The Specification and Implementation of 'Commercial' Security Requirements Including Dynamic Segregation of Duties

Simon N. Foley,
Department of Computer Science,
University College,
Cork, Ireland.
(s.foley@cs.ucc.ie)

Abstract

A framework for the specification of security policies is proposed. It can used to formally specify confidentiality and integrity policies, the latter can be given in terms of Clark-Wilson style access triples. The framework extends the Clark-Wilson model in that it can be used to specify dynamic segregation of duty.

For application systems where security is critical, a multilevel security based approach is defined. Security policies for less critical applications can be implemented using standard Unix based systems. Both implementation strategies are based on the standard protection mechanisms that are provided by the respective systems.

1 Introduction

Clark and Wilson [6] propose a model for (integrity) security that can be used for systems where security is enforced across both the operating system *and* the application systems. Their model is based on commercial data processing practices and can be used as a basis for evaluating the security of a complete application system. It's operating-system security requirements can be captured in terms of multilevel security (MLS), and can therefore be implemented and evaluated using 'existing technology' [14, 18]. However, [15] argues that, whereas the Clark-Wilson model considers static segregation of duty, it does not consider the formalization of dynamic segregation of duty.

In this paper, we describe a framework in which security policies, including dynamic segregation of duty, can be expressed. By expressing dynamic segregation of duty in terms of relabeling policies [11], it becomes possible to use the results in [14, 18] for implementation and evaluation of these policies. Our framework also provides a basis for policy refinement and composition [10]. These can be used in the development of complex policies which may include combinations of integrity and confidentiality requirements spread across different applications.

MLS systems are typically used when security is critical; a high degree of assurance is required that the security policy is upheld. For application systems that are less security critical, [16] outlines how they can be supported by a standard Unix system according to the Clark-Wilson model. We also show here how the approach in [16] can be adapted to support more general security policies.

To appear in the Proceedings of the 4th ACM Conference on Computer and Communications Security, April 1997, Zurich, Switzerland.

This paper is organized as follows: Section 2 considers how Clark-Wilson access triples may be expressed in terms of reflexive relations. This provides us with a basic framework for constructing complex policies which express both integrity and confidentiality policies [10]. Section 3 introduces a structure for specifying these policies, and considers their implementation in Unix and MLS systems. Section 3 may be regarded as a new application of the policy construction methods proposed in [10] to [14, 16, 18].

Section 4 considers how relabeling [11] can be adapted for our policy framework and also how these policies can, in turn, be supported by Unix and MLS systems. Using our framework, a dynamic segregation of duty policy and a Chinese Wall policy are formally specified, both of which can be enforced by Unix or MLS systems. Section 5 describes a technique that can reduce the number of security classifications and user-ids that are necessary for MLS and Unix system implementations.

The Z notation [19] is used to provide a consistent syntax for structuring and presenting the mathematics in this paper. In using Z, it has been possible to syntax- and type-check the definitions using the *f*UZZ tool. Appendix A.1 gives a brief overview of the Z notation.

2 The Clark-Wilson Model of Security

The Clark-Wilson (CW) model is defined in terms of *enforcement rules* and *certification rules*. Enforcement rules specify security requirements that should be supported by the protection mechanisms in the underlying operating system. The certification rules specify security requirements that the application system should uphold. There are nine rules in total, but we will consider only that rule concerned with supporting access control.

The model components include: the *Users* of the system; *Constrained Data Items* (CDIs) representing data objects with integrity, and *Transform Procedures* (TPs) that operate on CDIs and represent the well-formed transactions that provide the functionality of the application system.

2.1 Clark-Wilson Enforcement Rule E2

The main access-control requirement underlying the CW-model is that users may only access CDIs via TPs. And then only if that access is specified in an *E2rule* relation. For our purposes, an *E2rule* relation is a set of access-triples configured for a particular application system. An access-triple, given as (u, t, c), is interpreted to mean that the user u may use the TP t to access the CDI c. The set of all possible access triple relations is defined to be $\mathcal{AT}[C]$, where (generic) C represents the identifiers used for users, TPs and CDIs.

$$\begin{array}{l} \mathcal{AT}[C] == \\ \quad \{T : \mathbb{P}(C \times C \times C) \mid (\forall u, u', t, c, c' : C \bullet \\ \qquad ((u,t,c) \in T \wedge (u',t,c') \in T) \Rightarrow (u,t,c') \in T) \ \} \end{array}$$

We make an assumption that if a user u may invoke TP t, then that user may access (using t) *any* CDI that is accessible by TP t. This specification for access triple relations deviates slightly from the usual definition [6]. We use it because it leads to a simpler exposition of the results in this paper, but it in no way restricts it's application. By making additional instantiations, or copies, of TPs one can encode the access triples proposed in [6] as components of $\mathcal{AT}$.

Example 1 Under the Unix system, a user *smith* may modify the file of login passwords (*passwd*) only via a trusted function which we call *chpass*. This could be specified by the access triple relation:

$$PassTrips == \{(smith, chpass, passwd)\}$$

Similarly, in an inventory management system, the clerk *smith* may post only (incoming) invoices to the invoice file (CDI *invs*) using the TP *posti*. The clerk *jones* may file only (incoming) consignment notes to the consignments file using TP *postc*.

$$ClerkTrips == \{(smith, posti, invs), (jones, postc, cons)\}$$

This is an example of static segregation of duty. △

2.2 Access Triples and Reflexive Relations

An access triple (u, t, c) may be viewed in terms of a non-transitive ordering: u may access t and t may access c, but u may not (directly) access c. In this section we describe how relations from $\mathcal{AT}$ may be expressed as reflexive (binary) ordering relations. There are a number of advantages to taking this approach. In particular, reflexive relations become convenient abstractions of existing security policies [10]. We can then compose and refine these policies and also systematically construct complex policies that express both confidentiality and integrity requirements. Appendix A.2 defines the operators used in the construction of reflexive relations; the reader is referred to [10] for more details.

Reflexive relations are used to specify information flow policies. These policies define the different classes of information that can exist in a system and whether or not information may flow between these classes. In [10] we suggest that, in addition to considering the usual sensitivity levels such as **secret** and **topsecret**, we should also consider unique classes to represent significant system components, such as users, objects, programs and database components. If we do this, classes can be used to represent TPs and CDIs. This approach is also suggested in [5]. The set of all reflexive relations between classes of (generic) type C is defined by $\mathcal{R}[C]$, where

$$\mathcal{R}[C] == \{R : C \leftrightarrow C \mid \mathrm{id}(\mathrm{dom}\, R \cup \mathrm{ran}\, R) \subseteq R\}$$

If $R \in \mathcal{R}[C]$ and $a \mapsto b \in R$, then we say that a is less than, or equal to, b in R. If the notation $A \rightsquigarrow B$ defines a reflexive relation where all elements of A are less than all elements of B, then a simple multilevel-style policy can be specified as

$$MLS == \{\mathbf{unclass}, \mathbf{secret}\} \rightsquigarrow \{\mathbf{secret}, \mathbf{topsecret}\}$$

The alphabet of a reflexive relation defines the components of that relation. For example, we have

$$\alpha MLS = \{\mathbf{unclass}, \mathbf{secret}, \mathbf{topsecret}\}$$

The set $\mathcal{R}[C]$ forms a lattice under a partial ordering $\sqsubseteq$, and lowest upper bound operator $\sqcap$. Intuitively, $R \sqsubseteq Q$ means that Q is no less restrictive than R, that is, any flow that is not allowed by R will also not be allowed by Q. We view an $R \sqsubseteq Q$ relation as a refinement relation in the sense of [13]: the policy defined by Q is, in a security sense, an acceptable replacement for the policy R. Therefore, a system that is secure by policy Q is also secure by policy R. Since $R \sqcap Q$ is a lowest upper bound on R and Q, then it is, in a security sense, an acceptable replacement for R and Q.

