Factoring High Level Information Flow Specifications into Low Level Access Controls

Kevin Kahley Manigandan Radhakrishnan Jon A. Solworth

University of Illinois at Chicago
{kkahley, mradhakr, solworth}@cs.uic.edu

Abstract

Low level access controls must provide efficient mechanisms for allowing or denying operations and hence are typically based on the access matrix. However, when combining the goals of efficiency along with the support for least privilege and higher level authorization properties (such as information flow confidentiality), the resulting access controls become tedious to encode.

Compositional high level specifications can be much more succinct. When combined with administrative controls, they can be robust in changing what is authorized in a controlled manner. Such specifications offer the promise of being easier to configure and understand, and in fact can be automatically analyzed for authorization properties.

However, there remains the issue of how to generate the low level access control configuration from the high level specification. In this paper, we describe a factoring algorithm to algorithmically translate a high level specification of information flow authorization properties into low level access controls. In addition, several optimizations are given which dramatically reduce the size of the access control configuration generated.

1. Introduction

Sophisticated access control systems such as Role-Based Access Controls [6, 21, 20] and Type Enforcement [4] enable the specification of many authorization properties such as information flow confidentiality, information flow integrity, and limiting the executables which can access sensitive data. The ability to precisely control what is authorized enables, in principle, applications to be well protected.

The price for such control is increased complexity in specifying what is authorized. If the authorizations are too difficult to specify, then either they are never used (newer OS-based authorization systems are typically added on top of existing OSes) and when used, the process of specifying the security of a system becomes cumbersome and error-prone. Obviously, an unused system provides no benefits. There are two types of authorization specification errors: those which *over-authorize*—allowing actions to occur which should be denied—and those which *under-authorize*—forbidding actions which should be allowed. Errors which over-authorize subject the system to additional attacks and are difficult to detect. Errors which under-authorize are relatively easy to detect since legitimate actions are denied. Under-authorization errors can also result in vulnerabilities, as users try to circumvent the system in order to complete their work. Obviously, reduction in complexity reduces errors and increases use.

Over time, an organization's changing needs will require changes to what is authorized. Traditionally, systems have provided backdoors requiring totally trusted users to make such modifications, for example, superuser privilege in Unix. Such backdoors make the system extremely vulnerable to betrayal by these trusted users and to the errors they might make. As an alternative, administrative controls allow for controlled changes to what is authorized and limit trust in those individuals to whom the organization is most vulnerable. Administrative controls too have added complexity.

In this paper, we describe a multi-tiered authorization system that we are developing which is intended to support a broad suite of authorization properties. We focus here on the oldest of such authorization properties, the information flow properties of confidentiality

and integrity[1]. Our design is three tiered, consisting of an administrative level, a high level specification, and a low level, Operating System Kernel enforcement engine. Our authorization system is implemented in the Linux kernel, primarily as a Linux Security Module (LSM) [26] and constitutes some 13,000 lines of code.

The high level specification is stateless and is intended to be easy to understand. To enable succinct and comprehensible specifications, the high level is composable. A *composable* authorization system is described by *components* together with rules for combining the components and thus to determine what is authorized. The composition produces the sets of privileges that a process can possess at any instant, in other words an access matrix [15] (which can be viewed as a snapshot of any system). The advantage of a composable system is that a single change to one of the components can result in many changes to access matrix permissions; comparable semantics are more awkward in non-composable systems. Another example of a composable system is RBAC'96 [21].

Access matrix-level implementations are more efficient and hence desirable as run-time enforcement engines. But since they are cumbersome to encode and since small changes in requirements can lead to large number of changes in the number of rows in the access matrix and the permissions, it is desirable to have a higher level specification.

In this paper, we describe the automatic translation—which we call *factoring*—from high level specification to the runtime engine. An algorithm is given to factor the high level specification into the *kernelSec* layer; we then described optimizations which reduces the number of *kernelSec* elements generated.

The rest of the paper is organized as follows: In Section 2 we describe the three-tiered system. In Section 3 the basic algorithm to factor the high-level specification into the *kernelSec* layer is given. In Section 4 the optimizations that reduce the number of elements resulting from factoring are discussed. In Section 5 we describe related work and finally we conclude in Section 6.

2. System overview

Our overall system consists of three tiers:

administrative controls which enables the permissions of the system to be changed in a controlled way;

high-level specification describes the *current* permissions of the system (sufficient to implement information flow); and

kernelSec the low-level run-time engine implemented in the operating system kernel.

These tiers share mechanism between them, but serve different needs: the administrative controls allow the system to change over time; the high-level specification is intended to make it (relatively) easy to specify and to analyze the current system authorizations; and the *kernelSec* level is intended to be efficient at run-time and small enough to fit in an operating system kernel.

We note that the high level mechanism (and the combination of high level mechanism plus administrative controls) is composable. To enable changes to propagate as necessary during composition, indirection (e.g., objects have labels, labels map to privileges) and reuse (e.g., same label may be used on multiple objects) are used. If the system encapsulates the correct abstractions, the composition naturally provides the needed privileges and changes are relatively easy.

To reduce errors in authorization configuration and modification, we build upon recent advances of system components with decidable authorization properties [25, 18]. The decidability enables the high-level specification to be analyzed, to answer questions such as: "Can information in this object ever become publicly available"? The ability to automatically answer such questions effectively reduces the complexity of the system because it is easier to understand what can happen and thus the system is less error prone. It enables a separation between those that specify the configuration of the authorization system—e.g., an outside contractor—vs. those that determine whether the system is appropriate.

The decidability of our system plays another important role beyond determining how authorizations can evolve. The administrative controls of our system require analysis to determine what approvals are necessary to make changes to what is authorized. The decidability ensures that this analysis is precise and therefore well defined.

Another mechanism reducing complexity is the factoring of the high level specification into the *kernelSec* tier. The overall system is shown in Figure 1.

