

Application Security Support in the Operating System Kernel

Manigandan Radhakrishnan
University of Illinois at Chicago
851 S. Morgan Street, M/C 152
Room 1120 SEO
Chicago IL 60607-7053
mradhokr@cs.uic.edu

Jon A. Solworth
University of Illinois at Chicago
851 S. Morgan Street, M/C 152
Room 1120 SEO
Chicago IL 60607-7053
solworth@cs.uic.edu

ABSTRACT

Application security is typically coded in the application. In *kernelSec*, we are investigating mechanisms to implement application security in an operating system kernel. The mechanisms are oriented towards providing authorization properties, and this goal drives the design of permissions and protection mechanisms.

The resulting system is dynamic, allowing the set of permissions for a program to evolve during program execution. This reduces the need for users and applications to be aware of protection mechanism, since the protection mechanism provides the user with more freedom in how they do things. We explore these properties through a number of examples.

KernelSec also supports a group (role) mechanism which can define constrained groups enabling groups which only grow, only shrink, are constant, are mutually exclusive with other groups, and which allow inheritance. Moreover groups are used to regulate group membership and allow group administration by non-privileged users.

Categories and Subject Descriptors

D.4.6 [Operating System]: Security and Protection—*Access Controls, Information Flow Controls*

General Terms

Security, Operating Systems

Keywords

Authorization, Authorization Properties, Access Controls, Information Flow, Separation of Duty

1. INTRODUCTION

Operating Systems (OSes) play a crucial role in providing system security since they are, by necessity, part of the

Trusted Computing Base (TCB) [21]. The OS controls a process's interaction with the outside world; in order for a process to affect anything outside its private address space it must request the operating system to perform an operation on its behalf. Thus, the OS's *authorization system* allows or denies each operation requested by a process, thereby mediating the interaction. In general, authorization decisions are based upon the user who invoked the process (and on whose behalf it is assumed to be running), the program that it is executing, and the history of the process. Operating systems provide "least privilege" [22] via their authorization system, so that the processes have sufficient permissions to perform only *needed* functions.

In spite of these advantages, much of authorization today is provided in the applications. Many applications contain significant amount of code and documentation and require configuration to specify what the application is allowed to do; further complexity arises due to inconsistent specifications across applications. Many applications provide no security code and yet may pose dangers both to the user that executes them and to the system on which they run. In any event, the protections in the application can be bypassed by buffer overflow attacks, rendered ineffective by incorrect implementation, or simply misunderstood. Because the application base is extremely large and rapidly changing, spreading authorization across the application base is antithetical to shrinking and validating the TCB.

But even if it were possible to overcome the above difficulties, the composition of two secure applications is not necessarily secure, and hence application-by-application configuration is insufficient for systems with multiple applications.

By providing effective authorization at the operating system level, system-wide specification is achieved; thus compositional issues do not come into play. In addition, analytical tools can be used to determine what the security configuration does. Analytical tools are particularly important for security issues as a system may appear to be correct under normal use while its flaws become apparent only under attack. In authorization they are paramount, since each organization determines—and therefore needs to verify—its own authorization policy. (Even the smallest organization must tradeoff functionality vs. protection when deciding its system configuration).

Of course, application correctness—that the program does the right thing in the absence of security concerns—remains a crucial security concern. However, effective authorization limits what each application can do, rendering it less dangerous if successfully attacked (eg. via buffer overflow), preventing certain classes of attack, and enabling the identification of those applications whose correctness has the greatest security impact and thus requires the greatest scrutiny.

Structurally the authorization system is implemented at the topmost layer of the operating system. Hence, it is in principle possible to change an authorization system without changing other interfaces to the operating system. For example, CMW [3], SELinux [27], TrustedBSD [35] have been implemented on top of existing OSes.

Despite the substantial and inherent advantages of OS-based authorization, application-based authorization proliferate. OS-based authorization systems either lack the expressiveness to meet the varied authorization requirements of applications or are too complex.

The expressiveness can be described by the type of authorization properties that can be enforced by the authorization system. Traditionally, OS's provide Discretionary Access Controls (DACs) in which object-based protections can be specified by the “owner” of the object. Mandatory Access Controls (MACs)—which is used here to mean the access controls imposed by the organization (rather than the more narrowly defined lattice based access controls)—can provide system-wide protection, for example information flow, separation-of-duty, and group (or role) management. To supplant application-based authorization, the OS authorization system must be sufficiently expressive to provide the needed MAC protections.

The complexity of an authorization system can be examined based on the impact it has on three different classes of individuals. They are the system administrators (who specify the policy of a system), the users, and the applications (or their developers).

System administrators configure the system security, which necessitates that they understand the requirements of the applications—in general, each application will need different privileges—as well as specify and understand the authorization configuration. Of course, in general, the more expressive the system the more difficult it is to configure, but it is desirable to reduce the complexity for a given level of expressiveness.

Authorization systems restrict not only who can do a certain task but also how they must be done. Users must understand what is required of them (w.r.t the authorization system) to perform desired tasks. For some types of security policies like Multi-Level Security (MLS), it is both necessary and appropriate to indoctrinate users in the security policy. To reduce complexity, it is desirable to minimize the awareness of the authorization policies expected of the user. In *kernelSec*, this security awareness is reduced by having the permissions associated with a process change based on application actions. This ensures least privilege by minimizing the permissions associated with a process at any time, while

maintaining flexibility as application actions can implicitly cause permissions to be acquired (and released).