Example 2 A reflexive relation specification for the simple password policy is:

$$\begin{array}{l} PassReln == \perp\{smith, chpass, passwd\} \\ \qquad\qquad\qquad \sqcap \mathrm{not}(\{smith\} \rightsquigarrow \{passwd\}) \end{array}$$

where $\perp A$ gives the least restrictive policy with alphabet A. *PassReln* specifies that any flow is permitted, except from *smith* to *passwd*. This implies that *smith* may not (directly) modify *passwd*. Note that information is permitted to flow from *passwd* to *smith*. △

$$\begin{array}{|l} [C] \\ \hline usr, tp, cdi : (C \times C \times C) \to C \\ \mathrm{unzip} : \mathcal{AT}[C] \to \mathcal{R}[C] \\ \hline usr(u,t,c) = u \wedge tp(u,t,c) = t \wedge cdi(u,t,c) = c \\ \mathrm{unzip}\ \ T = \bigcup\{\ t : T \bullet \perp\{usr(t), tp(t), cdi(t)\}\ \} \\ \qquad\qquad \sqcap \mathrm{not}((usr(\!|T|\!) \setminus tp(\!|T|\!)) \rightsquigarrow (cdi(\!|T|\!) \setminus tp(\!|T|\!))) \end{array}$$

Given an access-triple relation T, unzip(T) returns its equivalent reflexive relation. The policy $\mathrm{not}((usr(\!|T|\!) \setminus tp(\!|T|\!)) \rightsquigarrow (cdi(\!|T|\!) \setminus tp(\!|T|\!)))$ specifies that information may not directly flow from (a class representing) a user to a CDI; however, for generality, the flow may be permitted if the user or CDI also corresponds to a TP. The policy $\bigcup\{\ t : T \bullet \perp\{usr(t), tp(t), cdi(t)\}\ \}$ specifies that for each triple t, information may flow between all the triple's components. The policy unzip(T) is the least restrictive policy that enforces the flow restrictions of *both* these component policies. In this policy we permit direct flows from CDIs to users, however users must use TPs when modifying CDIs. If required, it is a straightforward process to compose this policy with a further policy specifying that information may not flow directly from CDIs to users.

Example 3 From Examples 1 and 2, we have $PassReln = \mathrm{unzip}(PassTrips)$. The more restrictive policy $(PassReln \sqcap \mathrm{not}(\{passwd\} \rightsquigarrow \{smith\}))$ additionally specifies that information may not flow from *passwd* to *smith*. The policy $(\mathrm{unzip}(ClerkTrips) \sqcap MLS)$ is an example of a combined integrity $(ClerkTrips)$ and confidentiality (MLS) policy. △

3 Developing Security Policies

Let *ID* represent the set of all possible users, TPs and CDIs. A security policy is defined in terms of a particular set of these entities *ENTS*.

Ents

$$\begin{array}{|l}
ENTS, USR, TP, CDI : \mathbb{P}\, ID \\ \hline
\langle USR, TP, CDI \rangle \text{ partition } ENTS
\end{array}$$

Given *Ents*, then a set of classes of generic type C is defined which is used to represent the different classes of information expected in the system. This may include classes representing users, CDIs, sensitivity levels, and so forth. A security policy is specified in terms of

Policy[C]

$$\begin{array}{|l}
Ents \\
\beta : ID \rightarrowtail C \\
R : \mathcal{R}[C] \\ \hline
\operatorname{dom} \beta = ENTS \\
\operatorname{ran} \beta \subseteq \alpha R
\end{array}$$

R gives the reflexive flow relation and β associates each entity from ENT with some class from the alphabet of R.

Example 4 For the password policy, entities (identified using an *italic* font) *smith*, *chpass* and *passwd* are represented by `smth`, `chps` and `pswd`, (elements of basic type $CLASS$, identified using a `typewriter` font), respectively.

PassPolicy

$$\begin{array}{|l}
Policy[CLASS] \\ \hline
USR = \{smith\} \wedge TP = \{chpass\} \wedge CDI = \{passwd\} \\
R = \operatorname{unzip}\{(\mathtt{smth}, \mathtt{chps}, \mathtt{pswd})\} \\
\beta = \{smith \mapsto \mathtt{smth}, chpass \mapsto \mathtt{chps}, passwd \mapsto \mathtt{pswd}\}
\end{array}$$

$\triangle$

3.1 Multilevel Implementation

Shockley [18] and Lee [14] describe how MLS systems such as [1, 3] can be configured to support the CW-model, and in particular, how the access-triple relation can be encoded in terms of a lattice policy together with bindings for objects and partially trusted subjects. In this section we characterize these MLS policies and show how they can be computed from a *Policy* specification. The reader is referred to [14, 18] for specific implementation details of how to apply the policies described in this section.

Given basic type $CLASS$ representing the set of security classes, then we specify a multilevel policy based on the powerset lattice of $\mathbb{P}\, CLASS$ as

MLSPolicy

$$\begin{array}{|l}
Policy_1[\mathbb{P}\, CLASS \times \mathbb{P}\, CLASS] \\ \hline
(A, B) \in \alpha R_1 \Rightarrow A \subseteq B \\
((A, B), (C, D)) \in R_1 \Leftrightarrow A \subseteq C \wedge B \subseteq D
\end{array}$$

This specification, while somewhat stylized, is interpreted as follows. Variables are decorated by the subscript 1 to signify components of an implementation policy specification. R_1 is a lattice of pairs (of sets of classes). These pairs can be viewed as defining intervals on the powerset lattice based on $\mathbb{P}\, \alpha R_1$. This powerset lattice corresponds to the implementation lattice policy that forms part of an MLS system.

Each user u is bound to a pair $\beta_1(u) = (A, B)$ from αR_1. The user is considered cleared to classes between A and B in the powerset lattice $\mathbb{P}\, \alpha R_1$. Each TP t runs as a partially trusted subject, with $\beta_1(t)$ specifying the interval of trust for its alter-minimum and view-maximum $(amin, vmax)$ bindings. A CDI c is viewed as a single-level object in which $\beta_1(c) = (A, B)$ implies $A = B$. This could be, for example, a single-level file. The CDI is a multilevel object if $A \subset B$. In this case it might correspond, for example, to a multilevel database table with table classification constraints $\beta_1(c)$.

To minimize the size of the implementation powerset lattice, a smaller sublattice L_1 of the powerset lattice can be used. It is constructed by taking the partial order (subset) defined in terms of (implementation) classes $first(\!| \alpha R_1 |\!) \cup second(\!| \alpha R_1 |\!)$, and adding additional classes until a lattice is formed [7]. Instead of reproducing the algorithm in [7], we specify

MLSPolicyImp

$$\begin{array}{|l}
MLSPolicy \\
L_1 : \mathcal{R}[\mathbb{P}\, CLASS] \\ \hline
L_1 \subseteq \{\, A, B : \mathbb{P}\, \alpha R \mid A \subseteq B \,\} \\
L_1 \supseteq \{\, A, B : first(\!| \alpha R_1 |\!) \cup second(\!| \alpha R_1 |\!) \mid A \subseteq B \,\}
\end{array}$$

Informally, an MLS system is secure according to such a policy containing entities x, y, if information flows from x to y then $first(\beta_1(x)) \subseteq second(\beta_1(y))$ holds. This corresponds to the usual Simple Security Condition and Star Property, as given in [3].

Example 5 The Password policy can be specified as an *MLSPolicy* based on the implementation lattice L_1 and β_1 bindings given in Figure 1. Note that we use `s` to abbreviate `smth`, and so forth. Using these bindings with the implementation strategy in [14], it follows that *smith* may not modify *passwd*, except via *chpass*. $\triangle$

x	$\beta_1(x)$
smith	$[\{\mathtt{s},\mathtt{c},\mathtt{p}\},\{\mathtt{s},\mathtt{c},\mathtt{p}\}]$
chpass	$[\{\mathtt{c},\mathtt{p}\},\{\mathtt{s},\mathtt{c},\mathtt{p}\}]$
pass	$[\{\mathtt{c},\mathtt{p}\},\{\mathtt{c},\mathtt{p}\}]$

$$\begin{tikzcd} \{\mathtt{s},\mathtt{c},\mathtt{p}\} \\ \{\mathtt{c},\mathtt{p}\} \arrow[u] \end{tikzcd}$$

Figure 1: MLS Implementation of *PassPolicy*

It is tedious to manually construct such lattice based Clark-Wilson policies. In [10] we give a function Φ that maps an arbitrary reflexive relation into a lattice plus intervals on that lattice. We use this function, defined in Appendix A.2, to transform a (reflexive) specification *Policy* onto its lattice implementation *MLSPolicy*.