The three tiers overlap. The middle layer contains a current snapshot, and on top of that the administrative controls are built. The *kernelSec* layer is produced from the high-level specification. Hence, we next describe the details of the tiers in the order of high-level specification, administrative controls, and then the *kernelSec* layer.

[1] We consider only overt information flow, covert flows are beyond the scope of this paper

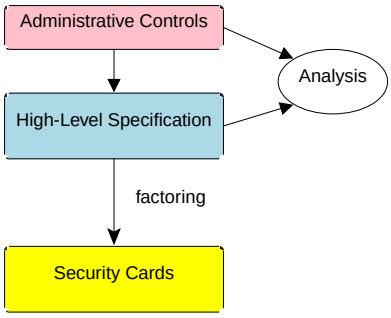

Figure 1. Factoring SPBAC into Security Cards

2.1. The high level specification

Each object has a label which defines it protection type. Associated with each label are three unary permissions:

r(l) the group of users who can read objects labeled l;

w(l) the group of users who can write objects labeled l; and

x(l) the group of users who can exec objects labeled l.

The execute permissions are listed here for completeness, and although they are needed to perform an `exec` we shall ignore them for the rest of this paper.

In addition there is a binary permission

$$mayFlow(l, l')$$

which is the group of users who can write l' after having read l. Note that for each label l that the process has read, the user must be a member of $mayFlow(l, l')$ to write l'; in addition the user needs the ability to both read l and write l'. Hence, to write l' given that rs is the set of labels read by the process, it is necessary that for all l in rs:

$$r(l) \cap mayflow(l, l') \cap w(l')$$

By convention we shall not explicitly define $mayFlow(l, l')$ but shall implicitly define $mayflow(l, l) = w(l)$—thus allowing flow between objects of the same label by a user which has both read and write permissions on l. The justification for not regulating information flow between objects with the same label is that authorization is at the granularity of labels (i.e. objects with the same label are not distinguished in terms of permissions). Hence for information flow between two object with the same label, the user needs to be able to only read and write that label.

Since mayflows need not be defined between all labels, we shall use the notation $mayflow(l, l') \downarrow$ to mean that $mayflow(l, l')$ has been defined.

We note, although not covered in detail here, that the high-level authorization system has a group mechanism which allows the specification of the structure of groups and the (groups of) users who can regulate membership of the group. While the properties of this group structure are important for the purposes of this paper, their detailed structure is not, and we shall not define the group structure here.

The compositional elements include groups (the same group can be used to define different permissions) and labels (the same label can be used on different objects).

Example: Consider the following permission definitions: $r(l_0) = w(l_1) = mayflow(l_0, l_1) = g_0$ and $r(l_1) = w(l_2) = mayflow(l_1, l_2) = g_1$. Then, assuming g_0 and g_1 are non-empty groups, information can flow from l_0 to l_1 and from l_1 to l_2 but not from l_0 to l_2 in a single process (mayflows are not transitive). Also note that information does not flow in the direction of decreasing (label) indices.

2.2. Administrative controls

The administrative controls are dependent on authorization properties, such as information flow confidentiality and information flow integrity. *Information flow confidentiality* means that in every state after information flows from label l to l', the users that can read l' are a subset of those that can read l. *Information flow integrity* requires an integrity inequality ($\geq$), that is, a partial order to be provided between pairs of labels; integrity is maintained for information flow from l to l' if $l \geq l'$.

Note that whether information flow is possible depends on the permissions which are defined in terms of groups. Hence, to answer such questions it is necessary to be able to determine whether a group (or intersection of groups or difference between groups) is nonempty. Our group mechanism enables such relationships to be structurally maintained (in every state) and automatically analyzed—that is, it is decidable—and thus it is possible to guarantee information flow confidentiality even while allowing information flow.

Information flow confidentiality and information flow integrity are always safe but are not always desirable. For example, it is necessary to violate confidential-

ity in real systems (even MLS systems) for downgrades (to allow declassification of information, for example, after sufficient time has elapsed or on when the information becomes public) and sanitization (removing sensitive information from a file, such as the name of informants so that what remains can be released to the public); information flow integrity is violated when cross checking of information from independent sources provides higher quality results than its inputs. Hence, information flow confidentiality and integrity are conservative and need to be violated in real systems. We note that one of the advantages of Type Enforcement systems over lattice mechanisms [1, 3] is that information flow authorization properties can be selectively enforced and violated in the same system.

To support administrative controls, two groups are associated with each label l:

ac(l) The group that can allow new violations to information flow confidentiality. Approval is needed from $ac(l)$ when a new flow is added in which information can flow from an object labeled l to another object which increases readership.

ai(l) The group that can allow new violations to information flow integrity. Approval is needed from $ai(l)$ when a new flow is added in which information can flow into an object labeled l from another object with lower integrity.

Note that violations to information flow confidentiality can occur only through the definition of new mayFlows, while violations to information flow integrity can occur only through the definition of new mayFlows and of integrity inequalities.

If the administrative group $ac(l)$ (resp. $ai(l)$) are permanently empty, then it is impossible to add new violations to information flow confidentiality (resp. integrity) for l. This provides the ability to lock down parts of the system, preventing further changes with respect to that label regardless of administrative action. (By locking down every label, none of the authorization properties can change.)

The administrative controls are not kernel based in the sense that the algorithms for determining when security property approval is necessary are determined in (system) processes. Moreover, the kernel level authorization configuration is downloaded by another system process. Defining new labels and mayFlows may change existing *kernelSec* components, requiring those components to be downloaded into kernel. In our current implementation, administrative changes require a reboot of the system. However, in future implementation a reboot will not be necessary if the permissions associated with a process are not decreased by the administrative actions.

Example: In this example, the groups have fixed membership denoted here by a set of users. Consider a system with two labels, l_0 and l_1 with $r(l_0) = \{u\}$ and $r(l_1) = \{u, u'\}$ and $w(l_0) = w(l_1) = \{u\}$ and $l_0 \geq l_1$. To define a $mayflow(l_0, l_1) = \{u\}$, approval is needed from $ac(l_0)$ because after information flow the set of readers for the information increases; no approval is needed from $ai(l_1)$ because because l_0 is of greater integrity than l_1.

