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 be 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 labeling 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 [30]. 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.

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 labeling [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 jZig 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 only access CDIs via TPs. And then only if that access is specified in an E2 rule relation. For our purposes, an E2 rule 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 access the TP \(t\) to access the CDI \(c\). The set of all
possible access triple relations is defined to be \(AT[C]\), where
(generic) \(C\) represents the identifiers used for users, TP's and
CDIs.

\[ AT[C] = \{ T : P \subseteq (C \times C \times C) \mid (\forall u, u', t, c, c' : C \bullet
((u, t, c) \in T \land (u', t, c') \in T) \Rightarrow (u, t, c') \in T) \} \]

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 be-
cause 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 TP's one can encode
the access triples proposed in [6] as components of \(AT\).

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

\[ PassTrips == \{(\text{smith}, chpasswd, passwd)\} \]

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 \(postc\).

\[ ClerkTrips == \{(\text{smith}, posti, invs), (jones, postc, cons)\} \]

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 de-
mension how relations from \(AT\) may be expressed as reflexive (bi-
nary) 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 infor-
mation that can exist in a system and whether or not infor-
mation may flow between these classes. In [10] we suggest
that, in addition to considering the usual sensitivity lev-
els 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 TP's and CDI's.
This approach is also suggested in [5]. The set of all reflexive
relations between classes of (generic) type \(C\) is defined by
\(R[C]\), where

\[ R[C] = \{ R : C \rightarrow C \mid \text{idl}(\text{dom } R \cup \text{ran } R) \subseteq R \} \]

If \(R \in R[C]\) and \(a \rightarrow b \in R\), then we say that \(a\) is less
than, or equal to, \(b\) in \(R\). If the notation \(A \sim B\) 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

\[ MLS = \{\text{unclass, secret} \sim \{\text{secret, topsecret}\} \}
\]

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

\[ \alpha MLS = \{\text{unclass, secret, topsecret}\} \]

The set \(R[C]\) forms a lattice under a partial ordering \(\sqsubseteq\),
and lowest upper bound operator \(\sqcup\). 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 \(R \cap Q\) 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:

\[ PassRein == \{\text{smith, chpasswd, passwd}\} \]

\[ \land \not((\{\text{smith}\} \sim \{\text{passwd}\})) \]

where \(\land A\) gives the least restrictive policy with alphabet \(A\).
\(PassRein\) specifies that any flow is permitted, except from
\(smith\) to \(passwd\). This implies that \(smith\) may (directly)
modify \(passwd\). Note that information is permitted to flow
from \(passwd\) to \(smith\).

Given an access-triple relation \(T\), \(unzip(T)\) returns its equiv-
lent reflexive relation. The policy \(\not((\text{usr}(T) \setminus \text{tp}(T)) \sim \text{cdi}(T) \setminus \text{tp}(T))\)
specifies that information may not di-
rectly flow from (a class representing) a user to a CDI; how-
ever, for generality, the flow may be permitted if the
user or CDI also corresponds to a TP. The policy \(\bigcup t\in T\{ t \bullet\land\{\text{usr}(t),\text{tp}(t),\text{cdi}(t)\} \}
\land \not((\text{usr}(T) \setminus \text{tp}(T)) \sim (\text{cdi}(T) \setminus \text{tp}(T))\)

Given an access-triple relation \(T\), \(unzip(T)\) returns its equiv-
lent reflexive relation. The policy \(\not((\text{usr}(T) \setminus \text{tp}(T)) \sim \text{cdi}(T) \setminus \text{tp}(T))\)
specifies that information may not di-
rectly flow from (a class representing) a user to a CDI; how-
ever, for generality, the flow may be permitted if the
user or CDI also corresponds to a TP. The policy \(\bigcup t\in T\{ t \bullet\land\{\text{usr}(t),\text{tp}(t),\text{cdi}(t)\} \}
\land \not((\text{usr}(T) \setminus \text{tp}(T)) \sim (\text{cdi}(T) \setminus \text{tp}(T))\)

### 3 Developing Security Policies

Let \(ID\) represent the set of all possible users, TP's and CDI's.
A security policy is defined in terms of a particular set of
these entities \(ENTS\).
Given $\text{Ents}$, 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

$$\text{Policy}[C]$$

$\beta : \text{ID} \rightarrow C$

$\forall \alpha \in \text{R} \cdot \beta \subseteq \alpha R$

$R$ gives the reflexive flow relation and $\beta$ associates each entity from $\text{ENT}$ 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 *smith*, *chpass* and *pswd*, (elements of basic type $\text{CLASS}$, identified using a *typewriter* font), respectively.

$$\text{PassPolicy} = \text{Policy[CLASS]}$$

$\text{USR} = \{\text{smith}\} \land \text{TP} = \{\text{chpass}\} \land \text{CDI} = \{\text{passwd}\}$

