at present, but can be estimated through runtime analysis [4]. We explore how this approach may be leveraged in the evaluation, see Section 4. Figure 7 illustrates how the data flow graph would be modified.

4 Evaluation

In this section, we present the results of running a prototype implementation of the Proactive Integrity Methodology on the web application system presented in Section 2.1. To demonstrate the tool, we examine two questions (see Section 4.2): (1) how can OS distributors use the tool to find the mediation required for their systems and (2) how can system administrators use the tool to ensure that all necessary mediation is required and to attribute integrity enforcement requirements among components.

[7]Since we are compute edge cuts, this is performed by creating a pseudonode for each node with and edge between them of the appropriate weight.

System	Problem Size							Solution	
	Rules	Subjs	Objs	Intfs	Ports	Nodes	Edges	Cut Subjs	Cut Intfs
Run One	1,248,590	325	1,775	7,093	37	9,230	77,091	50	2048
Run Two	1,248,590	325	1,771	6,802	37	8,935	35,105	35	498
Run Three	1,541,993	394	2,186	8,322	42	10,944	44,638	43	511

Table 2: Analysis problems and results for: (1) the web application with all authorized operations (Run One); (2) the web application with operations logged based on runtime analysis (Run Two); and (3) the web application plus a DNS server system with operations logged based on runtime analysis (Run Three).

4.1 Prototype System and Experiment

The prototype has an Eclipse front-end that provides access to: (1) parsers to build system-wide HSM data flow graphs for XSM/Flask [13], SELinux [42], Labeled IPsec [22], and iptables with the Secmark extension [34]; (2) mapping rules to infer integrity level mappings and integrity lattices for constructing information flow integrity models; (3) the Lemon graph library [43] to compute graph cuts for our information flow model; and (4) a module for generating DIFC-Flume policies from graph cuts. The code breakdown is as follows: (1) 4303 source lines of code (SLOC) for the Bison parsers; (2) 405 SLOC in Prolog for mapping/dependence rules; (3) 1189 SLOC for using the Lemon graph library to solve multiple cut problems; and (4) 1654 SLOC in C for generating DIFC-Flume policies.

We evaluate a version of the example system described in Section 2.1. The focus is on the web server host, which runs Xen VM system 4.1, consisting of Dom0 privileged VM, web server VM including an Apache web server and web application, and MySQL database VM. Also, there is a separate user VM that is not part of the web application (i.e., they are mutually distrustful). Each VM runs the Ubuntu 2.6.31-23-generic kernel. The Xen hypervisor enforces XSM/Flask policies [13], and each of the VMs enforce SELinux policies [42]. While all of the OS policies are SELinux, they are independent in the sense that each policy supports a distinct set of applications and these policies do not refer to the interactions among VMs. Also, each VM runs an iptables firewall with the Secmark extension [34] to govern network communications. We assume that secure communication (e.g., IPsec or SSL) is used to protect any channel that carries data between hosts on a labeled channel. Unlabeled channels are given the "External" level (i.e., system-low). Finally, the other hosts are clients of the web application and an external network server for processing DNS requests. All clients are assumed to be untrusted.

4.2 Experimental Results

Generating Mediations. In the first experiment, we compute the mediation required in the web application system. For this experiment, we consider two cases: (1) Run One, where we assume that all authorized operations in the MAC policies will be performed by the web application system, and (2) Run Two, where only operations collected from a runtime analysis of the system are used in the analysis. Run One provides an upper bound for the amount of mediation required in a distributed system since it includes all possible operations at all interfaces. Run Two provides an estimate for the lower bound for the amount of mediation required since only the operations logged in a typical deployment are included. We use the Linux test suites supplied with Apache web servers and MySQL databases to generate the runtime data. We logged the MAC policy labels used at each program interface to determine the permissions used at runtime [8].