2.3. KernelSec

The *kernelSec* layer is most similar to Type Enforcement. It differs in that (1) transitions are allowed on any missing privilege (not only on `exec` as in Type Enforcement) and (2) it allows user ability to use permissions to be expressed is an intersection of groups which, as we shall see, is useful in factoring. (It has other differences which are not germane to our purposes here[2]).

The primary entity is a Security Card which corresponds to a row in the access matrix. The Security Card consists of the following components:

name the name of the Security Card

groups a list of groups of the form $g_0 \& g_1 \& \ldots \& g_n$. A user must be a member of *each* listed group to use the Security Card.

permissions a list of read, write, and execute permissions on labels, denoted $r\langle l\rangle, w\langle l\rangle, x\langle l\rangle$ that a process has when using the Security Card. The permissions are called the *current permissions* of the Security Card.

security method a sequence of permission-action pairs, where each permission p is not a current permission and the action is a `switchTo(s)` where `s` is the name of a Security Card which contains permission p.

The security method is invoked when a process requests an operation which requires a permission that is missing from the current permissions. If the missing permission is in the security method and the `switchTo(s)` succeeds (the user is a member of each of the groups in `s`) then the current Security

[2] These include arbitrary relabel privileges; active transitions trigger operations to be performed (useful to implement dynamic separation of duty); ability to implement DAC permissions; and the same group mechanism as at the high level specification.

Card is changed to S and the operation is allowed. Otherwise, there is no change and the operation is denied.

Example: This Security Card can be used by any user belonging to the group g_{all}. It has privileges to read the 'Message of the Day (MotD)'. When the user tries to read any accounting data (labeled Accounting) the process switches cards to the Accounting card (with the privilege $r\langle Accounting\rangle$) if the user is a member of the Accounting card's groups. Similarly for personnel data.

Name	Simple User Card
Groups	g_{all}
Privileges	$r\langle MotD\rangle$

Security Method	$r\langle Accounting\rangle$:
	switchTo(Accounting card);
	$r\langle Personnel\rangle$:
	switchTo(Personnel card);

The ability to switch Security Cards, and thus change the set of permission associated with a process based on a missing read or write permission is crucial to our purposes here. Without that capability, a user would have to select in advance a Security Card which would provide sufficient but not excessive permissions[3] and if the permissions on that card do not suffice, the user must invoke a new process with the needed permissions, since the current process is blocked. With the security method's transitions on arbitrary missing privileges, the process's permissions will adapt to the past accesses, enabling new permissions to be acquired on demand as allowed by the high level specification.

[3] In TE, the problem is similar with the word domain replacing Security Card.

Example: Consider a MilSec-like system that enforces simple information flow security using two labels L (Low) and H (High). The system also has two groups g_L and g_H, such that $r(L) = w(L) = g_L$ and $r(H) = w(H) = g_H$.

Name	L Card
Groups	g_L
Privileges	$r\langle L\rangle, w\langle L\rangle$
Security Method	$r\langle H\rangle$: switchTo(H Card) $w\langle H\rangle$: switchTo(H Card)

Name	H Card
Groups	$g_L \& g_H$
Privileges	$r\langle L\rangle, r\langle H\rangle, w\langle H\rangle$
Security Method	

Every new process starts out with the 'L' card. If the process only reads and writes L objects then the process has all the privileges it requires. If it tries to read or write H objects, the security method is invoked and if the user (on whose behalf the process is executing) is a member of both groups g_L and g_H then the switch is performed to the 'H' card.

Each user has an initial Security Card which is associated with her first process. The initial Security Card generated by our factoring algorithm does not have any permissions; its sole function is to provide transitions to subsequent Security Cards which contain the needed permissions as allowed by the high level specification.

If a group's membership changes, such that a process becomes ineligible to use its Security Card (because the user on whose behalf the process executes is no longer a member of the requisite groups for the Security Card), the process is terminated.

Example: Security Cards can be used to implement *assured pipelines*—in which an input (here, a file labeled C that is to be printed) must go through a series of programs (here, labeler and spooler). Assured pipelines, of course, cannot be implemented with a lattice [4].

Name	Labeler Card
Groups	$g_{labeler}$
Privileges	$r\langle C\rangle$, $w\langle Spool\rangle$
Security Method	

Name	Printer Card
Groups	$g_{printer}$
Privileges	$r\langle Spool\rangle, w\langle Printer\rangle$
Security Method	

3. Factoring

In this section we describe how to *factor* the high-level specification into Security Cards. The description is as follows:

- the structure of the Security Cards generated,
- the algorithm for factoring, and
- an example of factoring a high level specification into Security Cards.

3.1. Structure of Security Cards generated

To dynamically modify the set of permissions that a process has, the mayflow semantics require that reads be tracked. Past writes, on the other hand, are irrelevant in determining allowable future actions. To track the set of labels read, each Security Card's read permissions correspond exactly to the set of labels read. Consider two Security Cards, s_0 and s_1: a transition is allowed from s_0 to s_1 only if (a) the transition is on a write and s_0 and s_1 contain the same set of read permissions or (b) the transition is on a read of l and s_1's read permission are the read permissions of s_0 and $r\langle l\rangle$.

Since the writes need not be tracked to determine information flow authorization properties, each Security Card will contain at most one write permission. The Security Card transitioned to will have a write permission only if the transition is on that write permission.

To prevent race conditions, we shall require that a user continue to be a member of each group necessary to perform any read or write. (This prevents temporal aberrations in which the group structure specifies that no user can at any point perform two conflicting operations but—without this restriction—it would be possible to circumvent this specification by performing one of the operations, performing a group membership change, and then performing the other operation.)