$\beta = \{\text{smith} \mapsto \text{smith}, \text{chpass} \mapsto \text{chpass}, \text{passwd} \mapsto \text{pswd}\}$

3.1 Multilevel Implementation

Shockey [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 $\text{CLASS}$ representing the set of security classes, then we specify a multilevel policy based on the powerset lattice of $P \text{CLASS}$ as

$$\text{MLSPolicy} = \text{Policy[P CLASS x P CLASS]}$$

$\forall (A, B) \in \alpha R_1 \Rightarrow A \subseteq B$

$\forall (A, B), (C, D) \in R_1 \Rightarrow A \subseteq C \land B \subseteq D$

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 $P \alpha R_1$. 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 $\alpha R_1$. The user is considered cleared to classes between $A$ and $B$ in the powerset lattice $P \alpha R_1$. 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 ($\text{amin}, \text{vmax}$) 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 $A \subseteq B$. 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 $\text{first}(\alpha R_1) \cup \text{second}(\alpha R_1)$, and adding additional classes until a lattice is formed [7]. Instead of reproducing the algorithm in [7], we specify

$$\text{MLSPolicyImp}$$

$\forall (A, B) : P \alpha R_1 \Rightarrow A \subseteq B$

$\exists L \subseteq \{A, B : \text{first}(\alpha R_1) \cup \text{second}(\alpha R_1) \mid A \subseteq B\}$

Informally, an MLS system is secure according to such a policy containing entities $x, y$, if information flows from $x$ to $y$ then $\text{first}(\beta_1(x)) \subseteq \text{second}(\beta_1(y))$ 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 $\beta_1$ bindings given in Figure 1. Note that we use $s$ to abbreviate *smith* and so forth. Using these bindings with the implementation strategy in [14], it follows that *smith* may not modify *passwd*, except via *chpass*.

$$\text{MLSPolicy$\Phi$}$$

$\forall \text{Ents} = \text{ENTs}$

$\beta_1 = \beta_1(\Phi, R)$

$R_1 = \Phi R$

It follows, from the order-preserving property of $\Phi$ [10], that

$$\text{MLSPolicy$\Phi$} \vdash \forall x, y \in \text{ENTs} \bullet \left( \text{first}(\beta_1(x)) \subseteq \text{second}(\beta_1(y)) \Rightarrow (\beta_1(x), \beta_1(y)) \in R \right)$$

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.
A Unix system can be configured to support the CW-model [16, 20] by implementing the access-triple relation as an en-coding 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 \( \{a \rightarrow \{b\}\} \) 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

```plaintext
UnixPolicy
Policy[UID]
g : \{a \rightarrow \{b\}\} GID
mb : \{a \rightarrow \{b\}\} GID
ran mb = ran g
dom mb = dom g
ran mb = R1 ; g
```

For our purposes, the injective function \( g \) associates a unique GID with each UID and relation \( mb \) specifies group membership (as implemented by file /usr/group). Note that \( g \) and \( mb \) are simply annotations for \( R1 \).

Each user \( u \) has an associated UID given by \( \beta(u) \). A CDI \( c \) is implemented as a file, owned by \( \beta(c) \) and with group-id \( g \) (\( \beta(c) \)). Its access permission bits should be set to only \( Rk \) for owner and group. TP \( t \) is an executable program, stored in a file, with UID \( \beta(t) \), GID \( g \) (\( \beta(t) \)), and has permission bits set to execute by user and group only, and has its UID 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 SUID, chps, and pswd, and corresponding GIDs gchps, gcps, and gpwd, respectively. Files chps and pswd have the self-explanatory protection profiles:

<table>
<thead>
<tr>
<th>chps</th>
<th>rwx</th>
<th>r-</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>gchps</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>passwd</th>
<th>r-</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcps</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>gpwd</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

The \( mb \) relation is configured as follows. Smith is permitted to access (execute) the TP chps (\( \text{chps} \in g \)), which in turn may access the file passwd (\( \text{chps} \in g \)). But Smith may not directly access the password file (\( \text{chps} \notin g \)).

Given a policy specified in terms of Policy[UID], then allocating \( g \) and calculating \( mb \), according to UnixPolicy gives its Unix implementation. Since \( mb = R1 ; g \), it follows that the Unix implementation preserves the access constraints of the policy, that is,

```
UnixPolicy \rightarrow
\forall x, y : ENTS •
(\( \beta(x) \), \( g \) (\( \beta(y) \)) \in mb \( \rightarrow \) (\( \beta(x) \), \( \beta(y) \)) \in R1
```

For example, PassPolicy and UnixPolicy specifies a Unix implementation policy that is similar to that in Example 6.