Table 2 shows the analysis results. First, the number of subjects and objects in the two versions of the system are nearly identical (i.e., some object labels were not used in the runtime case), but the number of objects accessed by individual subjects (i.e., graph edges) is much smaller for Run Two. As a result, the required cut necessary to mediate Run Two is also smaller. In Run One, the cut shows that mediation is required in 50 unique subjects[9] spanning 2,018 interfaces, clearly a daunting task. In Run Two, mediation is

[8] Specifically, a program interface is the instruction in the program executable file that invokes a libc function that corresponds to one or more system calls (e.g., `open` and `fopen`).

[9] The tool uses unique subjects, as we assume the mediation requirements of the programs in different deployments are sufficiently similar to reuse the effort.

	Number of Dual Capabilities										Total
	2	3	4	5	6	7	8	9	10	11	
Run Two	2	4	0	5	2	1	0	0	20	0	262
Run Three	10	2	1	0	8	1	0	0	14	7	302

(a)

System	DFG	IFM	Cuts	DIFC
Run One	18.7	1.0	32.6	8.2
Run Two	-	0.8	13.7	4.9
Run Three	4.0	0.9	24.6	8.7

(b)

Table 3: (a) DIFC-Flume Dual Capabilities for Runs Two and Three. Subjects are grouped per number of capabilities. (b) Execution Times, all in seconds. DFG: Construction of the hierarchical data flow graph, IFM: Constructions of the information flow integrity model, Cuts: running the MEDIATIONRESOLUTION program, DIFC: Computing DIFC-Flume labels and capabilities.

required for 35 unique subjects and 488 interfaces. We found that the major cause of the difference between the amount of mediation required in the two runs is due to the heavy use of attributed-based permissions in SELinux. Attributes are an easy way to express aggregate access by associating a permission to a set of labels, but few of the aggregate permissions appear to be used by the test cases. Based on this experiment, we envision that measures need to be explored to refine the use of attributes for deployments.

In the second experiment, we show that the tool computes the impact of assigning integrity dependence. In the first experiment above, we did not include the DNS server in the web application system, so it was automatically assumed to provide untrusted (External) input. However, distributed systems depend on DNS resolutions, so we add the DNS server to the distributed system[10] in Run Three of Table 2. The mediation requirements are similar because the mediation is moved from the guest VM to the DNS server in some cases. This cost could be amortized across many distributed systems, however.

While adding the DNS server had little impact on the mediation cost, it reduced the number of DIFC capabilities for the web server and database subjects as shown in Table 3a. As seen, several subjects with 3-5 capabilities in Run Two have been reduced to 2-4 capabilities, eliminating the need to protect themselves from DNS resolutions. The subjects with 10 and 11 capabilities are in Dom0 and the DNS server now, so they are entrusted with a variety of security decisions. Thus, while it is still necessary to provide mediation for the same number of interfaces, the security decisions that must be made in the guest VMs are reduced, focusing the defensive effort in the privileged VM and services.

System Administrator Use. Next, we describe how system administrators would use the results generated by OS distributors. OS distributors, in addition to giving us the policy, would also specify nodes in the data flow graph representing program interfaces that have mediation capability. As an experiment, we provide a white-list of interface nodes and the integrity level to which they can raise inputs. The tool can then choose among the white-listed mediation interface nodes for the deployment automatically by performing the cut-algorithm, determining if a complete cut is possible with these nodes only. The choices of which interfaces provide which mediation is outside the scope of this work.

Performance. Our experiments were run on a machine with a 2.3 GHz Intel Xeon with 8GB of RAM. Table 3b shows the execution time for the experiments by the cost of computing methodology tasks. The times reflect an average over 10 runs. Note that the hierarchical data flow graph only needs to be built once per analysis session, even if different runtime profiles for system are used. As a result, there is no additional time required data flow graph processing for Run Two (since it is the same) and only the DNS server is added in Run Three. The time of computing the cuts is dominated by the number of levels in the lattice (each level represents a different problem) and the computation of the minimum cut for each problem. The DIFC-Flume label computation is dominated by the transitive closure, described in Section 3.4.1.