Applications (and hence their developers) interact with the authorization system in two ways (1) when they request any operation and (2) when they try to change the authorization state. The impact the authorization system has on (1) is minimal because requests by the application may be denied for a variety of reasons (including lack of authorization) and need to be resilient against them. In (2), the applications (and the developer) need to be aware of the authorization system, and sometimes the exact authorization configuration, to make these changes. In *kernelSec*, the application's explicit interactions with the authorization system is reduced by incorporating mechanisms that change the permanent authorization state into the authorization configuration. Thus the authorization state changes can be triggered implicitly by application operations, eliminating the need for the developer to explicitly code them in the program.

In this paper, we describe how *kernelSec* supports application security, provides support for high level authorization properties such as dynamic separation of duty, dynamic information flow, and group management. In particular, the *kernelSec*'s group mechanism provides some administrative controls at the operating system level. This is the first OS implementation that we are aware of which provides this combination of general purpose and dynamic facilities.

The remainder of this paper is organized as follows. Section 2 highlights the key features in *kernelSec* while Section 3 describes the design in detail. Section 4 describes the expressiveness of the protections that can be constructed in *kernelSec*, using a series of examples. Section 5 describes the current state of the *kernelSec* implementation, including some performance numbers. Section 6 describes the overview of how *kernelSec* addresses the expressiveness vs. complexity tradeoffs discussed above. In Section 7 we describe related work and finally in Section 8 we conclude.

2. KERNELSEC: OVERVIEW

KernelSec can be viewed as a generalization of Type Enforcement (TE) [5, 20, 1, 27, 35] which in turn is based on the access matrix [17]. In TE, each object has a single label which partitions the objects by protection *type*. Each process executes in a *domain*; the domain defines for each label the permissions which a process executing in that domain has on objects of that type. Transitions between domains occur when changing the executable of a process (via the `exec` operation) which has a similar purpose to `setuid-bit` in Unix [8].

TE is an attractive base, since it can control information flow, implement static separation of duty, and associates permissions with an executable (domain) as well as user. But domains are entered only when `exec`'ing a new program. Hence, the *user* must be aware of which program (domain) will have suitable permissions for performing the task. For some applications, where the program is tightly tied to the data (as in Clark-Wilson [9]) this is not a problem. But many applications use general purpose programs, such as an editor, and hence a user may be working in a domain and be unable to complete her work, since a domain transition

(requiring an `exec`) would replace the entire user state of the process. At the least, this would require the application to save the state in the filesystem, the user to exit the current application and start up one with sufficient privileges, and for that application to load its state from the saved file. In addition, this requires both user knowledge as well as application programming to save and restore state.

In contrast, *kernelSec* is much more dynamic as it allows a process’s permissions to change as the result of accesses it makes. We next highlight the key attributes of *kernelSec* relative to TE:

2.1 Dynamic (run-time) domain transitions

Dynamic domain transitions allow the permissions associated with the process to change based on the operations performed by the process. For example, after reading a highly sensitive file, the process may not be able to write a publicly readable file.

In systems which do not have dynamic transitions, either (1) the needed process permissions must be chosen in advance of the accesses performed (eg. between one which can read a highly sensitive file and one which can not) or (2) least privilege must be violated and the application itself must enforce authorization. (1) requires the users to be cognizant of the authorization system while (2) results in the OS being too weak to enforce the necessary application authorization.

2.2 Active transitions

Active transitions couple changes in a process’s permissions with changes to the permanent authorization state. For example, in Dynamic Separation of Duty (DSoD) when a user performs a step, completion of the step and the fact that she performed it needs to be recorded. In *kernelSec*, the active transitions allow the authorization system to implicitly track which user performed the operation. Of course, the authorization configuration must specify when this tracking is to be done.

Since active transitions are part of the authorization system, the application need not be aware of the exact constraints of the authorization configuration (and application code is not needed to make these changes). This enables state invariants to be verified independent of the application code. As far as we know, ours is the first system that implements active transitions.

2.3 Group management

Group management uses the protection system itself to determine how group membership changes over time. The advantages of our group mechanism are that any user can create a new group; any group of users can be defined to regulate the membership of the group; and the permissions can be defined to restrict future group membership.

2.4 Reducing complexity

Least privilege combined with dynamic and active transitions can result in the need to produce a large number of kernel-level entities. Manually creating a new authorization configuration or modifying an existing authorization configuration could be tedious. By generating these entities

from a higher level specification, much of the tedium can be avoided; moreover, the higher level can be analyzed for the protections provided, rather than at the kernel level. We shall return to this subject in Section 6.

3. KERNELSEC COMPONENTS

The major components of the *KernelSec* system are: (1) the *users*, which represent humans; (2) the *labels*, which are used to define an object’s permissions; (3) the *privileges*, which are the permissions that can be used to determine accesses to objects; (4) the *groups* which are the basis of organizing users and assigning permissions, and are organized into “group sets”; and (5) the *Security Cards* which bind the privileges to a process.

3.1 Labels

Each object in *kernelSec* has a single label which (in conjunction with *Security Cards*) completely determines its protection class—that is the set of operations that can be performed on it and the group of users authorized to perform each of them. Each *kernelSec* label is a pair $\langle p, t \rangle$ where p is called the parameter, and t the tag. The tag determines the label type. The parameter varies based on label type and is used to encode additional information about the label (as shown below). *KernelSec* has group labels, DAC labels, and MAC labels as follows:

label type	group	MAC	DAC
label form	$\langle U, T_g \rangle$	$\langle G, T_n \rangle$	$\langle U, T_d \rangle$
parameter type	user ID	group set ID	user ID

The *ordinary objects*, which are the traditional objects of the operating system, are the DAC and MAC objects. Due to space limitations, we will not discuss DAC objects further. An object with a group label, can only be created during group creation or when adding a new user (defined in 3.2)—its sole purpose is in determining group membership.

An arbitrary number of labels can be minted in *kernelSec* and permissions are specified on a per label basis.