3.2. Factoring algorithm

The algorithm is shown in Figure 2. The primary function, `generateSecurityCards` takes as arguments the set of labels (L), and the high level permissions on the labels for read (R), write (W), and mayflow (Mayflow). It iterates over subsets of labels which record the labels read (lines 2-9). The subset of labels read determines which labels could be written if the associated groups contains a user to do so. A Security Card is generated containing just read permissions for those labels; in addition, for each label which can be written (because there exists the prerequisite mayflow) a Security Card is generated with the set of read permissions plus the single write.

The work of generating an individual Security Card is performed by `createSecurityCard`; it has parameters rs (the set of labels with read permission), ws (0 or 1 label with write permission), and L, R, W, and Mayflow as in `generateSecurityCards`. If ws is empty, the user must be a member of each read group in rs; if $ws = \{w\}$ the Mayflows must be satisfied as well as the write permission for w. The read permissions are generated for the labels in rs and the write permissions are generated for the labels in ws.

Finally, the security method transitions are created. There are two types of transitions, one for reads and one for writes. For each read permission not on the card, a transition to a Security Card with one more read permission (and no write permission) is created. For each write permission with the perquisite mayflows, a transition to a card with that write permission (and the current read permissions) is generated.

3.3. Factoring example

In Figure 3, the permissions are given for a lattice consisting of three levels (in order of increasing confidentiality)—`P` for public, `C` for confidential, and `S` for secret—and allowing downgrade from `C` to `P`. The mayflow permissions specification is not sufficient to implement a lattice protection scheme, in addition relationships need to be maintained between the groups, that is

$$g_S \subseteq g_C \subseteq g_P \tag{1}$$

```
1: function GENERATESECURITYCARDS(L, R, W, MayFlow)
2:     for all rs ∈ 2^L do
3:         for all w ∈ L do
4:             if ∀_{r∈rs} mayflow(r, w) ↓ then
5:                 CREATESECURITYCARD(rs, {w}, L, R, W, MayFlow)
6:             end if
7:         end for
8:         CREATESECURITYCARD(rs, {}, L, R, W, MayFlow)
9:     end for
10: end function

11: function CREATESECURITYCARD(rs, ws, L, R, W, MayFlow)
12:     if ws = {} then
13:         GROUPS(⋂_{r∈rs} R(r))
14:     else
15:         ∃w ∈ ws
16:         GROUPS(⋂_{r∈rs}(R(r) ∩ MayFlow(r, w)) ∩ W(w))
17:     end if
18:     for all r ∈ rs do
19:         PERMISSION("r", r)
20:     end for
21:     for all w ∈ ws do
22:         PERMISSION("w", w)
23:     end for
24:     for all l ∈ L do
25:         if l ∉ rs then
26:             SECURITYMETHOD("r", l, rs ∪{l}, {})
27:         end if
28:         if l ∉ ws ∧ ∀_{r∈rs} MayFlow(r, l) ↓ then
29:             SECURITYMETHOD("w", l, rs, {l})
30:         end if
31:     end for
32: end function

33: function SECURITYMETHOD(op, l, rs, ws)
34:     ▷ missing privilege is op, l; destination card is Read # rs # Write # ws # Card
35: end function
```

Figure 2. Security Card generation

Permission		group
r(S)	=	g_S
w(S)	=	g_S
r(C)	=	g_C
w(C)	=	g_C
r(P)	=	g_P
w(P)	=	g_P
mayflow(P, C)	=	g_C
mayflow(C, S)	=	g_S
mayflow(P, S)	=	g_S
mayflow(C, P)	=	g_D

Figure 3. Example of a high level specification

These relationships ensure that the readership gets more selective (is strictly contained within the lower level) with increasing levels of confidentiality. We show the mayFlow graph in Figure 4 corresponding to the permissions defined in Figure 3. In a mayflow graph the nodes are labels and there is a directed edge from l to l' if $mayflow(l, l')$ is defined.

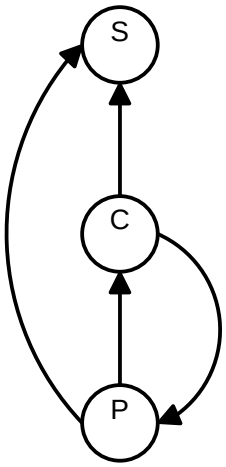

Figure 4. The mayflow graph corresponding to Figure 3.

We next consider the writes which are possible given the set of reads which have already been made by the process. Figure 5 shows the labels which could be written given a set of reads which have been performed. Note that in addition to the traditional lattice write (i.e., the *-property or no write down rule), after having read C one can write P, which is a downgrade.

We next consider the set of Security Cards which are

$rs \in 2^L$	*Possible Writes*
$\{\}$	$\{P, C, S\}$
$\{P\}$	$\{P, C, S\}$
$\{C\}$	$\{P, C, S\}$
$\{S\}$	$\{S\}$
$\{P, C\}$	$\{P, C, S\}$
$\{P, S\}$	$\{S\}$
$\{C, S\}$	$\{S\}$
$\{P, C, S\}$	$\{S\}$

Figure 5. Possible writes given the set of reads (rs) performed so far.

generated by our factoring algorithm. There are 24 Security Cards generated from Figure 5, out of 32 considered (8 were not possible because the prerequisite mayflows were not defined). All 24 Security Cards are given in Appendix A; we describe here, some of the Security Cards generated.

The initial Security Card (shown in Figure 6) has no permissions, and hence it is safe for it to be used by anyone. There are no groups listed; since a user must be a member of *every* group listed, when there are no groups listed this holds (vacuously) for every user.

Name	**InitialCard**
Groups	
Privileges	
Security Method	$r\langle C\rangle$: switchTo(Read_C_Card) $r\langle P\rangle$: switchTo(Read_P_Card) $r\langle S\rangle$: switchTo(Read_S_Card) $w\langle P\rangle$: switchTo(Write_P_Card) $w\langle C\rangle$: switchTo(Write_C_Card) $w\langle S\rangle$: switchTo(Write_S_Card)

Figure 6. Initial Security Card

Next we consider the card `Read_CP_Card` shown in Figure 7. This card is reached only after having read objects with labels of C and P. There is only one more label which can be read, that is S. Each of the writes (to $P, C,$ or S) are also possible.