We note that optimizations are possible that partition the system into pieces that may be processed independently, analogously to summary functions [55]. For example, since the user VM is supposed to isolated from the rest of the system by the privileged VM, the integrity levels of its inputs are independent of the rest of the system. Thus, the user VM mediation can be computed first, then applied to the rest of the

[10] An alternative would be to simply label the data received from the DNS server using labeled networking, but that would ignore whether the DNS server actually protects its own integrity. Once this is established, then the results can be reused, analogous to a summary function.

system. The insight is that the use of integrity dependence to protect another component is useful to create such partitions. We have not implemented these improvements yet, so they are part of our future work.

5 Related Work

Researchers have long known that end-to-end integrity protection can be modeled as an information flow problem. However, classical integrity models [9, 12] rely on formal assurance for all high integrity processes that receive untrusted inputs, but formal assurance methods have not been developed that scale to the size of modern programs. As a result, least privilege [51] has been adopted as the security goal for integrity protection in commodity systems [42, 40, 35, 63, 64]. However, different deployments imply different privileges, so commodity systems try to accommodate this through mechanisms to compute permissions requested by processes [47, 40, 6] and by providing runtime configuration options to specify permissions for options. Nonetheless, it is important that default configurations work in most cases, so researchers have found that the default MAC policies still permit several operations that would violate classical integrity [11]. Researchers have recently proposed practical approaches to integrity that approximate classical integrity models [54, 28, 57, 27, 37]. These practical integrity models are all strictly weaker than classical models, in that they do not require formal assurance for code with the authority to protect integrity while receiving untrusted inputs. However, they express restrictions on how such authority may be used by high-integrity programs. For example, the Decentralized Information Flow Control [27, 67] (DIFC) model provides processes with capabilities to make decisions about handling untrusted inputs.

With the emergence of MAC enforcement in commodity operating systems, researchers proposed various policy design tools to help system administrators and OS distributors configure policies [23, 60, 19, 53, 66, 11]. Broadly speaking, these tools enable a policy designer to evaluate compliance using *reachability* to identify whether an adversary can perform an unauthorized operation, even indirectly. Reachability analyses have also been performed for network policies in the form of *attack graphs* [44, 56, 39], but these represent attacker behavior rather than using the system policies. However, defining which operations are unauthorized and resolving any problems are manual tasks. SELinux policy generation from such resolutions is possible, but the reasons for the resolutions and the resulting mediation are lost [60]. Pike [45] and King *et al.* [25, 26] independently found that resolutions could be estimated as graph cuts automatically. However, these approaches approximate minimal solutions for two-level lattices only and assume that all locations are capable of any mediation, so they do account for the practical problems in distributed systems. Another key question is what security property should be optimized by the choice of such cuts. Researchers have recently focused on defending an *attack surface* [21, 31], which is the set of program interfaces accessible to adversaries. The idea would be that if we can minimize the number of interfaces accessible to attackers, we could minimize our defensive effort.

The problem of verifying that a MAC policy satisfies a set of security requirements is a policy compliance problem. In a *policy compliance problem*, a policy is said to *comply* with a goal if all the operations authorized by the policy satisfy the constraints of the goal [33, 17, 32, 20, 49]. The problem is that MAC policies often fail to comply with integrity requirements, as discussed above, so we must repair non-compliant cases. However, any MAC policy changes must also preserve necessary function, and balancing functional and security requirements is computationally complex in general.

6 Conclusion

We proposed the Proactive Integrity methodology to help OS Distributors and systems administrators configure systems according to integrity requirements in their target deployments. We do so by designing a model that composes multiple independently developed MAC policies into a concise hierarchical representation, and use that model to evaluate the safety of the system and automatically compute an near-optimal

cost solution that minimizes the effort necessary to have a safe system. We constructed a tool that implements this methodology to generate system-wide, Decentralized Information Flow Control (DIFC) MAC policies for Linux web application deployments in tens of seconds.

References

[1] ALUR, R. Formal analysis of hierarchical state machines. In *Verification: Theory and Practice*, vol. 2772 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 434–435.