Note that individual TPs 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 [16]. 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 \( RFuncs \):

```
RFuncs[C] = \{R : R[C] \subset C \rightarrow C \mid \forall f : ran F \in dom(f) \cup dom(ran f) \cup \{ran(ran f)\} \subseteq \alpha R \}
```

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

```
Policy[C] = Policy[C] \land RFuncs[C]
```

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

```
Simple Dynamic RELabeling
RPolicy[\{CLASS\}]
\{\{opt : \{post \} \rightarrow FID \}
USR = \{\text{smth} \} \land TP = \{\text{post} \} \land CDI = \{\text{inv, cons}\}
R = \{\text{unzip}, \{\text{smth, post, post} \}, \{\text{smth, post, inv, cons}\}, \}
(\{\text{smth, post, cons}\})
\beta = \{(\text{smth} \rightarrow \text{smth, post} \rightarrow \text{post},
\text{inv, cons} \rightarrow \text{cons})
F = (\lambda f : \text{ran opt} •
(\lambda s : \text{smth} • (\lambda t : \text{post} • \text{opt} ))
```

Function opt is used to allocate unique FIDs for the two relabel functions defined in \( F \). The relabel function identified as \( opt \) (post1), relabels class post by class post1, while the
function \( F(\text{opt}(\text{post}^c)) \) relabels class \( \text{post} \) by class \( \text{post}^c \). Only the user smith 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 \( \text{post} \), and initially it may not access any CDI. Only by using the relabel functions in \( F \) can smith change the label for TP \( \text{post} \) and subsequently gain access to a CDI.

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:

\[
\text{InitRPolicy}[C] \equiv \text{RPolicy}[C]
\]

Initially, a relabel policy may have any legal configuration. Transition \( \text{Relabel} \) is atomic and specifies the effect that a relabel function has on a relabel policy.

\[
\text{RPolicy}[C] \equiv \text{RPolicy}[C]
\]

\[
\text{opt} : \text{TKIND} \to \text{FID}
\]

\[
\text{USR} = \text{clerk}(\text{NAME}) \land \text{TP} = \{\text{post}\}
\]

\[
\text{CDI} = \text{file}(\text{TKIND})
\]

\[
\beta = \{ n : \text{NAME} \lor (\text{clerk}(n) \to \text{clerk}(\text{none})) \}
\]

\[
\{\text{post} \to \text{post}(\text{none})\} \lor \text{file} \quad \text{\texttt{f}}
\]

\[
\text{R} = \text{MS} \{ n : \text{NAME} ; k : \text{TKIND} \to (\text{clerk}(n, k) \to \text{post}(\text{f}(k)) \}
\]

\[
\{\text{post} \to \text{post}(\text{none})\} \lor \text{file} \quad \text{\texttt{f}}
\]

\[
\text{F} = (\lambda \text{fd} : \text{opt}(\text{TKIND}) \to \text{file}(\text{f}(k)))
\]

\[
\{\lambda \text{req} : \text{clerk}(\text{NAME}) \times \text{TKIND} \to \text{file}(\text{f}(k))\}
\]

\[
\{\lambda \text{target} : \text{post}(\text{TKIND}) \to \text{post}(\text{opt}^c(\text{fd}))\}
\]

\[
\{\lambda \text{target} : \text{clerk}(\text{NAME} \times \text{none}) \to \text{clerk}(\text{first}(\text{clerk}(\text{target})), \text{opt}^c(\text{fd}))\}
\]

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

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 \( \text{post}(k) \) with class \( \text{post}(k) \). 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 \( \text{post}(n) \) TP for each \( \text{clerk}(n) \). Initially, each TP \( \text{post}(n) \) has classification \( \text{post}(n, \text{none}) \), and \( \text{clerk}(n) \) requests a relabel (to this TP) to \( \text{post}(n, k) \) for kind \( k \) transactions. Access triples are of the form \( \text{clerk}(n, k), \text{post}(n, k), \text{file}(k) \). Under this scheme a clerk's classification does not change. This strategy is illustrated in the next example.

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 \( \text{NAME} \) represent the set of all possible names for clerks. The free type, \( \text{TKIND} \), defines the different types of transactions in the system. \( \text{ID} \) represents the entities in the system, and \( \text{CL} \) defines the classes for these entities.