MLSPolicyΦ

$$\begin{array}{|l}
Policy[CLASS] \\
MLSPolicy \\ \hline
\theta Ents_1 = \theta Ents \\
\beta_1 = \beta \,{}_9^\circ\, (\Phi_\iota\; R) \\
R_1 = \Phi\; R
\end{array}$$

It follows, from the order-preserving property of Φ [10], that

$$\begin{array}{l}
MLSPolicy\Phi \vdash \\
\quad \forall\, x, y : ENTS \bullet \\
\qquad first(\beta_1(x)) \subseteq second(\beta_1(y)) \Leftrightarrow (\beta(x), \beta(y)) \in R
\end{array}$$

For example, applying the transformation to specification *PassPolicy* gives an implementation which is equivalent to that defined in Figure 1. Similarly, we can compute an MLS implementation for *ClerkTrips*, but for reasons of space we cannot include the details here.

3.2 Unix Implementation

A Unix system can be configured to support the CW-model [16, 20] by implementing the access-triple relation as an encoding of user-groups, and with TPs as set-user-id (SUID) programs. In this section we specify these Unix policies and show how they can be computed from a *Policy* specification. The reader is referred to [16] for more specific implementation details.

Since standard Unix cannot enforce information flow controls, we interpret an information flow policy $\{a\} \leadsto \{b\}$ as meaning that user a may have *access* to files owned by user b. The essence of our approach is that user-id b will have a group-id gb with a being a member of gb. Let UID and GID represent the set of all possible user-ids and group-ids, respectively. A Unix user-group policy is specified as

$$
\begin{array}{|l}
\textit{UnixPolicy} \\
\hline
\textit{Policy}_1[\textit{UID}] \\
grp_1 : \textit{UID} \rightarrowtail\!\!\!\!\!\!\!+\;\; \textit{GID} \\
mbr_1 : \textit{UID} \leftrightarrow \textit{GID} \\
\hline
\mathrm{ran}\ mbr_1 = \mathrm{ran}\ grp_1 \\
\mathrm{dom}\ mbr_1 = \mathrm{dom}\ grp_1 = \mathrm{ran}\,\beta_1 \\
mbr_1 = R_1 \mathbin{;} grp_1 \\
\hline
\end{array}
$$

For our purposes, the injective function grp_1 associates a unique GID with each UID and relation mbr_1 specifies group membership (as implemented by file `/usr/group`). Note that grp_1 and mbr_1 are simply an alternative representation for R_1.

Each user u has an associated UID given by $\beta_1(u)$. A CDI c is implemented as a file, owned by $\beta_1(c)$ and with group-id $grp_1(\beta_1(c))$. Its access permission bits should be set to only `RW` for owner and group. TP t is an executable program, stored in a file, with UID $\beta_1(t)$, GID $grp_1(\beta_1(t))$, and has permission bits set to execute by user and group only, and has its SUID bit set. CDI and TP UIDs typically correspond to phantom user-ids—that is they are not login-user-ids and have no associated human user.

Example 6 The password policy can be specified in terms of *UnixPolicy* with UIDs **smth**, **chps**, and **pswd**, and corresponding GIDs **gsmth**, **gchps**, and **gpswd**, respectively. Files *chpass* and *passwd* have the self-explanatory protection profiles:

	user	group	other
chpass	`rws`	`rx-`	`---`
	chps	**gchps**	
passwd	`rw-`	`rw-`	`---`
	pswd	**gpswd**	

The $mbrs_1$ relation is configured as follows. Smith is permitted to access (execute) the TP *chpass* ($\mathbf{smth} \in grp_1(\mathbf{gchps})$), which in turn may access file *passwd* ($\mathbf{chps} \in grp_1(\mathbf{gpswd})$). But Smith may not directly access the password file ($\mathbf{smth} \notin grp_1(\mathbf{gpswd})$). △

Given a policy specified in terms of $\textit{Policy}_1[\textit{UID}]$, then allocating grp_1 and calculating mbr_1 according to *UnixPolicy* gives its Unix implementation. Since $mbr_1 = R_1 \mathbin{;} grp_1$, it follows that the Unix implementation preserves the access constraints of the policy, that is,

$$
\begin{array}{l}
\textit{UnixPolicy} \vdash \\
\quad \forall\, x, y : \textit{ENTS} \bullet \\
\qquad (\beta_1(x), grp_1(\beta_1(y))) \in mbr_1 \Leftrightarrow (\beta_1(x), \beta_1(y)) \in R_1
\end{array}
$$

For example, $\textit{PassPolicy}_1 \wedge \textit{UnixPolicy}$ specifies a Unix implementation policy that is similar to that in Example 6.

Note that individual TP's are SUID programs and need to be checked for vulnerabilities to SUID attack [12]. However, they are not owned by `root`, and a compromise may be considered less critical than compromise of SUID `root` programs. This checking should form part of the certification performed on TPs during the CW-model evaluation of the application.

4 Dynamic Security Policies

The CW-model, while considering static segregation of duty like that described in *ClerkPolicy*, does not consider the formalization of dynamic segregation of duty [15]. For example, we cannot use a standard access-triple relation to express the requirement that: a clerk may either process an invoice, or verify a consignment, but not both. In this section we show how our *Policy* framework can be extended to support such a requirement.

Relabel policies [11] are lattice-based MLS policies that are augmented by a collection of *relabel functions*. These functions define how subject and object security class labels may change and can be used to encode dynamic aspects of MLS requirements.

Since reflexive flow policies are simply abstractions of lattice based policies, we argue that relabel functions can be specified in terms of reflexive relations and these in turn can be mapped to an implementation lattice. Let FID define the set of identifiers that represent relabel functions. A basic relabel policy is then specified in terms of *RFuns*:

$$
\begin{array}{|l}
\textit{RFuns}[C] \\
\hline
R : \mathcal{R}[C] \\
F : \textit{FID} \mathrel{+\!\!\!\rightarrow} C \mathrel{+\!\!\!\rightarrow} C \mathrel{+\!\!\!\rightarrow} C \\
\hline
\forall\, f : \mathrm{ran}\, F \bullet \\
\quad \mathrm{dom}\, f \cup \bigcup(\mathrm{dom}(\!|\mathrm{ran}\, f|\!)) \cup \bigcup(\mathrm{ran}(\!|\mathrm{ran}\, f|\!)) \subseteq \alpha R \\
\hline
\end{array}
$$

R gives the flow policy and F is a set of relabel functions, where given $\textit{fid} \in \mathrm{dom}\, F$, then $F(\textit{fid})$ defines the relabel function identified as $\textit{fid}$. Given classes $a \in \mathrm{dom}(F(\textit{fid}))$ and $b \in \mathrm{dom}(F(\textit{fid})(a))$, then a user at class a may request to relabel an entity at class b by class $F(\textit{fid})(a)(b)$. For example, a simple relabeling function that reclassifies an entity's multilevel classification to that of the requester is $(\lambda\, s : \alpha\textit{MLS} \bullet (\lambda\, a : \alpha\textit{MLS} \bullet s))$. We say a relabel policy is a *Policy* with relabel functions.

$$\textit{RPolicy}[C] \mathrel{\widehat{=}} \textit{Policy}[C] \wedge \textit{RFuns}[C]$$

Example 7 Clerk *smith* may post either invoices or consignment notes, but not both.