[2] ALUR, R., AND YANNAKAKIS, M. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst. 23*, 3 (2001), 273–303.

[3] ANDERSON, J. P. Computer Security Technology Planning Study, Volume II. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), October 1972.

[4] ANONYMOUS. Integrity walls: Finding attack surfaces from mandatory access control policies. `https://sites.google.com/site/anontechrep/anontechreports/wall.pdf`.

[5] ANONYMOUS. Mininum cut algorithm with integrity constraints. `https://sites.google.com/site/anontechrep/anontechreports/policy_cut.pdf`.

[6] `http://fedoraproject.org/wiki/SELinux/audit2allow`, 1996.

[7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (Oct. 2003).

[8] BELL, D. E., AND LAPADULA, L. J. Secure Computer System: Unified Exposition and Multics Interpretation. Tech. Rep. ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), March 1976.

[9] BIBA, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[10] CARTER, J. Using GConf as an Example of How to Create a Userspace Object Manager.

[11] CHEN, H., LI, N., AND MAO, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS* (2009).

[12] CLARK, D. D., AND WILSON, D. A Comparison of Military and Commercial Security Policies. In *IEEE Symposium on Security and Privacy* (1987).

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

[14] DAHLHAUS, E., JOHNSON, D. S., PAPADIMITRIOU, C. H., SEYMOUR, P. D., AND YANNAKAKIS, M. The complexity of multiterminal cuts. *SIAM J. Comput. 23* (August 1994), 864–894.

[15] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient smt solver. In *TACAS* (2008), pp. 337–340.

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

[17] DRAGONI, N., MASSACCI, F., NALIUKA, K., AND SIAHAAN, I. Security-by-contract: Towards a semantics for digital signatures on mobile code. In *EuroPKI* (2007).

[18] GASSER, M. *Building a secure computer system*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[19] GUTTMAN, J. D., HERZOG, A. L., RAMSDELL, J. D., AND SKORUPKA, C. W. Verifying Information Flow Goals in Security-Enhanced Linux. *Journal of Computer Security 13*, 1 (2005).

[20] HICKS, B., RUEDA, S., ST. CLAIR, L., JAEGER, T., AND MCDANIEL, P. A logical specification and analysis for SELinux MLS policy. *ACM Transaction on Information and System Security 13*, 3 (2010).

[21] HOWARD, M., PINCUS, J., AND WING, J. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security* (2003).

[22] JAEGER, T., BUTLER, K., KING, D. H., HALLYN, S., LATTEN, J., AND ZHANG, X. Leveraging IPsec for Mandatory Access Control Across Systems. In *Proc. 2nd Intl. Conf. on Security and Privacy in Communication Networks* (Aug. 2006).

[23] JAEGER, T., SAILER, R., AND ZHANG, X. Analyzing integrity protection in the SELinux example policy. In *USENIX Security Symposium* (Aug. 2003).

[24] JHA, S. K., LIMAYE, R. S., AND SESHIA, S. A. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. Tech. Rep. UCB/EECS-2009-95, EECS Department, University of California, Berkeley, Jun 2009.

[25] KING, D., JHA, S., JAEGER, T., JHA, S., AND SESHIA, S. A. Towards Automated Security Mediation Placement. Tech. Rep. NAS-TR-0100-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, November 2008.

[26] KING, D., JHA, S., MUTHUKUMARAN, D., JAEGER, T., JHA, S., AND SESHIA, S. A. Automating security mediation placement. In *ESOP* (2010).

[27] KROHN, M. N., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Pr inciples* (Oct. 2007), pp. 321–334.

[28] LI, N., MAO, Z., AND CHEN, H. Usable Mandatory Integrity Protection For Operating Systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007).

[29] LINUX KVM. Kernel based virtual machine. `http://www.linux-kvm.org`.