3.1.1 Notation

The following notation is used: ‘*’ matches any value; ‘**u*’ refers to the user on whose behalf the process is executing; and ‘**g*’ refers to the group set ID of the current MAC label (if any). Moreover, if a rule specifies more than one label, then a ‘.’ in a component of the second label means that the component must have the same value as the corresponding component of the first label. For example, a user map rule $\langle *u, G \rangle \rightarrow \langle ., G' \rangle$ means that group object for any user U in the base group with tag G gets mapped to a group object for U with tag G' in the new group set.

3.2 Groups

KernelSec groups are a means of organizing sets of users. But unlike traditional systems, *kernelSec* groups are maintained by the same mechanisms providing ordinary object protection. The *kernelSec* group mechanism is based on the abstract mechanism defined in [30]. We include a brief (and somewhat simplified) description here so that the paper is

self-contained and also because the *kernelSec* groups differ in some respects from their abstract counterparts. The advantages of this group mechanism are that any user can create a group; any group of users can be defined to control the membership of the group; and the permissions can be defined to restrict group membership.

Each group in *kernelSec* is a member of one *group set*, which is a collection of related groups. The group set contains a collection of *group objects*—that is, objects having group labels—and each group object in a group set has a unique user ID as its parameter. Hence a user appears at most once in a group set. Group objects are used to determine group membership; relabels are used to change group membership.

Each group set has a *template* which defines how users, or more precisely group objects, are added to the group set and the mapping between group tags and groups of the group set. The group set template has the following components:

group definition: A template’s group tags are unique to that template. The group definition specifies for each such group tag T_g , a rule of the form: $T_g \rightarrow \{g_0, g_1, \dots, g_n\}$ which means if a group object has label $\langle U, T_g \rangle$, then U is a member of each of the groups g_0, g_1, \dots, g_n .

new user: specifies a tag T_g . When a new user U_{new} is added to the system, a group object is created in the group set with label $\langle U_{new}, T_g \rangle$.

initialization: Defines how the groups are populated with users when the group set is created.

user base: An existing group set from which the initial users are drawn.

user map rules of the form $\langle \mathcal{U}, \mathcal{T} \rangle \rightarrow \langle z, T'_g \rangle$: Where \mathcal{U} can either be a specific user or “*” matching any user; \mathcal{T} can either be a specific tag or “*” matching any tag. For each group object in the base group set matching the left hand side of a user map rule, a group object is created in the group set with the same user and tag T'_g .

Relabeling a group object, i.e. changing the tag on the group label, removes the user represented by that group object from zero or more groups and adds her to zero or more groups, all within the same group set. Relabel permissions are associated with *Security Cards*, and collectively the relabel permissions determine how group membership can be managed.

The template can be used to create multiple independent group sets, all of the same form and therefore associated with the same group set template ID. To create a new group set, its name and template ID must be specified. Although a group is specified by a group set name and a group name, for simplicity in this paper we shall omit the group set name.

KernelSec group sets implement many of the properties of RBAC [24] including inheritance, mutual exclusion, restrictions on user-role assignment, and administrative control over who performs user-role assignment.

Example 1: All users. The group set of all users contains a single group

Name	All User Template
Groups	User $\rightarrow \{\text{allUser}\}$
New User	User

and, since it is created before any users are added, does not have an initialization.

Example 2: Users grouped by company. In the group set template to the right, users are either unassigned (with tag **Industry**) or assigned to a particular company. All users in the system are part of this group.

Name	Industry Template
Groups	Industry $\rightarrow \{\text{industryGrp}\}$ Company1 $\rightarrow \{\text{company1Grp}\} \dots$ CompanyN $\rightarrow \{\text{companyNGrp}\}$
New User	<i>Industry</i>
User Base	AllUsers
User Map	$\langle *, * \rangle \rightarrow \langle \cdot, \text{Industry} \rangle$

3.3 Privileges

A *privilege* is a pair $\langle \text{permission}, \text{label} \rangle$, where *permission* corresponds to the operation (such as read, write, etc) that can be performed on the object with label *label*. Hence the privileges that a process has determines the set of allowed operations for that process.

KernelSec supports five types of privileges on ordinary objects: For files, these privileges are read, create, write, execute, and relabel. For directories, these privileges are search, insert, delete, traverse, and relabel.

Name	File	Directory	Specification
read	read	search	$r_* \langle t \rangle$
write	write	delete	$w_* \langle t \rangle$
execute	execute	traverse	$x_* \langle t \rangle$
create	create	insert	$c_g \langle t \rangle$
relabel	relabel	relabel	$rl_* \langle t \rightarrow t' \rangle$

Table 1: Files and directory privileges

Table 1 shows the *kernelSec* privileges for ordinary objects. All of the privileges are defined in terms of their label’s tag, denoted by t or t' . For read, write, and execute the permission (r, w or x) and a tag t are specified. For create, the permission (c), a group set template ID, g , and tag are specified. If the group set template ID is non-zero, then when an object is created with that tag, a group set is also created and the new group set’s ID becomes the parameter of the object’s MAC label. Finally, a relabel privilege specifies two labels, and can be used to change the tag on a label from t to t' .

The only privilege on group objects is a relabel of the form

$$rl_{\mathcal{P}} \langle t \rightarrow t' \rangle$$

where \mathcal{P} can either be a “*” matching any user’s group object or “*u” matching only the group object (if any) of the user on whose behalf the process is executing.

We note that SELinux has a mechanism called **type_change** which allows relabels in a domain from any of a set of source

labels to any of a set of destination labels, thus defining a single many-to-many relabel map. *KernelSec*'s relabel permissions are more detailed, allowing each *Security Card* to specify a set of source-destination label pairs, thus defining an arbitrary directed graph of relabels. The effect is that a single *Security Card* can specify an arbitrary number of relabel permissions while in SELinux, multiple domains would, in general, be required.