$$
\begin{array}{|l}
\textit{SimpleDynamic} \\
\hline
\textit{RPolicy}[\textit{CLASS}] \\
opt : \{\mathbf{posti}, \mathbf{post}\} \rightarrowtail\!\!\!\!\!\!\!+\;\; \textit{FID} \\
\hline
\textit{USR} = \{\textit{smith}\} \wedge \textit{TP} = \{\textit{post}\} \wedge \textit{CDI} = \{\textit{invs}, \textit{cons}\} \\
R = \mathrm{unzip}\{(\mathbf{smth}, \mathbf{post}, \mathbf{post}), (\mathbf{smth}, \mathbf{posti}, \mathbf{invs}), \\
\qquad\qquad (\mathbf{smth}, \mathbf{postc}, \mathbf{cons})\} \\
\beta = \{\textit{smith} \mapsto \mathbf{smth}, \textit{post} \mapsto \mathbf{post}, \\
\qquad\qquad \textit{invs} \mapsto \mathbf{invs}, \textit{cons} \mapsto \mathbf{cons}\} \\
F = (\lambda\, f : \mathrm{ran}\ opt \bullet \\
\qquad\qquad (\lambda\, s : \{\mathbf{smth}\} \bullet (\lambda\, t : \{\mathbf{post}\} \bullet opt^{\sim} f))) \\
\hline
\end{array}
$$

Function opt is used to allocate unique FIDs for the two relabel functions defined in F. The relabel function identified as $opt(\mathbf{posti})$, relabels class **post** by class **posti**, while the function $F(opt(\mathbf{postc}))$ relabels class **post** by class **postc**. Only the user **smth** may use these functions. Note that from the access-triples: *smith* can potentially access all CDI classes. But there is only one TP, identified by *post*, and initially it may not access any CDI. Only by using the relabel functions in F can *smith* change the label for TP *post* and subsequently gain access to a CDI. $\triangle$

The previous example shows that relabel functions can be used to encode dynamic segregation of duties. The definition of relabeling [11] can be adapted to the policy framework proposed here as follows:

$$InitRPolicy[C] \mathrel{\widehat{=}} RPolicy[C]$$

Initially, a relabel policy may have any legal configuration. Transition *Relabel* is atomic and specifies the effect that a relabel function has on a relabel policy.

$$
\begin{array}{|l}
\textit{Relabel}[C] \\
\hline
\Delta RPolicy[C] \\
req? : ID \\
target? : C \\
f? : FID \\
\hline
req? \in USR \wedge f? \in \mathrm{dom}\ F \\
\beta\ req? \in \mathrm{dom}(F\ f?) \\
target? \in \mathrm{dom}(F\ f?\ (\beta\ req?)) \\
\theta Ents = \theta Ents' \wedge \theta RFuns = \theta RFuns' \\
\beta' = \beta \oplus \{\ x : ID \mid \beta\ x = target? \\
\qquad\qquad\qquad \bullet (x \mapsto (F\ f?\ (\beta\ req?)\ target?))\ \} \\
\hline
\end{array}
$$

A relabel request is made by a user *req?* to relabel *all* entities that are bound to class *target?* using the relabel function *fid?*. It changes only the β component of the relabel policy. Note that the granularity of the policy is important here; if the policy specifier wishes to apply the function to just one specific entity (rather than all entities of a particular class), then he must identify that entity by a special class. The motivation for taking this particular approach will become apparent in Section 5.

Example 8 The dynamic segregation of duty policy can be generalized to any number of clerks. Clerks may post either incoming consignment notes, incoming invoices, or payments, to their respective data stores (files).

Let *NAME* represent the set of all possible names for clerks. The free type, *TKIND*, defines the different types of transactions in the system, *ID* represents the entities in the system, and *CL* defines the classes for these entities.

$$
\begin{array}{lll}
TKIND & ::= & \mathbf{invoice} \mid \mathbf{cnote} \mid \mathbf{payment} \mid \mathbf{none} \\
ID & ::= & clerk\langle\!\langle NAME \rangle\!\rangle \mid post \mid file\langle\!\langle TKIND \rangle\!\rangle \\
CL & ::= & \mathbf{clerk}\langle\!\langle NAME \times TKIND \rangle\!\rangle \\
 & \mid & \mathbf{post}\langle\!\langle TKIND \rangle\!\rangle \mid \mathbf{file}\langle\!\langle TKIND \rangle\!\rangle
\end{array}
$$

Entity $clerk(n)$ is the entity with name n; *post* is the post TP entity, and $file(k)$ is the CDI file containing kind k transactions. Given the construction $clerk(n)$, we can extract the name of the original clerk using the destructor function (inverse) $clerk^{\sim}$, that is $clerk^{\sim}(clerk(n)) = n$, and similarly for the other constructors in types *ID* and *CL*. Class $\mathbf{clerk}(n,k)$ represents the clerk with name n who has posted kind k transactions (initially **none**). This class is used to effectively encode a history of what the clerk has done. Class $\mathbf{post}(k)$ represents a TP posting kind k transactions, and $\mathbf{file}(k)$ represents CDI $file(k)$. The policy is specified as:

$$
\begin{array}{|l}
\textit{Segregation} \\
\hline
RPolicy[CLASS] \\
opt : TKIND \rightarrowtail FID \\
\hline
USR = clerk\llparenthesis NAME \rrparenthesis \wedge TP = \{post\} \\
CDI = file\llparenthesis TKIND \rrparenthesis \\
\beta = \{\ n : NAME \bullet (clerk(n) \mapsto \mathbf{clerk}(n, \mathbf{none}))\ \} \\
\quad \cup \{(post \mapsto \mathbf{post}(\mathbf{none}))\} \cup file^{\sim} \mathbin{;} \mathbf{file} \\
R = \mathrm{unzip}\{\ n : NAME;\ k : TKIND\ \bullet \\
\qquad\qquad (\mathbf{clerk}(n,k), \mathbf{post}(k), \mathbf{file}(k))\ \} \\
F = (\lambda\, fid : opt\llparenthesis TKIND \rrparenthesis\ \bullet \\
\quad (\lambda\, req : \mathbf{clerk}\llparenthesis NAME \times TKIND \rrparenthesis\ \bullet \\
\qquad (\lambda\, target : \mathbf{post}\llparenthesis TKIND \rrparenthesis \bullet \mathbf{post}(opt^{\sim}(fid))) \\
\qquad \oplus \\
\qquad (\lambda\, target : \mathbf{clerk}\llparenthesis (NAME \times \{\mathbf{none}\}) \rrparenthesis\ \bullet \\
\qquad\quad \mathbf{clerk}(first(\mathbf{clerk}^{\sim}(target)), opt^{\sim}(fid))))) \\
\hline
\end{array}
$$

Access-triples of the form $(\mathbf{clerk}(n,k), \mathbf{post}(k), \mathbf{file}(k))$ reflect the fact that a clerk (who has posted kind k transactions) may use a TP with class $\mathbf{post}(k)$ to access a k transaction file. Initially, each clerk is bound to $\mathbf{clerk}(n, \mathbf{none})$, and must use the relabel functions to *opt* for posting a particular type of transaction. This is done in two stages. First, the clerk must request to change his own label from $\mathbf{clerk}(n, \mathbf{none})$ to $\mathbf{clerk}(n,k)$, where k indicates transaction kind. Relabeling $(F\ opt(k)\ \mathbf{clerk}(n, \mathbf{none})\ \mathbf{clerk}(n, \mathbf{none}))$ achieves this. Then, if necessary, the clerk requests to change the post TP binding to $\mathbf{post}(k)$, so that he may access the CDI $\mathbf{file}(k)$. Note that once opted for posting a particular kind of transaction, the relabel functions will not permit a subsequent request by the clerk to post the other kinds of transactions. $\triangle$

There are alternative ways to specify dynamic segregation of duty. In the previous example, we could declare multiple copies of the *post* TP, one for each transaction kind (entity $post(k)$ with class $\mathbf{post}(k)$). Under this scheme it is not necessary for clerks to request relabeling of the *post* TP.