[30] LOSCOCCO, P. A., SMALLEY, S. D., MUCKELBAUER, P. A., TAYLOR, R. C., TURNER, S. J., AND FARRELL, J. F. The inevitability of failure: The flawed assumption of security in modern computing environments. In *In Proceedings of the 21st National Information Systems Security Conference* (1998), pp. 303–314.

[31] MANADHATA, P., TAN, K., MAXION, R., AND WING, J. M. An Approach to Measuring A System's Attack Surface. Tech. Rep. CMU-CS-07-146, School of Computer Science, Carnegie Mellon University, 2007.

[32] MASSACCI, F., AND SIAHAAN, I. Matching Midlet's security claims with a platform security policy using automata modulo theory. In *In proceedings of NordSec* (2007).

[33] MCDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. *ACM Trans. Inf. Syst. Secur.* (2006).

[34] MORRIS, J. New secmark-based network controls for selinux. `http://james-morris.livejournal.com/11010.html`.

[35] MSDN. Mandatory Integrity Control (Windows). `http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx`.

[36] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *POPL '99* (1999), pp. 228–241.

[37] MYERS, A. C., AND LISKOV, B. A decentralized model for information flow control. *ACM Operating Systems Review 31*, 5 (Oct. 1997), 129–142.

[38] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java information flow. `http://www.cs.cornell.edu/jif`, `July2001-2003`.

[39] NOEL, S., JAJODIA, S., O'BERRY, B., AND JACOBS, M. Efficient minimum-cost network hardening via exploit dependency graphs. In *ACSAC* (2003).

[40] NOVELL. AppArmor Linux Application Security. `http://www.novell.com/linux/security/apparmor/`.

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

[42] Security-enhanced linux. `http://www.nsa.gov/selinux`.

[43] ON COMBINATORIAL OPTIMIZATION, E. R. G. Lemon Graph Library. `http://lemon.cs.elte.hu/trac/lemon`.

[44] OU, X., BOYER, W. F., AND MCQUEEN, M. A. A scalable approach to attack graph generation. In *CCS* (2006).

[45] PIKE, L. Post-hoc separation policy analysis with graph algorithms. In *Workshop on Foundations of Computer Security (FCS'09). Affiliated with Logic in Computer Science (LICS)* (August 2009).

[46] Sepostgresql introduction. `http://wiki.postgresql.org/wiki/SEPostgreSQL`, August 2009.

[47] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium* (August 2003).

[48] REDHAT. Chapter 17. sVirt. `http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization/chap-sVirt.html`.

[49] RUEDA, S., KING, D., AND JAEGER, T. Verifying Compliance of Trusted Programs. In *Proc. 17th USENIX Security Symp.* (2008).

[50] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., PEREZ, R., BERGER, S., GRIFFIN, J. L., AND VAN DOORN, L. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *ACSAC* (Washington, DC, USA, 2005).

[51] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sep. 1975).

[52] SANS. http://www.sans.org/top-cyber-security-risks/tutorial.php.

[53] SARNA-STAROSTA, B., AND STOLLER, S. D. Policy analysis for security-enhanced linux. In *WITS* (April 2004).

[54] SHANKAR, U., JAEGER, T., AND SAILER, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium* (February 2006).

[55] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications* (1981), S. S. Muchnick and N. D. Jones, Eds., Prentice Hall, pp. 189–233.

[56] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2002), IEEE Computer Society, p. 273.

[57] SUN, W., SEKAR, R., POOTHIA, G., AND KARANDIKAR, T. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *Proceedings of the IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 248–262.

[58] SUN MICROSYSTEMS. Trusted solaris operating environment - a technical overview. http://www.sun.com.

[59] TRESYS. Policy management server. http://www.tresys.com/selinux/selinux_policy_server.html.

[60] TRESYS. Policy tools for SELinux policy management. http://oss.tresys.com/projects/.

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

[62] WALSH, E. Application of the Flask architecture to the X window system server. In *2007 SELinux Symposium* (2007).

[63] WATSON, R. N. M. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), pp. 15–28.

[64] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), pp. 17–31.