3.4 Security Cards

At any given time, each process has a single *Security Card* which determines the process's privileges. A *Security Card* transition, changing the current *Security Card*, may be allowed if (a) the current *Security Card* has insufficient privileges to perform a requested operation, (b) there is a successor to the current *Security Card* containing the missing privilege and (c) the process's user is a member of one of the groups in the successor *Security Card*. (See the *security method* below for complete details). Each user, in the system, has an *initial Security Card* and every new (login) session for that user starts with her initial *Security Card*.

The components of a *Security Card* are:

Groups: The groups of users whose processes can use this *Security Card*.

Privileges: The list of privileges the process has, which are each of the form described in Section 3.3¹.

Security Method: The security method is invoked when a process attempts to perform an operation for which the current *Security Card* does not have sufficient privileges. When invoked, the security method can both perform actions (to record changes to the authorization state) and (optionally) switch *Security Cards*. These transitions are called *active transitions* as they can perform changes to the authorization state as part of the transition.

The security method is composed of a series of pairs. Each pair consists of *match section*—which describes a missing privilege—and an *action section*—which contains a sequence of actions to be performed atomically. The match section describes a list of one or more privileges, using the same form of privilege specification as in the privileges section of the *Security Card* (see above). Each action section consist of: A set of zero or more **relabels** (**group_relabel** for group objects and **ordinary_relabel** for MAC objects) and ultimately either

- a **switchto** (which changes the *Security Card* associated with the process) or
- a **usePrivilege** (which allows the operation to proceed without switching *Security Cards*).

¹ In addition, the negation of that form, indicated by a preceding “!” is allowed, but is not used in any of our examples here. The first privilege that matches applies. The ordering and negative permissions provide a mechanism for differencing—that is, all those that match a privilege except those that match a negated predecessor of the same privilege.

The set of actions, specified from the point of the security method invocation until the last action for a given matched privilege, are performed atomically—that is, either they must all succeed and the process is allowed to perform the operation or none can and the operation fails.

Example 3: Simple Security Card which allows the process to read any ordinary object whose tag is **Stuff**. If a user tries to read or write an ordinary object with tag **Z**, it will switch the *Security Card* to **Zcard** (we presume that **Zcard** has sufficient privileges, if not the operation will fail and the *Security Card* will not be switched). Finally, if any other privileges are needed the operation will fail.

Name	Simple card
Groups	allUserGroup
Privilege(s)	$r_*\langle\text{Stuff}\rangle$
Security Method	$r_*\langle Z\rangle, w_*\langle Z\rangle$: switchTo(Zcard);

3.5 KernelSec operations affecting the authorization state

The new OS operations which modify the *kernelSec* authorization state are:

group_relabel(*gs*, *u*, *t*) changes the group tag to *t* on a user *u*'s group object in group set *gs*, thereby affecting the group memberships for that user.

ordinary_relabel(*o*,*t*) changes the MAC tag to *t* on a MAC object *o*.

create_group_set(*t*) creates a new group set using an existing group set template *t*, it returns the group set ID of the newly created group.

4. EXPRESSIVENESS

In this section, *kernelSec* expressiveness is explored through the following series of examples:

System Administrators classifies users as either system administrators or ordinary users.

Chinese Wall implements the classic Chinese Wall protection model.

Dynamic Information Flow describes how a process changes the level of created objects as a result of its access history.

Dynamic Separation of Duty to implement the classic purchase payable workflow.

We note that static versions of some of these mechanisms can be implemented in TE, notably, Chinese Wall and Information Flow. However, changes to the authorization state (configuration) must be made by trusted system administrator, and are outside the scope of TE. Hence, TE doesn't implement administrative controls which is needed to implement system administrators example.

Name	OrdinaryAndSysAdmins Template
Groups	sysAdmin → {sysAdmin} limbo → {limbo, sysAdmin} ordinary → {ordinary, limbo, sysAdmin}
User Base	User base group set name
User Map	$\langle *, * \rangle \rightarrow \langle \cdot, sysAdmin \rangle$
New User	<i>ordinary</i>

Figure 1: Group set template example.

Name	SysAdmin card
Groups	sysAdmin
Privilege(s)	$rl_*(ordinary \rightarrow sysAdmin)$, $rl_*(limbo \rightarrow sysAdmin)$, $rl_*(limbo \rightarrow ordinary)$
Security Method	$rl_*(sysAdmin \rightarrow limbo)$: switchTo(LimboCard);
Name	Limbo card
Groups	limbo
Privilege(s)	$rl_*(sysAdmin \rightarrow limbo)$, $rl_{*u}(limbo \rightarrow sysAdmin)$
Security Method	$rl_*(* \rightarrow *)$: switchTo(SysAdminCard);

Figure 2: Security Cards for SysAdmin and Limbo.

4.1 System Administrators

Our first example primarily illustrates groups. The group template is shown in Figure 1 and the *Security Card* is shown in Figure 2. Its goal is to separate the users into two categories, system administrators and ordinary users. New users are added as ordinary users, and then may be relabeled to be system administrators. Similarly, system administrators can be relabeled to be ordinary users.

However, we wish to safeguard against accidents. For example, the sole system administrator becoming an ordinary user (resulting in a system which cannot be administered) or an error which causes a system administrator to block a system administrator from using the system. Hence, the demoting of a system administrator to an ordinary user takes place in two steps: the first is a move to LIMBO which is then followed by a move to ORDINARY. Only a sysAdmin can move a user from LIMBO to ORDINARY. Hence, if a sole sysAdmin changes her status to LIMBO, she cannot be made into an ORDINARY user (as there are no sysAdmins left to do the relabel), but she may relabel herself back to sysAdmin using the limbo *Security Card*.