Policy *Segregation* works by relabeling classification labels. Its MLS interpretation corresponds to a modification of user clearances. However, its Unix interpretation would appear, at least initially, to correspond to changing the User's UID, which may not be practical or desirable. Section 5 will consider how *Segregation* can be implemented without having to change entity bindings. Another approach is to re-specify *Segregation* such that it defines a separate copy of the post TP for each clerk, that is, a $post(n)$ TP for each $clerk(n)$. Initially, each TP $post(n)$ has classification $\mathbf{post}(n, \mathbf{none})$, and $clerk(n)$ requests a relabel (to this TP) to $\mathbf{post}(n,k)$ for kind k transactions. Access triples are of the form $(\mathbf{clerk}(n), \mathbf{post}(n,k), \mathbf{file}(k))$. Under this scheme a clerk's classification does not change. This strategy is illustrated in the next example.

Example 9 The Chinese Wall policy [4] can be regarded as a confidentiality dual of a dynamic segregation of duty policy: a stock market analyst may not advise an organization if he has insider knowledge of another competing organization. This policy can be implemented in terms of a MLS lattice-based relabel policy [11], and using Unix user-groups [8]. We encode it here in terms of a more abstract *RPolicy* specification.

Let types *NAME* and *ORG* represent the set of all market analysts and organizations, respectively. Conflict of interest is defined in terms of relation $(\_ \ddagger \_)$, where for $a, b \in ORG$, $a \ddagger b$ means that a is in competition with b.

The Chinese Wall policy was originally defined in terms of analysts (*USR*) and organization datasets (*CDI*). A policy could be constructed such that each organization o has one CDI $dataset(o)$, bound to a class $\mathbf{dset}(o)$. Each analyst $analyst(n)$ is initially bound to a class $\mathbf{anl}(n, \varnothing)$, indicating that he not yet accessed any datasets. The relabel functions ensure that an analyst's binding may be changed only if it does not result in a conflict of interest. This is the essence of the approaches in [9, 11, 17].

However, a potential problem arises when implementing this policy directly in Unix. The Unix interpretation implies that an analyst cannot gain *access* to datasets owned by competing organizations; it does not constrain the *propagation* of organization data. An analyst can copy organization data into a public area. This could be done inadvertently, deliberately or by a Trojan Horse. In [8] this attack is limited by requiring that analysts access datasets only via TPs. The informal approach in [8] can be formalized in terms of a more general *Relabel* policy specification as follows.

$$ID ::= dataset\langle\!\langle ORG \rangle\!\rangle \mid analyst\langle\!\langle NAME \rangle\!\rangle \mid advise\langle\!\langle NAME \rangle\!\rangle$$

$$\begin{array}{ll} CL ::= & \mathbf{dset}\langle\!\langle ORG \rangle\!\rangle \mid \mathbf{anl}\langle\!\langle NAME \rangle\!\rangle \\ \mid & \mathbf{advs}\langle\!\langle NAME \times \mathbb{F}\ ORG \rangle\!\rangle \end{array}$$

Each $analyst(n)$ uses his own copy of the TP $advise(n)$ to access datasets. These could be actual copies of the TP or *TP wrappers* (as described in [16]). A TP's class $\mathbf{advs}(n, O)$ indicates the organizations that $analyst(n)$ is advising. Thus, a request to access the dataset of an organization o involves a relabeling (using function $opt(o)$) of just the requester's *advise* TP.

$$\begin{array}{|l} \textit{ChineseWall} \\ \hline RPolicy[CL] \\ \_ \ddagger \_ : ORG \leftrightarrow ORG \\ opt : ORG \rightarrowtail\!\!\!\!\!\!+\ FID \\ \hline USR = analyst(\!| NAME |\!) \wedge TP = advise(\!| NAME |\!) \\ CDI = dataset(\!| ORG |\!) \\ R = \mathrm{unzip}\{\ o : ORG;\ O : \mathbb{F}\ ORG;\ n : NAME \\ \qquad\qquad \bullet (\mathbf{anl}(n), \mathbf{advs}(n, O \cup \{o\}), \mathbf{dset}(o))\ \} \\ \qquad \sqcap \mathrm{not}(\mathbf{dset}(\!| ORG |\!) \rightsquigarrow \mathbf{anl}(\!| NAME |\!)) \\ \beta = analyst^{\sim} \mathbin{;} \mathbf{anl} \cup dataset^{\sim} \mathbin{;} \mathbf{dset} \\ \qquad \cup \{\ n : NAME \bullet advise(n) \mapsto \mathbf{advs}(n, \varnothing)\ \} \\ F = (\lambda\, fid : opt(\!| ORG |\!) \bullet \\ \qquad (\lambda\, req : \mathbf{anl}(\!| NAME |\!) \bullet \\ \qquad\quad (\lambda\, tp : \mathbf{advs}(\!| \{\mathbf{anl}^{\sim}(req)\} \times \mathbb{F}\ ORG |\!) \\ \qquad\qquad \mid (\forall\, a : second(\mathbf{advs}^{\sim}(tp)) \bullet \neg\ (a \ddagger opt^{\sim}(fid))) \\ \qquad\qquad \bullet \mathbf{advs}(\mathbf{anl}^{\sim}(req), \\ \qquad\qquad\qquad second(\mathbf{advs}^{\sim}(tp)) \cup \{opt^{\sim}(fid)\})))) \\ \hline \end{array}$$

This is an example of a combined confidentiality and integrity (including dynamic segregation of duty) policy. This policy can be enforced by MLS systems and, unlike [9, 11, 17], by standard Unix systems. As a general observation, it has been our experience that the specification of policies, such as *ChineseWall*, in terms of reflexive flow relations, are easier to develop (and comprehend) than lattice-based constructions such as [11, 17]. $\triangle$

4.1 Multilevel Implementation

It is straightforward to generalize the MLS implementation described in Section 3.1 to relabel policies. The same transformation $MLSPolicy\Phi$ is used to transform a *Policy* specification into a *MLSPolicy* implementation, and thus the orderings of the reflexive policy are preserved in its lattice implementation. The relabel functions can be transformed by the Φ mapping according to

$$\begin{array}{|l} \textit{MLSRFun}\Phi \\ \hline RFuns[CLASS] \\ RFuns_1[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS] \\ \hline R_1 = \Phi\ R \\ F_1 = (\lambda\, f : \mathrm{dom}\ F \bullet \\ \qquad (\Phi_\iota\ R)^{\sim} \\ \qquad \mathbin{;} (\lambda\, r : \mathrm{dom}(F\ f) \bullet \\ \qquad\qquad (\Phi_\iota\ R)^{\sim} \\ \qquad\qquad \mathbin{;} (\lambda\, t : \mathrm{dom}(F\ f\ r) \bullet \Phi_\iota\ R\ (F\ f\ r\ t)))) \\ \hline \end{array}$$

F_1 defines the relabel functions from F in terms of the classes in *MLSPolicy*. An MLS implementation of a relabel policy is simply a relabel policy applied to the transformed lattice.

$$\begin{array}{l} MLSRPolicy \mathrel{\widehat=} RPolicy_1[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS] \\ MLSRelabel \mathrel{\widehat=} Relabel_1[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS] \end{array}$$

Finally, the MLS implementation of the initial relabel policy can be constructed using the transformation:

$$InitMLSRPolicy\Phi \mathrel{\widehat=} MLSPolicy\Phi \wedge MLSRFun\Phi$$

Thus, the initial MLS policy for *Segregation* can be computed using $InitMLSRPolicy\Phi$. An MLS implementation of the *MLSRelabel* transformation should be implemented as a relabel macro [11]. This operation implements relabel functions in terms of the security primitives provided by an underlying security kernel.

4.2 Unix Implementation

A Unix policy is *Policy* augmented with grp_1 and mbr_1.

$$\begin{array}{l} UnixRPol \mathrel{\widehat=} UnixPol \wedge RFuns_1[UID] \\ UnixRelabel \mathrel{\widehat=} Relabel_1[UID] \wedge \Delta\, UnixRPol \end{array}$$

With relabeling, only β_1 changes, and since $R_1 = R_1'$, it follows from *UnixRPol* that grp_1 and mbr_1 are also unchanged. The initial Unix configuration for $RPolicy_1[UID]$ is gotten by constructing grp_1 and mbr_1 according to *UnixRPol*. Thus, the initial Unix configuration for *Segregation* (Example 8) is computed as ($Segregation_1 \wedge UnixRPol$). A Unix implementation of the *UnixRelabel* transformation modifies the ownership of entities, and thus should be (carefully) implemented as a SUID `root` program.