Finally, we consider the card with the maximal number of reads, see Figure 8. The only

Name	**Read_CP_Card**
Groups	$g_C \& g_P$
Privileges	$r\langle P\rangle, r\langle C\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Card) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Figure 7. Read_CP_Card

transition for `Read_CPS_Card` is a transition to `Read_CPS_Write_S_Card` which additionally enables a write to objects with label S.

Name	**Read_CPS_Card**
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle, r\langle S\rangle, r\langle P\rangle$
Security Method	$w\langle S\rangle$: switchTo(Read_CPS_Write_S_Card)

Name	**Read_CPS_Write_S_Card**
Groups	g_C & g_P & g_S
Privileges	$r\langle C\rangle, r\langle S\rangle, r\langle P\rangle, w\langle S\rangle$
Security Method	

Figure 8. Read_CPS_Card and Read_CPS_Write_S_Card

4. Optimizations

While the algorithm given in Section 3.2 is sufficient to generate the needed Security Cards, it generates 24 Security Cards for our small example. In this section, we describe some optimizations which significantly reduce the number of security cards generated.

The optimizations considered here are:

no writers: as the number of labels read increases the number of labels which can be written decreases. This optimization is applicable when there are no labels which can be written given the set of labels read.

lattice: this takes advantage of lattice (and partial lattice) structures to remove Security Cards with fewer read permissions in favor of those with more read permissions.

bottom: remove cards which do not have read permission on the most readable label in favor of those that do.

write augmentation: this optimization removes a Security Card which contains only read permissions in favor of one which contains the same read plus an additional write.

We note that all of these operations have enabling conditions which must be satisfied before they are applied.

4.1. No writers

Consider the case where the algorithm generates a Security Card with no write permissions and no write transitions. In this case it is impossible to do any more writes by the process.

For each read transitions (i.e. on permission $r\langle X\rangle$) in such a card is replaced with a transition to `SingletonReadXCard`, which contains the single permission $r\langle X\rangle$. The `SingletonReadXCard` has only read transitions and all transitions are to `SingletonReadXCard`.

While the no writes optimization does not play a role in our running example, it is important in large distributed systems. In such a case, the system would largely decompose, for example, into departments. Reads from two different departments would likely have no destination label in common that they could write to. Hence, the Security Card would not need to keep track of combinations of such labels, potentially dramatically reducing the number of Security Cards generated.

4.2. Lattice optimization

One very elegant property of lattices is that the combination of labels always results in a single label. Hence, in lattice based representations it is not necessary to represent subsets of labels read.

We hence consider the conditions under which this property can be exploited. Let $g \sqsubseteq g'$ meaning that in every state, each user in g is also in g'. Then the lattice property holds when both

$$r(x) \sqsubseteq r(y)$$

and

$$\forall_{z|mayflow(x,z)\downarrow} flow(x,z) \sqsubseteq flow(y,z)$$

where

$$flow(l,l') = r(l) \cap w(l') \cap mayflow(l,l') \quad .$$

The meaning of this property is that the permissions for y are a superset of the permissions for x and so that (1) if a process can read x it can read y and (2) whether y is actually read or not is immaterial for information flow since the information flow restraints having read x are stricter than those having read y.

If the lattice property holds, then a card whose permissions p include $r\langle x\rangle$ but not $r\langle y\rangle$ can be replaced with the card whose permissions are $p \cup r\langle y\rangle$.

Example:

Name	**Read_C_Card**
Groups	g_C
Privileges	$r\langle C\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_CS_Card) $w\langle P\rangle$: switchTo(Read_C_Write_P_Card) $w\langle C\rangle$: switchTo(Read_C_Write_C_Card) $w\langle S\rangle$: switchTo(Read_C_Write_S_Card)

Given the above security card, $r\langle P\rangle$ can be added because for x = C and y = P, the Lattice optimization requirements are met. This results in the elimination of the Read_C_Card, and all references to it being replaced with the Read_CP_Card, shown in Figure 7.

4.3. Bottom optimization

The bottom element in a lattice, $\perp$ can be added to every Security Card. Since the mayflows can describe non-lattice structures, this optimization is possible when for all labels $x \in L$:

$$r(x) \sqsubseteq r(\perp)$$

$$mayFlow(\perp, x)\downarrow$$

and

$$w(x) \sqsubseteq r(\perp) \cap mayflow(\perp, x) \cap w(x) \quad .$$

The second and third means that the mayflow restrictions added by having read $\perp$ do not limit what can be written.

If this condition holds, Security Cards with permissions p not containing $r\langle\perp\rangle$ are replaced by ones that have permissions $p \cup r\langle\perp\rangle$.

Example: Consider the initial security card shown in Figure 6. This card does not have any privileges, so it is a prime candidate for the bottom optimization. P is the most readable label, and since a $r\langle P\rangle$ does not limit what labels can be written, it satisfies the bottom optimization. References to the initial security card can be replaced with the Read_P_Card shown below.

Name	**Read_P_Card**
Groups	g_P
Privileges	$r\langle P\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_PS_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card) $w\langle S\rangle$: switchTo(Read_P_Write_S_Card)

4.4. Write augmentation optimization

The next optimization is called write augmentation, since it replaces a Security Card with one that has an additional write permission.

If a Security Card without any write privileges, s_0 has a transition to a Security Card s_1 with a write privilege and a user is a member of every groups on s_0 iff she is a member of every group on s_1, make all transitions to s_0 be instead transitions to s_1 and remove s_0.

Example: Consider the Read_CPS_Card shown in Figure 8 which contains only read privileges. Since a mayFlow for each C, P, and S exists to S, the security method transition to $w\langle S\rangle$ exists. Lastly, since the groups for Read_CPS_Write_S_Card, also shown in the figure, are the same as for the Read_CPS_Card, any user is allowed access to either both or neither card. Having met the requirements for the write augmentation optimization, any reference to Read_CPS_Card can be replaced with Read_CPS_Write_S_Card.