4.2 Chinese Wall

Chinese Wall is a security model for investment banking and other industries which hold confidential information on clients [6]. Chinese Wall allows an investment bank to hold information on competing companies—those within the same industry—but prevent any individual from having information on two competing companies. Hence, Chinese Wall ensures that each user may access at most one company in each industry.

In Example 2, the groups for a particular industry within

Name	Industry card
Groups	All Users
Privilege(s)	$rl_{*u}\langle Industry \rightarrow Company1 \rangle$, $rl_{*u}\langle Industry \rightarrow Company2 \rangle$, ... $rl_{*u}\langle Industry \rightarrow CompanyN \rangle$
Security Method	$r_*(Company1)$, $w_*(Company1)$: group_relabel(*g, *u, Company1), switchTo(Company1 Card); ... $r_*(CompanyN)$, $w_*(CompanyN)$: group_relabel(*g, *u, CompanyN), switchTo(CompanyN Card);
Name	CompanyI card
Groups	CompanyIGrp
Privilege(s)	$r_*(CompanyI)$, $c_*(CompanyI)$, $w_*(CompanyI)$
Security Method	

Figure 3: Security Card for the industry and companyI.

Chinese Wall were shown. The *Security Cards*, discussed next, will ensure the correct temporal (i.e. dynamic) properties.

There are two *Security Cards* to implement Chinese Wall as shown in Figure 3. The first is an industry card. The industry card does not have the privilege to read or write any object. Hence, if an attempt is made to access a company’s file, the security method is executed, which attempts to relabel the user’s group object tag to that of the company accessed. This succeeds only if the user has either already accessed files of that company (the group object has the company tag) or the user hasn’t accessed any company. Also, since the *Security Cards* only allow relabels from industry to company groups, each user can select a company in that industry at most once. If the relabel is successful, it is followed by a switch to that company’s card, which has the necessary privileges. The CompanyI Card shown in the figure is an example of a typical company card.

One of the interesting parts of this scheme is how flexible the implementation is. The solution presented has implicit selection of companies as a side effect of performing reads. Alternatively, an explicit relabel can be required before successfully invoking the security method to switch to the company’s *Security Card*.

4.3 Information flow

Controlling information flow is a fundamental problem in computer systems as it directly affects confidentiality [2] and integrity [4]. It is also effective in controlling malware, as it can prevent malware propagation.

We present a mechanism which can implement lattice-based access controls [23]; like TE it is not described with a lattice and it directly allows downgrades². Hence it can be used

² We note that MilSec systems also, of necessity allow downgrades but this mechanism is outside of the lattice-based description, making it, we believe, cumbersome for non-MilSec

Name	Base card
Groups	baseGroup
Privilege(s)	$r_*(base)$, $w_*(base)$
Security Method	$r_*(Confidential)$, $w_*(Confidential)$: switchTo(Confidential);
Name	Confidential card
Groups	confidentialGroup
Privilege(s)	$r_*(base)$, $r_*(confidential)$, $w_*(confidential)$
Security Method	

Figure 4: Dynamically limiting information flow

to prevent information from being copied into objects which might be read by more than the original group of users or to prevent information flowing in from unvetted sources and thus contaminating high quality information.

In Figure 4 the *Security Cards* are shown for two levels at which the process can operate. The process starts out with a Base Card and as long as it only accesses objects with label **base**, the *Security Card* doesn't change. However, if a **confidential** object is accessed, then the security method is invoked which if the user is a member of the Confidential group, switches to the Confidential card after which base objects can no longer be written.

4.4 Dynamic Separation-of-Duty

Consider a task consisting of multiple steps. *Separation-of-Duty (SoD)* requires that different steps be performed by different individuals to prevent fraud or malfeasance. One form of this rule is *Static SoD (SSoD)*, in which the users are divided into disjoint groups and different steps are performed by members of different groups. SSoD is easy to implement in almost all operating systems. On the other hand, in *Dynamic SoD (DSoD)* individuals are not a priori partitioned into groups, yet a given individual is prevented from performing more than one step in a task: DSoD is more flexible but more difficult to implement. In fact, the only other kernel-based project that implements DSoD, of which we are aware, is DTAC [32]. Our use of groups is similar to Crampton's blacklist sets [10] but is built upon a general purpose group mechanism.

Name	PO Template
Groups	IssuingClerk → {Issuer}
	ShippingClerk → {PkgSigner}
	EndUser → {User}
	PayablesClerk → {Payer}
	None → {None}
User Base	User base group set name
User Map	$\langle *, * \rangle \rightarrow \langle ., None \rangle$
New User	None

Figure 5: Groupset template for dynamic SoD

We consider a DSoD example of purchase orders. Its purpose is to make it harder for a clerk to create a phony sup-
applications

plier to bill his employer. The steps are: (1) **create**: the creation of a purchase order by a Purchase Clerk; (2) **receiveItem**: the receipt of the item by a Clerk in the Shipping and Receive department; (3) **endUserApproval**: the receipt of the item by the end user and approval upon inspection; and (4) **payment**: the payment for the item by an Accounts Clerk.

For each Purchase Order (PO), the configuration has a MAC PO object, the parameter of this object's label specifies a group set created for that PO and used to track which clerk is performing each step. Figure 5 shows the group set template **PO Template** used to create the group set. The tag part of the PO Object's label indicates the current state of the PO, which is one of **IssuedPO** (the PO was issued), **RcvdShipping** (received by the Shipping dept.), **RcvdEndUser** (received by the ultimate user), and **Paid**. There are 4 groups of interest, one for each step of the task. For space reasons, in Figure 6, we only show the *Security Card* for the ShippingClerk, who receives the package.