5 Shadowed Relabel Policies

A potential problem with our approach to constructing security policies is that it can lead to a large number of security classes. For example, in the Unix implementation, each security class must be configured as a (phantom) UID, with corresponding GID and entries in `/etc/group`.

This section proposes a solution whereby an implementation policy is computed for only those classes referenced in the initial bindings for users, TPs and CDIs. When a relabel request is made for a class not in this initial configuration, the relations in the implemented policy are modified in a manner that produces the same effect, instead of performing the relabeling.

Given $R : \mathcal{R}[C]$ and $g : C \nrightarrow C$, then $R \wr g$ is the policy abstraction $R@(\mathrm{ran}\ g)$, except that each class $a \in (\mathrm{dom}\, g)$ is used in $R \wr g$ to represent class $g(a) \in \alpha(R@(\mathrm{ran}\ g))$.

$$
\begin{array}{|l}
[C] \\
\_ \wr \_ : \mathcal{R}[C] \times (C \mathrel{+\!\!\!\!\to} C) \to \mathcal{R}[C] \\
\hline
\forall R : \mathcal{R}[C];\ g : C \mathrel{+\!\!\!\!\to} C \bullet \\
\quad R \wr g = g \mathbin{;} R \mathbin{;} g^{\sim}
\end{array}
$$

It follows from this definition that $R \wr g$ preserves the orderings of R, in the sense that:

$$
\begin{array}{l}
\vdash \forall R : \mathcal{R}[C];\ g : C \mathrel{+\!\!\!\!\to} C \bullet \\
\quad \forall a, b : \mathrm{dom}\, g \bullet \\
\qquad (a, b) \in R \wr g \Leftrightarrow (g(a), g(b)) \in R @ (\mathrm{ran}\, g)
\end{array} \qquad [L1]
$$

A number of other results follow immediately from this definition.

$$
\begin{array}{l}
\vdash \forall R : \mathcal{R}[C];\ g : C \mathrel{+\!\!\!\!\to} C \bullet \\
\quad \alpha(R \wr g) = (\alpha R) \cap \mathrm{dom}\ g
\end{array} \qquad [L2]
$$

$$
\begin{array}{l}
\vdash \forall R : \mathcal{R}[C];\ g, h : C \mathrel{+\!\!\!\!\to} C \mid \mathrm{dom}\, h \subseteq \mathrm{dom}\, g \bullet \\
\quad \alpha(R \wr (g \oplus h)) = \alpha(R \wr g)
\end{array} \qquad [L3]
$$

$$
\begin{array}{l}
\vdash \forall R : \mathcal{R}[C];\ g, h : C \mathrel{+\!\!\!\!\to} C \mid \mathrm{dom}\, h \subseteq \mathrm{dom}\, g \bullet \\
\quad R \wr (g \oplus h) = (R \wr g) @ (\mathrm{dom}\, g \setminus \mathrm{dom}\, h) \\
\qquad \cup ((g \oplus h) \mathbin{;} R \mathbin{;} h^{\sim}) \cup (h \mathbin{;} R \mathbin{;} (g \oplus h)^{\sim})
\end{array} \qquad [L4]
$$

This last law holds due to the disjunctivity of the $\mathbin{;}$ and $^{\sim}$ operators [19], and given the fact that $(R \wr g) @ (\mathrm{dom}\, g \setminus \mathrm{dom}\, h) = R \wr (\mathrm{dom}\, h \lhd g)$. These laws form the basis of our *shadowing* of relabel policies. Consider a request to relabel all entities bound to class a to another class b. Rather than performing the relabeling, we could instead modify the flow relation R to $R \wr ((\mathrm{id}\, \alpha R) \oplus \{a \mapsto b\})$, and note that class a in this new policy is a shadow, or an alternate representation, of class b. In this case: Law L1 implies that the original orderings are preserved; Law L3 implies that entity classifications need not be changed, and Law L4 indicates that the calculation of the new policy is based on the previous value for R plus a re-calculation of orderings for flows involving b.

This scheme is generalized as follows. Given a reflexive relation R_1 we maintain a (smaller) shadow relation $shadow_1$, and a representation function rep_1 such that $rep_1(a)$ gives the class in αR_1 that is currently represented by class a in $\alpha\, shadow_1$. Formally,

$$
\begin{array}{|l}
\textit{ShadowPol}[C] \\
\textit{RPolicy}_1[C] \\
shadow_1 : \mathcal{R}[C] \\
rep_1 : C \mathrel{+\!\!\!\!\to} C \\
\hline
\mathrm{dom}\ rep_1 = \mathrm{ran}\, \beta_1 \\
\mathrm{ran}\, rep_1 \subseteq \alpha R_1 \\
shadow_1 = R_1 \wr rep_1
\end{array}
$$

Note that rep_1 is defined for only those classes referenced in β_1, and thus $\alpha\, shadow_1 \subseteq \alpha R_1$. Informally, one may think of $shadow_1$ as that part of R_1 that is currently 'swapped-in'. Initially, the shadow of a policy R_1 is just those classes that are referenced in the initial binding of a relabel policy, that is $R_1 @ (\mathrm{ran}\, \beta_1)$. This can be specified in terms of rep_1 as:

$$
\begin{array}{|l}
\textit{InitialShadow}[C] \\
\textit{ShadowPol}[C] \\
\hline
rep_1 = \mathrm{id}(\mathrm{ran}\ \beta_1)
\end{array}
$$

A theorem that follows immediately from Law L1 is that: for any legal configuration of a shadow policy, then the flow restrictions between entities, specified in R_1, are preserved by the shadow of R_1. That is,

$$
\begin{array}{l}
\textit{ShadowPol}[C] \vdash \\
\quad \forall x, y : ENTS_1 \bullet \\
\qquad (\beta_1(x), \beta_1(y)) \in shadow_1 \\
\qquad \Leftrightarrow (rep_1(\beta_1(x)), rep_1(\beta_1(y))) \in R_1
\end{array}
$$

A relabel request is implemented as a modification of $shadow_1$ and rep_1.

$$
\begin{array}{|l}
\textit{RelabelShadow}[C] \\
\Delta \textit{ShadowPol}[C] \\
req? : ID \\
target? : C \\
f? : FID \\
\hline
req? \in USR_1 \wedge f? \in \mathrm{dom}\, F_1 \\
rep_1(\beta_1\ req?) \in \mathrm{dom}(F_1\ f?) \\
target? \in \mathrm{dom}(F_1\ f?\ (rep_1(\beta_1\ req?))) \\
rep_1' = rep_1 \oplus \{\ a : C \mid rep_1\ a = target? \bullet \\
\qquad (a \mapsto F_1\ f?\ (rep_1(\beta_1\ req?))\ target?)\ \} \\
shadow_1' = R_1 \wr rep_1' \\
\theta RPolicy_1 = \theta RPolicy_1'
\end{array}
$$

A valid request to relabel all entities bound to class $target?$ as $(F_1\ f?\ (rep_1(\beta_1\ req?))\ target?)$ is implemented by updating rep_1 so that class $target?$ represents the new class, and then re-evaluating $shadow_1$ for this new representation. In contrast to specification *Relabel*, the entity binding β_1 here remains fixed. Note that since the definition of rep_1' is given in terms of a function override on rep_1 for the target class, then Law L4 implies that the calculation, $shadow_1' = R_1 \wr rep_1'$, can be implemented in terms of the original $shadow_1$ and a re-calculation of orderings involving just $target?$. Proof that *RelabelShadow* is a valid implementation of *Relabel* is given in Appendix B.