4.5. Applying optimization to the running example

Table 1 details each removed card along with the optimization and conditions which made it possible. For some cards, multiple optimizations can apply; however, only one will be listed. In the table, lattice(*l*,*l'*) means that the lattice properties hold for x = *l* and y = *l'*. Recall that for this example that mayFlow(*l*,*l*) = $r\langle$l$\rangle$ = $w\langle$l$\rangle$.

The optimizations result in removal of Security Cards and a corresponding change in transitions between Security Cards. After these optimizations, there remains 7 Security Cards, down from 24 before optimizations. The resulting Security Cards after optimizations are shown in Appendix B.

5. Related Work

The work presented here is part of a broader line of work by the authors in constructing high assurance authorization systems, which are sufficiently expressive and have lower complexity than alternative schemes. Because of the anonymous submission requirements, we will summarize our related work without giving detailed citations.

- The high level specification and administrative controls have been published in a major security conference, in which we both defined the mechanism and show that an algorithm exists to determine when the administrative approvals defined in Section 2.2 are needed.
- We have submitted for publication a proof that it is decidable what authorizations can be allowed within our system. As far as we know, this is the first proof of decidability for general purpose administrative controls.
- A detailed description of the kernel-level mechanism of *kernelSec* has been accepted in a security conference, which more fully describes *kernelSec* including mechanisms which support dynamic separation of duty, discretionary access controls and our group mechanism.
- We have submitted for publication a description of the *kernelSec* implementation to an Operating Systems conference.

We have implemented the *kernelSec* layer in the Linux kernel, primarily using LSM [26], and in a systems process called kernelSecD. The factoring has also been implemented.

Eliminated Card		Replaced by Card			
Reads	Writes	Reads	Writes	Optimization	Conditions satisfied
		P		bottom	bottom(P)
P		P	P	write augmentation	$r\langle P\rangle = w\langle P\rangle$ = mayFlow(P,P)
C		C,P		lattice	lattice(C,P)
S		S	S	write augmentation	$r\langle S\rangle = w\langle S\rangle$
C,P		C,P	C	write augmentation	$r\langle CP\rangle = w\langle C\rangle$ = mayFlow(P,C)
P,S		P,S	S	write augmentation	$r\langle PS\rangle = w\langle S\rangle$ = mayFlow(P,S)
C,S		C,S	S	write augmentation	$r\langle CS\rangle = w\langle S\rangle$ = mayFlow(C,S)
C,P,S		P,C,S	S	write augmentation	$r\langle PCS\rangle = w\langle S\rangle$ = mayFlow(P,S) = mayFlow(C,S)
	P	P	P	bottom	bottom(P)
	C	P	C	bottom	bottom(P)
	S	P	S	bottom	bottom(P)
C	P	C,P	P	lattice	lattice(C,P)
C	C	C,P	C	lattice	lattice(C,P)
C	S	C,P	S	lattice	lattice(C,P)
S	S	P,S	S	lattice	lattice(S,P)
P,S	S	P,C,S	S	lattice	lattice(S,C)
C,S	S	P,C,S	S	lattice	lattice(C,P)

Table 1. Optimization table

There are three types of related work discussed here. The first is the high-level specification of authorization, the second is the low-level specification of authorization, and finally the focus of this paper which is the translation between the high and low level.

Specification of permissions in terms of authorization properties is, of course, not new. Lattice based schemes do this in terms of a (rigid) hierarchy, in which information flow confidentiality or integrity can never be violated (within the system). However, one of the reasons for the growing popularity of Type Enforcement is that it is able to enforce these properties in part of the system and violate them in other parts, thus enabling specification of downgrade, etc.

Our high level specification is similar to RBAC in the sense that it is based on groups (or roles) and that it is compositional. However, our high-level specification differs in that it is designed to directly represent authorization properties. The notion of roles goes back to at least Landwehr, Heitmeyer, and McLean's elegant work on military message systems [17]—one of the issues they site is dynamically associating roles with users so that the users select their role—we note that comparable semantics (although not covered here) is to execute a program and thus (implicitly) select a role.

Jajodia, Samarati, and Subrahmanian [13] describe a logical language for expressing authorizations. Policies are expressed by rules which enforce derivation of authorizations, conflict rules, and integrity constraint checking. Their approach is very general; ours is specialized for authorization properties in that it embeds the properties and their analysis (decidability) in its construction. We believe this reduces the complexity of our specifications but this may be at the cost of some expressiveness.

The Ponder specification language [5] is for distributed systems, is object-oriented, and is broader in scope than our project. It is declarative, like our approach, but does not support administrative controls.

As we have noted, the *kernelSec* layer is most similar to Type Enforcement, which is a version of the access matrix. However, Security Cards have more dynamic features (e.g., groups and transitions on arbitrary missing permissions). But what drives the design of Security Cards is the goal of having them support authorization properties and hence the design of Security Cards is tuned to these needs.

Other authorization systems have allowed changes to the set of permissions based on the accesses the process makes. For example, CMW allows the label on an object to "float up" (within some range) rather than to deny a write due to information flow restrictions [2]; LOMAC enables the automatic downgrade of the process if an untrusted file is read [7].

We do not consider covert channels [16, 8, 9] here, because although they are necessary for MilSec systems

they do apply only to systems whose mayflow graphs are acyclic and they add a great deal of complexity to the specification of what is authorized.

Specifying permissions for access matrix level implementations is tedious. Least privilege [19] means that processes should get the minimum privileges allowed. An alternative to factoring used in SELinux [24] is to run the application in "permissive mode" to determine which permissions are used by an application; and then when run the application in production mode with the needed permissions. Analysis tools, such as [10] can verify information flow properties.