For example, assume **P09834** is a PO Object for an item which has just been received on the shipping dock; To indicate the package's receipt, the clerk's process executes:

```
ordinary_relabel('P09834', RcvdShipping);
```

The purpose of this call is to change the tag part of the label on the PO object named "P09834", to **RcvdShipping**; for the receive to be valid, a purchase order must have been issued for the object (**IssuedPO**) and the clerk must not have performed a previous step with respect to "P09834". Since the *Security Card* does not have sufficient permissions to do the relabel of the PO object, it enters the security method in which

- the **group_relabel** succeeds only if the clerk has not performed any operation in the sequence (and therefore his group object tag is **None**) and
- the **usePrivilege** succeeds only if the current tag of the PO object is **IssuedPO**.

Note that the restrictions are relative to a given purchase order, so that the same clerk can perform any (single) step on any other purchase order.

Name	Shipping Clerk's card
Groups	Anyone
Privilege(s)	$rl_{*u}\langle None \rightarrow PkgSigner \rangle$
Security Method	$rl_{*g}\langle IssuedPO \rightarrow RcvdShipping \rangle$: group_relabel(*g, *u, PkgSigner); usePrivilege ();

Figure 6: Security Card for the step where the shipping clerk records the receipt of the item

Given this *Security Card*, it is impossible to change the purchase order's tag without the clerk meeting DSoD constraints. The subsequent steps in the approval sequence have

similar *Security Cards* to the one shown for the `receiveShipping` step. Note that the atomicity of security method invocation and active transitions are essential to implement dynamic separation of duty’s paired operations.

4.5 Discussion

We have presented familiar authorization models as they are easier to understand and make better use of our limited space. However, it is worth highlighting some issues where we differ from other authorization systems:

- The group mechanism is very flexible, providing fine grained control over who can regulate group membership, the relationship between groups (partitioning or hierarchy) and the successor groups that a user can belong to, given the current group.
- Chinese Wall is a simple combination of group membership and information flow properties. Many variants of Chinese Wall can be encoded.
- Information flow type problems can be written allowing
 - floating labels: for example, allowing an object labeled Confidential to be automatically relabeled to Secret when writing it after reading a Secret object. (The degree of floating, if any, can also be completely controlled).
 - floating process level: the process effectively works at a higher level via a switch in *Security Cards*.
 - Downgrade can be allowed via relabel, transitivity is not required.
- Dynamic separation of duty
 - Relabels ensure that all users are, for example, agreeing to the same purchase order since no user has write privileges.
 - Relabels of the MAC tags allow arbitrarily complex DSoD problems to be specified including those with loops or alternative outcomes.
- all (overt) channels are mediated in *kernelSec*, including pipes, message queues, and shared memory. We have not tried to prevent covert channels [13, 19, 18]. One reason is that covert channel prevention is expensive in terms of validation and performance. But the larger problem with covert channel analysis is its effectiveness, since it requires a unidirectional flow of information. This is possible only in MLS. We note that covert channels can be eliminated with NRL pumps [16] between systems running at a different level.

5. IMPLEMENTATION

We give just a brief description of the implementation here; complete implementation details are intended to be presented as an OS-oriented paper. *KernelSec* is implemented in Linux, and consists of two components:

kernelSecKernelModule a Linux kernel module, which is dynamically loaded into the Linux kernel and implements its protections primarily using the facilities of Linux Security Modules (LSM) [37].

Operation	<i>kernelSec</i> (μ s)
verify privilege	4.59
switch to	30.7
relabel	1541

Table 2: Op Performance

Example	Time (μ s)
Information Flow	17
Chinese Wall	1345
Dynamic SoD	2822

Table 3: Example Performance

kernelSecD a user-space daemon which downloads the configuration into the **kernelSecKernelModule** using netlink, and is also responsible for the creation of new group sets.

The **kernelSecKernelModule** is currently about 6500 lines of C code, while **kernelSecD** is currently about 7300 lines of C++ code. The current code implements all of the protections described in this paper.

Because kernel implementations affect all programs, it is important that kernel mechanisms have low overhead. Thus, we present here a few microbenchmarks to measure the *kernelSec* performance overheads.

We benchmarked three operations: (1) verify that a card has the required privilege, (2) switch to a new *Security Card*, and (3) relabel a group or ordinary object. Operations (2) and (3) are done as part of the security method. These benchmarks measured the elapsed time at the application level; for the higher cost operations a context switch is performed and other work is done while waiting for the operation to complete. Table 2 gives the time in microseconds for a few operations; for the most common Unix operations, the *kernelSec* percentage overhead is in the single digits. The tests were carried out on a 2.8 GHz Pentium 4 machine (with 512 MB RAM) running Linux kernel version 2.6.10.

Figure 3 gives the elapse times for the examples discussed in Section 4. For the cases of Information Flow and Chinese Wall, the operation was a `open()` on a MAC object, which involved just a `switchTo` in the former and a relabel and `switchTo` in the latter. For the Dynamic SoD example, two relabels were needed one a MAC relabel and the other a group relabel.

The overheads are modest even in our unoptimized implementation. The one expensive operation is relabels, which must be synchronously written to disk to avoid security holes. Hence, the cost includes writing the object’s inode (which contains the label) to the disk controller before returning to the application, preventing protection from being bypassed even due to system crashes. (The CPU time is much smaller, on the order of 30μ s, so that useful work can be performed while waiting for the write out to disk.) However, we believe that group operations will be relatively rare and hence the impact on overall program performance

even with a sync is very modest.