[65] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *SOSP* (2009).

[66] ZANIN, G., AND MANCINI, L. V. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT* (2004).

[67] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *OSDI* (2006).

A IFM and DIFC-Flume Equivalence

In this document we show show the equivalence of the Information Flow Model(IFM) we propose and the DIFC-Flume model.

Based on the translation above, we can prove that the notions of safety in both models are equivalent.

Definition 1. *[IFM safe]. A system S is safe under the IFM model if and only if for any flow $u \rightarrow v$, $OL_{ifm}(u) \preceq OL_{ifm}(v)$ or v is a mediator.*

Definition 2. *[Flume secure]. A system S is secure under the Flume model (Flume secure) if and only if: (1) any label change for a process p from L to L' adheres to the rule that $\{L' - L\}^+ \cup \{L - L'\}^- \subseteq O_p$, where O_p is the set of positive and negative capabilities assigned to p and (2) a message can be sent from p to q only if $I_q \subseteq I_p$ or $I_q - D_q \subseteq I_p \cup D_p$ where D_q and D_p are the dual capabilities of q and p respectively.*

Theorem 1. *[Flume-IFM equivalence]. A system S is safe (for integrity) under the IFM model (IFM safe), if and only if, S is secure (for integrity) under the Flume model (Flume secure).*

Proof. We start by defining mapping functions from the IFM lattice to DIFC-Flume lattice and vice versa.

Mapping Functions: We first define mapping functions, I2F and F2I, to map IFM levels into DIFC-Flume labels and vice versa. Note that we use the term IFM level, as the term IFM label has a different meaning in our model (a label is a set of levels). Figure 8 illustrates the mapping functions. Given an IFM lattice $\mathcal{L}_{ifm} = (L_{ifm}, \preceq_{ifm})$ let I2F : $L_{ifm} \rightarrow L_{flume}$ be I2F$(l) = \{x \in L_{ifm} |\ l \preceq_{ifm} x\}$. We build the corresponding DIFC-Flume lattice $\mathcal{L}_{flume}$=$(L_{flume}, \preceq_{flume})$ as follows:

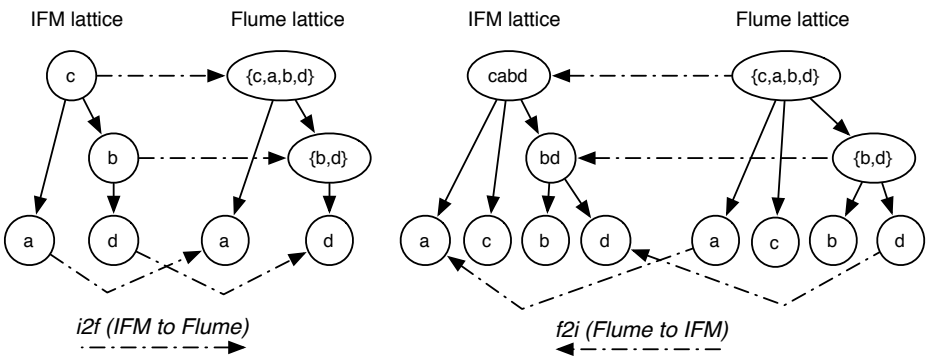

Figure 8: IFM to DIFC-Flume and DIFC-Flume to IFM Mapping Functions.

- for every $l \in L_{ifm}$, we add $\text{I2F}(l)$ to L_{flume}
- for every $(l_1, l_2).\ l_1, l_2 \in L_{ifm} \wedge l_1 \preceq_{ifm} l_2$, we add $(\text{I2F}(l_1), \text{I2F}(l_2))$ to $\preceq_{flume}$

Note that the relation *can flow to* $\preceq$ corresponds to the relation superset $\supseteq$ on sets as defined by the DIFC-Flume model.