5.1 Multilevel Implementation

The MLS implementation of shadow policies is similar to the MLS implementation of the original relabel policies. We have

$$
\begin{array}{l}
\textit{MLSShadowPol} \mathrel{\widehat{=}} \textit{ShadowPol}[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS] \\
\textit{MLSRelabelS} \mathrel{\widehat{=}} \textit{RelabelShadow}[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS] \\
\textit{InitMLSShadowPol}\Phi \mathrel{\widehat{=}} \textit{InitMLSRPolicy}\Phi \\
\qquad \wedge\ \textit{InitialShadow}[\mathbb{P}\ CLASS \times \mathbb{P}\ CLASS]
\end{array}
$$

While we use $\textit{InitMLSRPolicy}\Phi$ to compute the initial policy, the actual MLS implementation lattice constructed for the initial configuration of the policy should be built from the initial shadow policy $shadow_1 = R_1 \wr (\mathrm{id}\, \beta_1)$, rather than from reflexive relation R_1. Thus we define the implementation lattice as

$$
\begin{array}{|l}
\textit{MLSShadowPolImp} \\
\textit{MLSShadowPol} \\
L_1 : \mathcal{R}[\mathbb{P}\ CLASS] \\
\hline
L_1 \supseteq \{\ A, B : \mathit{first}(\!|\alpha\, shadow_1|\!) \cup \mathit{second}(\!|\alpha\, shadow_1|\!) \\
\qquad \mid A \subseteq B\ \}
\end{array}
$$

5.2 Unix Implementation

Following the Unix implementation approach in Section 3.2 we augment *ShadowPol* by grp_1 and mbr_1, but configured for $shadow_1$, rather than for R_1.

$$
\begin{array}{|l}
\hline
\textit{UnixShadowPol} \\
\textit{ShadowPol}[\textit{UID}] \\
\textit{UnixPolicy} \\
\hline
\mathrm{dom}\, grp_1 = \mathrm{ran}\, \beta_1 \\
mbr_1 = shadow_1 \mathbin{⨾} grp_1 \\
\hline
\end{array}
$$

Since mbr_1 is defined in terms of $shadow_1$, then an update to $shadow_1$ should result in a corresponding update to mbr_1.

$$
\begin{array}{l}
\textit{UnixRelabelShadow} \mathrel{\widehat{=}} \\
\qquad \textit{RelabelShadow}[\textit{UID}] \wedge \Delta\, \textit{UnixShadowPol}
\end{array}
$$

And the initial configuration for grp_1 and mbr_1 can be determined as *InitialShadow*, that is

$$
\begin{array}{l}
\textit{InitUnixShadowPol} \mathrel{\widehat{=}} \\
\qquad \textit{UnixShadowPol} \wedge \textit{InitialShadow}[\textit{UID}]
\end{array}
$$

6 Conclusion

The proposed policy framework can be used to express a wide variety of confidentiality and integrity requirements and these can, in turn, be implemented as MLS or Unix based policies. The MLS approach may be used for security critical systems, while the Unix approach may be taken for less critical applications. Our policy framework is a further example of the usefulness of the tiered verification approach, which is based on relabeling, as proposed in [11].

The paper makes a number of contributions; in particular, it represents a generalization and unification of results from [6, 10, 11, 14, 16, 18]. The paper illustrates how relabeling policies [11] can be expressed in terms of reflexive flow policies [10], providing a systematic way of developing and implementing security requirements. We show how these policies can be used to specify Clark-Wilson access requirements including dynamic segregation of duty, the implementation of which, extends the original results in [14, 16, 18]. Shadowing is used to reduce the size of an implementation lattice, or the number of Unix UIDs allocated. In the latter case it provides a technique for 'relabeling' users, but without having to re-allocate UIDs.

The Chinese-Wall policy in Example 9 can be implemented in Unix. Our results on shadowing imply that such relabel policies can also be captured in terms of re-configuring the underlying security policy. It turns out that this alternative implementation corresponds to our original Unix encoding of the Chinese-Wall policy [8] which was implemented by re-configuring the `/etc/group/` as accesses are requested.

We believe that our policies can be implemented in terms of other security mechanisms, such as type-enforcement [2, 20]. Segregation of duty is specified as a relabel policy and therefore we should be able use the analysis techniques proposed in [11] to determine if dynamic segregation of duty can result in covert channels. This is ongoing work which we hope to report on in the future.

Acknowledgments

This work was supported by Forbairt under Basic Research Grant SC/96/611. The author would like to thank his colleague John Morrison and the anonymous referees for their comments on an earlier version of this paper.

References

[1] D. Bell. Secure computer systems: A network interpretation. In *Proceedings of the Aerospace Computer Security Applications Conference*, pages 32–39. IEEE Computer Society Press, 1986.

[2] W. Bobert and R. Kain. A practical alternative to hierarchical integrity properties. In *Proceedings of the National Computer Security Conference*, pages 18–27, 1985.

[3] M. Branstad et al. Trusted Mach design issues. In *Proceedings Third Aerospace Computer Security Conference*, 1987.

[4] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.

[5] C. Bryce. Lattice-based enforcement of access control policies. Technical Report 1011, GMD, Institute SET-RS, Sankt Augustin, Germany, Aug. 1996.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security models. In *Proceedings 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, Apr. 1987.

[7] Denning, D.E. On the derivation of lattice structured information flow policies. Technical Report CSD TR180, Purdue University, 1976.

[8] S. Foley. Building Chinese Walls in standard Unix. In Supplement to the Proceedings of the 1996 IEEE Symposium on Security and Privacy (Five-Minute Abstracts). Full length version submitted for publication.

[9] S. Foley. Aggregation and separation as noninterference properties. *Journal of Computer Security*, 1(2):159–188, 1992.

[10] S. Foley. Reasoning about confidentiality requirements. In *Proceedings of the Computer Security Foundations Workshop*, pages 150–160, Franconia, NH, June 1994. IEEE Computer Society.

[11] S. Foley, L. Gong, and X. Qian. A security model of dynamic labeling providing a tiered approach to verification. In *Proceedings of the Symposium on Security and Privacy*, pages 142–153, Oakland, CA, May 1996. IEEE Computer Society Press.

[12] S. Garfinkel and G. Spafford. *Practical Unix & Internet Security*. O'Reilly & Associates, 1996.

[13] J. Jacob. The varieties of refinement. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 441–455. Springer-Verlag, 1991.

[14] T. Lee. Using mandatory integrity to enforce 'commerical' security. In *Proceedings of the Symposium on Security and Privacy*, pages 140–146, 1988.

[15] M. Nash and K. Poland. Some conundrums concerning separation of duty. In *Proceedings of the Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990. IEEE Computer Society Press.

[16] W. Polk. Approximating Clark-Wilson access triples with basic UNIX controls. In *Unix Security Symposium IV*, pages 145–154, 1993.

[17] R. Sandhu. Lattice based access control models. *IEEE Computer*, 26(11):9–19, Nov. 1993.

[18] W. Shockley. Implementing the Clark Wilson integrity policy using current technology. In *Proceedings of the National Computer Security Conference*, pages 29–36, 1988.

[19] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, second edition, 1992.

[20] D. Thomsen and J. Haigh. A comparison of type enforcement and Unix setuid implementation of well-formed transactions. In *Computer Security Applications Conference*, pages 304–312. IEEE Computer Society Press, 1990.

A Notation

A.1 The Z Notation

A set may be defined in Z using set specification in comprehension. This is of the form $\{ D \mid P \bullet E \}$, where D represents declarations, P is a predicate and E an expression. The components of $\{ D \mid P \bullet E \}$ are the values taken by expression E when the variables introduced by D take all possible values that make the predicate P true. When there is only one variable in the declaration and the expression consists of just that variable, then the expression may be dropped if desired.

In Z, relations and functions are represented as sets of pairs. A (binary) relation R, declared as having type $A \leftrightarrow B$, is a component of $\mathbb{P}(A \times B)$. For $a \in A$ and $b \in B$, then the pair (a, b) is written as $a \mapsto b$, and $a \mapsto b \in R$ means that a is related to b under relation R. Functions are treated as special forms of relations. A lambda abstraction, written as $(\lambda\, x : X \mid P(x) \bullet E(x))$ specifies a partial function that maps values $x : X$ (where $P(x)$ holds) to $E(x)$.