6. REDUCING COMPLEXITY

We now describe how the goals of expressiveness and reduced complexity are addressed in *kernelSec*.

6.1 Factoring

it may be tedious to manually create the *Security Cards* because of (1) the number of *Security Cards* and (2) the number of transitions between *Security Cards*. We are particularly concerned about what happens when the authorization configuration is changed, for example, to allow a new application.

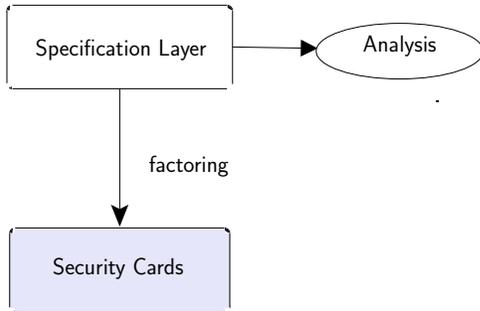


Figure 7: Overall system design with high-level specification which is (1) factored into *kernelSec* layer and (2) analyzed for security properties.

Hence, our security model has two levels: A high level authorization specification such as RBAC [11, 24] or SPBAC [31] and a kernel-based enforcement mechanism which implements these protections. The authorizations are described at a high level and then *factored* into *Security Cards*. Although a detailed discussion of the high level specification and the algorithms for factoring are beyond the scope of this paper, this separation enables the *Security Cards* to be optimized for run-time efficiency; the specification level is optimized for specifying and understanding the authorizations.

For example, an SPBAC specification layer is stateless, it uses permissions to implement constraints. A process must hold multiple permissions to perform certain operations; these permissions can be viewed as simultaneously requiring different constraints to hold. In general, small changes in the specification layer results in many more changes in the number of, and the transitions between, *Security Cards*. Hence, the specification level is more stable and the changes more incremental.

In addition to reducing the authorization specification, it enables *Security Cards* to be viewed as pure mechanism. Hence, we are not concerned here with analyzing whether *Security Cards* are well formed with respect to implementing given protections—only that they are sufficiently expressive to implement them.

6.2 Authorization Properties

In order to obtain least privilege, systems have historically partitioned privileges. For example, SELinux has approx-

imately 80 privileges associated with ordinary (non-root) permissions. We have significantly reduced the number of privileges because we (1) do not consider covert channels and (2) by organizing privileges around authorization properties.

An *authorization property* characterizes allowed executions, for example information flow confidentiality, separation of duty, or restricting the update of certain objects to an executable. Hence, each permissions is used to enforce some authorization property and this provides a basis for determining what permissions are needed. An authorization property approach provides not only a metric for choosing the granularity of permission, but also for ensuring that the permissions defined are well suited for the security being enforced.

In the context of our broader authorization project, we have been studying the design of authorization properties. One of the issues has been how to combine administrative controls with authorization properties: we have described this for the oldest of the authorization properties, information flow confidentiality and integrity [31, 29]. Another type of authorization property that we have been investigating is the well formedness of dynamic separation of duty problems—that is, they do not get stuck because they run out of users who are eligible to perform needed tasks [28].

7. RELATED WORK

One of the earliest MAC mechanisms in operating systems are Lattices [36, 22, 2]. For example, LOMAC [12] has a special purpose mechanism to enforce Biba integrity in which the level of the process changes, which could be implemented in *kernelSec* with a *Security Card* transition. Compartmented Mode Workstations (CMW) [3], which have both current and maximum lattice labels for each object, can dynamically relabel the current object for increased flexibility. But their dynamic semantics were hardwired, whereas *kernelSec*'s are individually specifiable. While lattices are an elegant formalism, they provide only limited forms of protection.

TE provides the ability to both require that certain authorization properties hold, such as assured pipelines or overt information flow, while enabling these properties to be selectively violated when necessary, as for declassification operations. But TE's domain transitions are very static, requiring the domain to be chosen before performing the operations, and hence requiring the user to understand the protection configuration. *Security Cards* can be viewed as a generalization to TE. The *kernelSec* project further differs from TE implementations in (1) our focus on more abstract objects such as our group structure and (2) the emphasis on providing authorization properties.

Our groups provide a number of RBAC facilities [24] within group sets including inheritance (the tags of one group are a subset of those of another), mutual exclusion (the tags are disjoint), and user-role management (the users who control group membership). However, our groups differ from RBAC roles both in the way they are constructed and administered.

Varadharajan and Allen investigated the requirements of implementing Joint Action (including Separation of Duty) in

an Operating System [34].

A project which had similar goals to *kernelSec* is DTAC which adds constraints to the system to provide a more dynamic TE [33, 32]. Their system relies heavily on explicit constraints [14]. In contrast *kernelSec* uses permissions and *Security Card* transitions as a type of implicit constraint. As a result we believe *kernelSec*'s mechanisms are simpler, more natural, and more efficient.

EROS is a capability-based system [25] which also supports fine grain mechanisms. In EROS the focus is on providing least privileges while *kernelSec* is oriented at providing authorization properties at a high level of abstraction, although there is some support for mechanisms which enable, for example, *MilSec* information flow rules [26].

For alternative approaches to provide functionality at the OS kernel interface, see for example Jain and Sekar [15].

A complementary approach to ours is *Privtrans* which seeks to automatically partition application code into privileged and unprivileged components [7].

We believe that *kernelSec* is the first operating system protection model to be implemented which provides (1) structured groups, (2) support for dynamic separation of duty, (3) support for flexible information flow, and (4) support for a general relabel mechanism.

8. CONCLUSION

System-wide security cannot be provided at the application level, and hence can only be provided in the operating system. For an OS-based mechanism to provide the necessary security, it should be sufficiently expressive to capture a variety of security policies and yet not be overly complex.