Given a Flume lattice $\mathcal{L}_{flume} = (L_{flume}, \preceq_{flume})$
let $\text{F2I} : L_{flume} \rightarrow L_{ifm}$ be $\text{F2I}(\{l_1, ..., l_n\}) = concat(l_1, ..., l_n)$.
We build the correspondent IFM lattice $\mathcal{L}_{ifm} = (L_{ifm}, \preceq_{ifm})$ as follows:

- for every $l \in L_{flume}$, we add $\text{F2I}(l)$ to L_{ifm}
- for every $(l_1, l_2).\ l_1, l_2 \in L_{flume} \wedge\ l_1 \preceq_{flume} l_2$ (or equivalently $l_1 \supseteq l_2$), we add $(\text{F2I}(l_1), \text{F2I}(l_2))$ to $\preceq_{ifm}$

We have defined the translation from IFM nodes to DIFC-Flume subjects and objects. The reverse translation from DIFC-Flume subjects and objects to IFM nodes works the same way. Definitions 1 and 2 specify safety requirements for each model.

We can now define the *safety-equivalence* property between the IFM lattice and DIFC-Flume.
Theorem 1 *A system S is safe (for integrity) under the IFM model (IFM safe), if and only if, S is secure (for integrity) under the DIFC-Flume model (Flume secure).*

Proof of Theorem 1: We first prove: (A) if a system is IFM safe then it is Flume secure, and then (B) if a systems is Flume secure then it is IFM safe.
A. IFM safe to Flume secure. Let flow $p \rightarrow q$ be IFM safe. This gives us two cases.
[Case 1]: q is a non-mediator node. Therefore, $OL_{ifm}(p) \preceq OL_{ifm}(q)$. From the translation rules to DIFC-Flume this would imply that $\text{I2F}(OL_{ifm}(p)) \supseteq \text{I2F}(OL_{ifm}(q))$ This would be a safe flow in DIFC-Flume as well.
[Case 2]: Node q is an mediator node and $OL_{ifm}(q) \preceq OL_{ifm}(p)$. When we translate this to DIFC-Flume, we ensure that the dual capabilities of q, D_q, contains tags for tags $\text{I2F}(OL_{ifm}(q))$ and for integrity levels dominated by $\text{I2F}(OL_{ifm}(q))$ (this includes tags for $\text{I2F}(OL_{ifm}(p))$, since $OL_{ifm}(q) \preceq OL_{ifm}(p) \in L_{ifm}$) in the DIFC-Flume lattice. This ensures that this flow is secure under DIFC-Flume.
B. Flume secure to IFM safe. A system S is Flume secure if and only if all allowed process label changes are safe and all allowed messages are safe.

[Case 1]: All allowed process label changes are safe. This means that any process q can change its label only within the range of its capabilities D_q. We map processes with capabilities in Flume to mediators in the IFM model with raiselevel (RL_ifm) equal to the union of all tags in the dual capability set D_q. Mediators are safe nodes in the IFM model as they automatically downgrade higher integrity levels to its raiselevel and endorse lower integrity levels to its raiselevel.

[Case 2]: All allowed messages are safe. This means that for all messages from p to q: $I_q - D_q \subseteq Ip \cup D_p$.
[2.1]: Assume $D_p = D_q = \{\}$ thus $I_q \subseteq I_p$. That means $\text{F2I}(I_q) \succeq \text{F2I}(I_p)$. This is a safe information flow in IFM.
[2.2]: Assume $D_p \neq \{\} \wedge D_q = \{\}$ thus p will always run with integrity I_p or higher. Since $D_q = \{\}$, then $I_q \subseteq I_p$ in the extreme case. Therefore, $\text{F2I}(I_q) \succeq \text{F2I}(I_p)$. This is a safe information flow in IFM.

[2.3 and 2.4]: Assume ($D_p = \{\} \wedge D_q \neq \{\}$), or ($D_p \neq \{\} \wedge D_q \neq \{\}$). In both cases we map q to a mediator in the IFM model. Mediators are safe nodes in the IFM model as they automatically downgrade higher integrity levels to its raiselevel and endorse lower integrity levels to its raiselevel.