Our high level specification is written in a way that is closer to access control lists than capabilities. We believe that it is possible to adapt factoring to apply to capability based systems such as EROS [22]. For example, EROS currently supports lattice-based semantics through a combination of reference monitor plus sealing mechanism [23] (without some extra mechanism, this would be impossible [14]); by generalizing the reference monitor we believe that more authorization properties could be provided in the authorization system rather than relying on specially coded components in the OS.

Two earlier systems provided a high level specification which was translated into an implementation-level checker. These systems differ from our system significantly in the high-level specification and the low-level implementation. Neither of these systems supported administrative controls

The system which is closest to ours is Miró [11] which consists of a instance language to specify security configurations and a constraint language to specify security policies. Since Miró was mapped to existing access matrix level implementation, it neither could support the semantics described in this paper (e.g., information flow restrictions), nor could the low-level be modified to improve efficiency. Our system reuses components between layers, for example the group structure.

Hoagland, Pandey, and Levitt [12] describe a graphical language LaSCO which is then translated into Java code. This language cannot represent information flow constraints and of course its target is not an access matrix.

6. Conclusions

We have described a three tiered authorization system consisting of:

administrative controls to enable controlled changes to what is authorized;

high level specification for a succinct and stateless specification of the current authorizations; and

kernelSec our low level access controls implemented in the Linux Kernel.

The top two layers of our system have decidable information flow authorization properties, meaning that whether information flow confidentiality or information flow integrity holds can be determined algorithmically. Moreover in our systems, as in Type Enforcement, such properties can be guaranteed to hold for some objects and be violated for others. These guarantees are made at a label granularity. We believe that this level is significantly easier to specify and analyze than at the access matrix level (in our case, *kernelSec*).

We have also developed an access matrix level mechanism which is efficient called *kernelSec*. In addition to efficiency, its goal is to be sufficient for application-level authorization needs so that application level code for authorization is greatly reduced or eliminated. An additional goal is to be able automatically generate the *kernelSec* layer from the higher level specifications, and that too has effected the *kernelSec* design; we could not have generated existing access matrix level targets without a loss of flexibility.

In this paper we describe a factoring algorithm which takes our high level specification and produces a *kernelSec* configuration. The naive algorithm, while straightforward (it is given in the paper) produces far too many Security Cards. We describe a number of optimizations which dramatically reduce the number of Security Cards generated, from an initial 24 down to 7 for our running example. We have provided separately an information assurance argument which describes why the factoring is sound.

We believe that this approach provides the best of both worlds. A specification level which is optimized to concisely describe the authorization needs and an implementation level which is optimized for performance and guaranteed to provide the semantics needed by the higher level specification. To bridge the gap and remove error from the translation, algorithmic factoring ensure equivalence of the two representations. This method of specifying authorizations is analogous to the compilation high-level programs into machine language. This, we believe, will ultimately be the way all authorizations are specified.

References

[1] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, July 1976.

[2] 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.

[3] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.

[4] W. E. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *8th National Computer Security Conference*, pages 18–27, 1985.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[6] D. F. Ferraiolo and R. Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, 1992.

[7] T. Fraser. LOMAC: MAC you can live with. In *Freenix Track: 2001 USENIX Annual Technical Conference*, Boston, Mass., 2001.

[8] 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.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[10] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[11] A. Heydon and J. D. Tygar. Specifying and checking UNIX security constraints. *Computing Systems*, 7(1):91–112, 1994.

[12] J. Hoagland, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. *Technical report CSE-98-3, The University of California, Davis Department of Computer Science*, 1998.

[13] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 Conference on Security and Privacy (S&P-97)*, pages 31–43, Los Alamitos, May 4–7 1997. IEEE Press.

[14] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. IEEE Symp. Security and Privacy*, pages 2–12, 1984.

[15] B. Lampson. Protection. In *ACM Operating Systems Review*, volume 8, pages 18–24. ACM, 1974.

[16] B. W. Lampson. A note on the confinement problem. *Communications of the ACM (CACM)*, 16(10):613–615, 1973.

[17] C. Landwehr, C. Heitmeyer, and J. McLean. A security model for military message systems: Retrospective. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, page 174, Washington, DC, USA, 2001. IEEE Computer Society.

[18] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *Proc. IEEE Symp. Security and Privacy*, 2005.

[19] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[20] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[21] 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.

[22] 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.

[23] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. IEEE Symp. Security and Privacy*, pages 166–176, 2000.

[24] S. Smalley. Configuring the SELinux policy. Report #02-007, NAI Labs, Feb. 2002. Revised April 2002.

[25] 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.

[26] 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.

A. Non-Optimized Security Cards

Name	**InitialCard**
Groups	
Privileges	
Security Method	$r\langle C\rangle$: switchTo(Read_C_Card) $r\langle P\rangle$: switchTo(Read_P_Card) $r\langle S\rangle$: switchTo(Read_S_Card) $w\langle P\rangle$: switchTo(Write_P_Card) $w\langle C\rangle$: switchTo(Write_C_Card) $w\langle S\rangle$: switchTo(Write_S_Card)

Name	**Read_P_Card**
Groups	g_P
Privileges	$r\langle P\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_PS_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card) $w\langle S\rangle$: switchTo(Read_P_Write_S_Card)

Name	**Read_C_Card**
Groups	g_C
Privileges	$r\langle C\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_CS_Card) $w\langle P\rangle$: switchTo(Read_C_Write_P_Card) $w\langle C\rangle$: switchTo(Read_C_Write_C_Card) $w\langle S\rangle$: switchTo(Read_C_Write_S_Card)

Name	Read_S_Card
Groups	g_S
Privileges	$r\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_PS_Card) $r\langle C\rangle$: switchTo(Read_CS_Card) $w\langle S\rangle$: switchTo(Read_S_Write_S_Card)

Name	Read_CP_Card
Groups	$g_C \& g_P$
Privileges	$r\langle C\rangle, r\langle P\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Name	Read_PS_Card
Groups	$g_P \& g_S$
Privileges	$r\langle P\rangle, r\langle S\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CPS_Card) $w\langle S\rangle$: switchTo(Read_PS_Write_S_Card)