The Schema notation is used to structure specifications in Z. A schema such as *Ents* defines a collection of variables (limited to the scope of the schema), and specifies how they are related. Schemas may be defined in terms of other schemas. For example, the inclusion of *Ents* within schema *Policy* is equivalent to the syntactic inclusion of the variables and predicates of *Ents* within *Policy*. Schemas may be composed using logical operators. For example, $UnixPol \wedge RFuns_1[UID]$ is a schema with variables and predicates from both *UnixPol* and $RFuns_1$.

The decorated schema $Policy_1$ is *Policy* with all variables decorated by the subscript 1. The schema $\Delta RPolicy$ is a syntatic sugar for $RPolicy \wedge RPolicy'$. It is typically used for specifying state transitions, with undecorated variables representing 'before values' and decorated (primed) variables representing 'after values'. $\theta Ents$ gives a schema type with variables from *Ents*. Put simply, predicate $\theta Ents = \theta Ents_1$ in schema $MLSPolicy\Phi$ is equivalent to specifying $USR = USR_1$, $TP = TP_1$, and so forth.

$first(a, b)$	Component a of ordered pair (a, b)
$second(a, b)$	Component b of ordered pair (a, b)
$\mathbb{P}\, A$	The power set of A
$\mathbb{F}\, A$	The set of finite sets from $\mathbb{P}\, A$
$\bigcup K$	Distributive union over the sets in K
$A \leftrightarrow B$	Relations between A and B
$A \rightarrow B$	Total functions from A to B
$A \nrightarrow B$	Partial functions in $A \rightarrow B$
$A \rightarrowtail\!\!\!\!\!+ B$	Partial injective functions in $A \nrightarrow B$
$\mathrm{dom}\ R, \mathrm{ran}\ R$	Domain and Range of relation R
$\mathrm{id}\ A$	Identity relation over values from A
$R \mathbin{;} S$	Relational composition
$R(\!\mid A \mid\!)$	Image of set A through relation R.
$R^{\sim}$	The inverse of relation R
$R \oplus G$	The relational override of R by G
$A \lhd R$	Relation R with its domain restricted to values from A
$\mathrm{id}\ A$	Identity relation over A

A.2 Constructing Reflexive Relations

The policy construction operators used in this paper are defined as:

$$
\begin{array}{l}
[X] \\
\hline
\bot : (\mathbb{P}\ X) \rightarrow \mathcal{R}[X] \\
\_ \rightsquigarrow \_ : ((\mathbb{P}\ X) \times (\mathbb{P}\ X)) \rightarrow \mathcal{R}[X] \\
\_@\_, \_ \uparrow \_ : (\mathcal{R}[X] \times \mathbb{P}\ X) \rightarrow \mathcal{R}[X] \\
\_ \sqcap \_ : (\mathcal{R}[X] \times \mathcal{R}[X]) \rightarrow \mathcal{R}[X] \\
\mathrm{not} : \mathcal{R}[X] \rightarrow \mathcal{R}[X] \\
\hline
\bot A = A \times A \\
A \rightsquigarrow B = \mathrm{id}(A \cup B) \cup (A \times B) \\
R@A = \{a, b : (A \cap \alpha R) \mid (a, b) \in R\} \\
R \uparrow A = \{a, b : (A \cup \alpha R) \mid \{a, b\} \subseteq \alpha R \Rightarrow (a, b) \in R\} \\
R \sqcap Q = (R \uparrow \alpha Q) \cap (Q \uparrow \alpha R) \\
\mathrm{not}\ R = (\top(\alpha R)) \cup ((\bot(\alpha R)) \setminus R)
\end{array}
$$

A reflexive relation is mapped to a lattice by function Φ_ι.

$$
\begin{array}{l}
[X] \\
\hline
\Phi_\iota : \mathcal{R}[X] \rightarrow X \nrightarrow (\mathbb{P}\ X \times \mathbb{P}\ X) \\
\Phi : \mathcal{R}[X] \rightarrow \mathcal{R}[\mathbb{P}\ X \times \mathbb{P}\ X] \\
\hline
\Phi_\iota\ R\ a = (\{b : X \mid \mathrm{dom}(\{a\} \lhd R) \subseteq \mathrm{dom}(\{b\} \lhd R)\}, \\
\qquad \{b : X \mid (b, a) \in R\}) \\
\Phi\ R = \{a, b : \alpha R \bullet (\Phi_\iota\ R\ a, \Phi_\iota\ R\ b)\}
\end{array}
$$

The mapping is order-preserving, in that for any $R : \mathcal{R}[X]$:

$$
\begin{array}{l}
\vdash \forall\, a, b : \alpha R \bullet \\
\qquad (a, b) \in R \Leftrightarrow first(\Phi_\iota\ R\ a) \subseteq second(\Phi_\iota\ R\ b)
\end{array}
$$

B Correctness of Shadow Policies

We prove that the shadow policy implementation is a refinement, in the sense of [19], of the relabel policy specification.

B.1 Data Refinement

The relabel policy implemented by a shadow policy can be specified according to the following abstraction (retrieve) function *Abs*.

$$
\begin{array}{l}
Abs[C] \\
\hline
RPolicy[C] \\
ShadowPol[C] \\
\hline
\theta Ents = \theta Ents_1 \\
\theta RFuns = \theta RFuns_1 \\
\beta = \beta_1 \mathbin{;} rep_1
\end{array}
$$

The policy components are the same except that we retrieve β (which changes with relabeling) from β_1 (remains static) and rep_1 (changes with relabeling).

Initial States Theorem. Since *ShadowPol* is defined in terms of *Relabel*, then it follows that we can retrieve from an initial shadow policy, using *Abs*, its abstract relabel policy. That is,

$$InitialShadow'[C] \wedge Abs[C] \vdash InitRPolicy[C]$$

B.2 Operation Refinement

The *Relabel* transition updates β by relabeling all entities bound to *target?* by a new class specified by the relabel function. It is the only component of the policy that changes, and therefore, if the relabeling is applicable given $req?$ and $target?$, then the transition is applicable. Thus we can compute the precondition of $Relabel$ to be

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\,PreRelabel[C]\,\rule{0.6\textwidth}{0.4pt}} \\
RPolicy[C] \\
req? : ID \\
target? : C \\
f? : FID \\
\hline
req? \in USR \land f? \in \operatorname{dom} F \\
\beta\ req? \in \operatorname{dom}(F\ f?) \\
target? \in \operatorname{dom}(F\ f?\ (\beta\ req?)) \\
\hline
\end{array}
$$

The $RelabelShadow$ does not modify β_1, but updates rep_1 and $shadow_1$. No other variables are modified, and since we have $shadow'_1 = R_1 \wr rep'_1$, which is equivalent to $shadow'_1 = R'_1 \wr rep'_1$, then the invariant holds on $ShadowPol'$. Thus it follows that we can compute the precondition of $RelabelShadow$ to be

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\,PreRelabelShadow[C]\,\rule{0.5\textwidth}{0.4pt}} \\
ShadowPol[C] \\
req? : ID \\
target? : C \\
f? : FID \\
\hline
req? \in USR_1 \land f? \in \operatorname{dom} F_1 \\
rep_1(\beta_1\ req?) \in \operatorname{dom}(F_1\ f?) \\
target? \in \operatorname{dom}(F_1\ f?\ (rep_1(\beta_1\ req?))) \\
\hline
\end{array}
$$

Applicability Theorem. It must be safe to apply a shadow relabel request whenever it would be safe to apply the same request to its corresponding abstract relabel policy. The retrieve function defines $\beta = \beta_1 \fatsemi rep_1$ and thus we have $\beta(req?) = rep_1(\beta_1(req?))$. Thus it follows that

$$PreRelabel[C] \land Abs[C] \vdash PreRelabelShadow[C]$$

Correctness Theorem. $RelabelShadow$ must update a shadow policy correctly. Compare the definition of rep'_1 in schema $RelabelShadow$ with β' in $RelabelPolicy$. They perform the same function in sense that we have $\beta' = \beta'_1 \fatsemi rep'_1$. Thus we have

$$
\begin{array}{l}
PreRelabel[C] \land Abs[C] \\
\land\ RelabelShadow[C] \land Abs'[C] \vdash Relabel[C]
\end{array}
$$