To this effect, the *kernelSec* model is implemented at two levels: the system specification level (level 1) supports DAC, Mandatory Access Control (MAC), and administrative controls and has decidable authorization properties. The enforcement engine (level 2), using *Security Cards*, provides the run-time engine which implements the specified protections. This separation paves the way to have an efficient mechanism and yet have a simplified means for encoding specification.

We have described *Security Cards*, the filesystem permissions supported, and *kernelSec*'s group mechanism. *Security Cards* can be seen as a generalization of Type Enforcement with the following differences: It supports arbitrary relabel privileges; it has the ability to specify privileges as patterns; and allow operations to be performed as part of the transition mechanism between *Security Cards*.

Key to the *kernelSec* enforcement engine is the group structure which, when combined with *Security Cards*, allows the definition of group sets (collections of related groups), the creation of group sets, and membership administration to be unprivileged operations. Moreover, group sets allow new users to be automatically added to the group set. Relabel permissions constrain how group membership can change and are a powerful mechanism.

These mechanisms substantially increase the dynamic properties of the systems, enabling unified DAC/MAC support, dynamic separation-of-duty, dynamic information flow, and the ability to provide rich variants of many protection schemes. We have implemented *kernelSec* in the Linux Kernel using Linux Security Modules.

KernelSec is effective in minimizing the awareness that users and application developers need to have about the authorization configuration. By allowing the permission to change based on program actions (using *Security Card* transitions), *kernelSec* is able to support least privilege without being overly restrictive. By being able to encode changes to the authorization state to be part of the authorization configuration, the dependence on the application developer to incorporate the protection in the application can be greatly reduced.

9. ACKNOWLEDGEMENTS

The authors would like to thank Damian Roqueiro for his careful reading of an earlier draft of this paper as well as Ashley Poole, Prof. V. N. Venkatakrishnan, Jorge Hernandez-Herrero, Kevin Kahley, Mike Ter Louw, Hareesh Nagarajan, and the anonymous referees for their comments.

10. REFERENCES

- [1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghihat. A domain and type enforcement UNIX prototype. In *Proc. of the USENIX Security Symposium*, Salt Lake City, 1995.
- [2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Mitre Corporation, Bedford MA, 1973.
- [3] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, 1990. Special Section on Security and Privacy.
- [4] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.
- [5] W. E. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *8th National Computer Security Conference*, pages 18–27, 1985.
- [6] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proc. IEEE Symp. Security and Privacy*, pages 206–214, 1989.
- [7] D. Brumley and D. X. Song. *Privtrans*: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.
- [8] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. of the USENIX Security Symposium*. USENIX, 2002.
- [9] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symp. Security and Privacy*, pages 184–194, 1987.

- [10] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 43–50. ACM Press, 2003.
- [11] D. F. Ferraiolo and R. Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, 1992.
- [12] T. Fraser. LOMAC—low water-mark mandatory access control for Linux. In *Proc. of the USENIX Security Symposium*, Washington D.C., 1999.
- [13] V. Gligor. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, Ft. George G. Meade, Maryland, U.S.A., Nov. 1993. Approved for public release: distribution unlimited.
- [14] T. Jaeger. On the increasing importance of constraints. In *Proc. of the ACM Workshop on Role-Based Access Controls (RBAC)*, pages 33–42, 1999.
- [15] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*, 2000.
- [16] M. H. Kang, A. P. Moore, and I. S. Moskowitz. Design and assurance strategy for the NRL pump. *Computer*, 31(4):56–64, 1998.
- [17] B. Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, 1971.
- [18] B. W. Lampson. A note on the confinement problem. *Communications of the ACM (CACM)*, 16(10):613–615, 1973.
- [19] J. Millen. Twenty years of covert channel modeling and analysis. In *Proc. IEEE Symp. Security and Privacy*, pages 20–114, 1999.
- [20] R. O’Brien and C. Rogers. Developing applications on LOCK. In *Proc. 14th NIST-NCSC National Computer Security Conference*, pages 147–156, 1991.
- [21] D. of Defense. Trusted computer system evaluation criteria. Technical Report DOD 5200.28–STD, U. S. Department of Defense, 1985.
- [22] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [23] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.
- [24] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
- [25] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP’99)*, pages 170–185, 1999.
- [26] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. IEEE Symp. Security and Privacy*, pages 166–176, 2000.
- [27] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised April 2002.
- [28] J. A. Solworth. Approvability. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS’06)*, page to appear, Taipei, Taiwan, May 2006.
- [29] J. A. Solworth and R. H. Sloan. Decidable administrative controls based on security properties, 2004. Available at <http://parsys.cs.uic.edu/~solworth/kernelSec.html>.
- [30] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable properties. In *Proc. IEEE Symp. Security and Privacy*, pages 56–67, 2004.
- [31] J. A. Solworth and R. H. Sloan. Security property-based administrative controls. In *Proc. European Symp. Research in Computer Security (ESORICS)*, volume 3139 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2004.
- [32] J. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 154–163, 2000.
- [33] J. F. Tidswell and T. Jaeger. Integrated constraints and inheritance in DTAC. In *Proc. of the ACM Workshop on Role-Based Access Controls (RBAC)*, pages 93–102, 2000.
- [34] V. Varadharajan and P. Allen. Joint actions based authorization schemes. *Operating Systems Review*, 30(3):32–45, 1996.
- [35] R. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*, Boston, MA, 2001.
- [36] C. Weissman. Security controls in the ADEPT-50 time-sharing system. *Proc. FJCC, AFIPS*, 35, 1969.
- [37] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux Kernel. In *Proc. of the USENIX Security Symposium*, San Francisco, Ca., 2002.