Name	Read_CS_Card
Groups	$g_C \& g_S$
Privileges	$r\langle C\rangle, r\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CPS_Card) $w\langle S\rangle$: switchTo(Read_CS_Write_S_Card)

Name	Read_CPS_Card
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle, r\langle P\rangle, r\langle S\rangle$
Security Method	$w\langle S\rangle$: switchTo(Read_CPS_Write_S_Card)

Name	Write_P_Card
Groups	g_P
Privileges	$w\langle P\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_P_Card) $r\langle C\rangle$: switchTo(Read_C_Card) $r\langle S\rangle$: switchTo(Read_S_Card) $w\langle C\rangle$: switchTo(Write_C_Card) $w\langle S\rangle$: switchTo(Write_S_Card)

Name	Write_C_Card
Groups	g_C
Privileges	$w\langle C\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_P_Card) $r\langle C\rangle$: switchTo(Read_C_Card) $r\langle S\rangle$: switchTo(Read_S_Card) $w\langle P\rangle$: switchTo(Write_P_Card) $w\langle S\rangle$: switchTo(Write_S_Card)

Name	Write_S_Card
Groups	g_S
Privileges	$w\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_P_Card) $r\langle C\rangle$: switchTo(Read_C_Card) $r\langle S\rangle$: switchTo(Read_S_Card) $w\langle P\rangle$: switchTo(Write_P_Card) $w\langle C\rangle$: switchTo(Write_C_Card)

Name	Read_P_Write_P_Card
Groups	g_P
Privileges	$r\langle P\rangle$, $w\langle P\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_PS_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card $w\langle S\rangle$: switchTo(Read_P_Write_S_Card

Name	Read_P_Write_C_Card
Groups	$g_P \& g_C$
Privileges	$r\langle P\rangle$, $w\langle C\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_PS_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle S\rangle$: switchTo(Read_P_Write_S_Card)

Name	Read_P_Write_S_Card
Groups	$g_P \& g_S$
Privileges	$r\langle P\rangle$, $w\langle S\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_PS_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card)

Name	Read_C_Write_P_Card
Groups	$g_P \& g_C \& g_D$
Privileges	$r\langle C\rangle$, $w\langle P\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_CS_Card) $w\langle C\rangle$: switchTo(Read_C_Write_C_Card) $w\langle S\rangle$: switchTo(Read_C_Write_S_Card)

Name	Read_C_Write_C_Card
Groups	g_C
Privileges	$r\langle C\rangle$, $w\langle C\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_CS_Card) $w\langle P\rangle$: switchTo(Read_C_Write_P_Card) $w\langle S\rangle$: switchTo(Read_C_Write_S_Card)

Name	Read_C_Write_S_Card
Groups	$g_C \& g_S$
Privileges	$r\langle C\rangle$, $w\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CP_Card) $r\langle S\rangle$: switchTo(Read_CS_Card) $w\langle P\rangle$: switchTo(Read_C_Write_P_Card) $w\langle C\rangle$: switchTo(Read_C_Write_C_Card)

Name	Read_S_Write_S_Card
Groups	g_S
Privileges	$r\langle S\rangle$, $w\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_PS_Card) $r\langle S\rangle$: switchTo(Read_CS_Card)

Name	Read_CP_Write_P_Card
Groups	$g_C \& g_P \& g_D$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle P\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Card) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Name	Read_CP_Write_C_Card
Groups	$g_C \& g_P$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle C\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Name	Read_CP_Write_S_Card
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle S\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card)

Name	Read_PS_Write_S_Card
Groups	$g_P \& g_S$
Privileges	$r\langle P\rangle$, $r\langle S\rangle$, $w\langle S\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CPS_Card)

Name	Read_CS_Write_S_Card
Groups	$g_C \& g_S$
Privileges	$r\langle C\rangle$, $r\langle S\rangle$, $w\langle S\rangle$
Security Method	$r\langle P\rangle$: switchTo(Read_CPS_Card)

Name	Read_CPS_Write_S_Card
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $r\langle S\rangle$, $w\langle S\rangle$
Security Method	

B. Optimized Security Cards

Name	Read_P_Write_P_Card
Groups	g_P
Privileges	$r\langle P\rangle$, $w\langle P\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $r\langle S\rangle$: switchTo(Read_CPS_Write_S_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card $w\langle S\rangle$: switchTo(Read_P_Write_S_Card

Name	Read_P_Write_C_Card
Groups	$g_P \& g_C$
Privileges	$r\langle P\rangle$, $w\langle C\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $r\langle S\rangle$: switchTo(Read_CPS_Write_S_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle S\rangle$: switchTo(Read_P_Write_S_Card)

Name	Read_P_Write_S_Card
Groups	$g_P \& g_S$
Privileges	$r\langle P\rangle$, $w\langle S\rangle$
Security Method	$r\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $r\langle S\rangle$: switchTo(Read_CPS_Write_S_Card) $w\langle P\rangle$: switchTo(Read_P_Write_P_Card) $w\langle C\rangle$: switchTo(Read_P_Write_C_Card)

Name	Read_CP_Write_P_Card
Groups	$g_C \& g_P \& g_D$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle P\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Write__SCard) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Name	Read_CP_Write_C_Card
Groups	$g_C \& g_P$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle C\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Write_S_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle S\rangle$: switchTo(Read_CP_Write_S_Card)

Name	Read_CP_Write_S_Card
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $w\langle S\rangle$
Security Method	$r\langle S\rangle$: switchTo(Read_CPS_Write_S_Card) $w\langle P\rangle$: switchTo(Read_CP_Write_P_Card) $w\langle C\rangle$: switchTo(Read_CP_Write_C_Card)

Name	**Read_CPS_Write_S_Card**
Groups	$g_C \& g_P \& g_S$
Privileges	$r\langle C\rangle$, $r\langle P\rangle$, $r\langle S\rangle$, $w\langle S\rangle$
Security Method	