\( \text{TKIND} : = \text{invoice} \mid \text{note} \mid \text{payment} \mid \text{none} \)

\( \text{ID} : = \text{clerk}(\text{NAME}) \mid \text{post} \mid \text{file}(\text{TKIND}) \)

\( \text{CL} : = \text{clerk}(\text{NAME} \times \text{TKIND}) \mid \text{post}(\text{TKIND}) \mid \text{file}(\text{TKIND}) \)

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

\( \text{Segregation} \)

\( \text{RPolicy}(\text{CLASS}) \)

\( \text{opt} : \text{TKIND} \to \text{FID} \)

\( \text{USR} = \text{clerk}(\text{NAME}) \land \text{TP} = \{\text{post}\} \)

\( \text{CDI} = \text{file}(\text{TKIND}) \)

\( \beta = \{ n : \text{NAME} \lor (\text{clerk}(n) \to \text{clerk}(\text{none})) \}
\]

\( \{\text{post} \to \text{post}(\text{none})\} \lor \text{file} \quad \text{\texttt{f}}
\]

\( \text{R} = \text{MS} \{ n : \text{NAME} ; k : \text{TKIND} \to (\text{clerk}(n, k) \to \text{post}(\text{f}(k)) \}
\]

\( \{\text{post} \to \text{post}(\text{none})\} \lor \text{file} \quad \text{\texttt{f}}
\]

\( \{\lambda \text{req} : \text{clerk}(\text{NAME}) \times \text{TKIND} \to \text{file}(\text{f}(k))\}
\]

\( \{\lambda \text{target} : \text{post}(\text{TKIND}) \to \text{post}(\text{opt}^c(\text{fd}))\}
\]

\( \{\lambda \text{target} : \text{clerk}(\text{NAME} \times \text{none}) \to \text{clerk}(\text{first}(\text{clerk}(\text{target})), \text{opt}^c(\text{fd}))\}
\)
The informal approach in [8] can be formalized in terms of
security relations involving access to datasets. This requires that analysts access datasets only via TP's. The informal approach in [8] can be formalized in terms of a more general Relabel strategy as follows.

\[ ID := \text{dataset}(\text{ORG}) \cup \text{analyst}(\text{NAME}) \cup \text{advise}(\text{NAME}) \]

\[ CL := \text{dataset}(\text{ORG}) \cup \text{analyst}(\text{NAME}) \cup \text{adv}(\text{NAME} \times F \text{ ORG}) \]

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

\[ \text{Chinese Wall} \]

\[ \text{RPolicy}[\text{CL}] \]

\[ \beta := \text{ORG} \rightarrow \text{ORG} \]

\[ \text{opt} := \text{ORG} \rightarrow \text{FID} \]

\[ \text{USR} = \text{analyst}(\text{NAME}) \land \text{TP} = \text{advise}(\text{NAME}) \]

\[ \text{CDI} = \text{dataset}(\text{ORG}) \]

\[ R = \text{unzip}\{ o : \text{ORG}; F : \text{ORG}; n : \text{NAME} \}
\]

\[ = \{ \text{mbr}(n), \text{ads}(n, O \cup \{O\}), \text{dataset}(o) \} \}

\[ \land \text{not}(\text{dataset}(\text{ORG}) \rightarrow \text{analyst}(\text{NAME})) \]

\[ \beta = \text{analyst}(\text{NAME}) \cup \text{dataset} \cup \text{dataset} \cup \text{dataset} \cup \text{dataset} \]

\[ F = (\lambda \text{fd} : \text{opt}(\text{ORG}) \bullet
\]

\[ \text{RFuture} \]

\[ \text{RFuture}[\text{CLASS} \times \text{CLASS}] \]

\[ R_0 = \Phi \]

\[ F_1 = (\lambda \text{f} : \text{dom} F \bullet
\]

\[ (\Phi, R)^\perp
\]

\[ (\lambda \text{r} : \text{dom}(F \text{ f}) \bullet
\]

\[ (\Phi, R)^\perp
\]

\[ (\lambda \text{t} : \text{dom}(F \text{ f r t}) \bullet \Phi, R (F \text{ f r t}))))) \]

\[ F_1 \text{ defines the relabel functions from } F \text{ in terms of the classes in MLSPolicy} \text{. An MLS implementation of a relabel policy is simply a relabel policy applied to the transformed lattice.} \]

\[ \text{MLSPolicy} \equiv \text{RPolicy}[\text{CLASS} \times \text{CLASS}] \]

\[ \text{MLSRlabel} \equiv \text{Relabel}[\text{UID}] \times \Delta \text{UnixRPol} \]

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

\[ \text{InitMLSPolicy} \equiv \text{MLSPolicy} \bullet \text{MLSRlabel} \]

\[ \text{Thus, the initial MLS policy for Segregation can be computed using } \text{InitMLSPolicy}. \text{ An MLS implementation of the } \text{MLSRlabel} \text{ transformation should be implemented as a relabel macro} \].

\[ \text{4.2 Unix Implementation} \]

A Unix policy is Policy augmented with grp and mbr.

\[ \text{UnixRPol} \equiv \text{UnixPol} \land \text{RFuture}[\text{UID}] \]

\[ \text{UnixRLabel} \equiv \text{Relabel}[\text{UID}] \land \Delta \text{UnixRPol} \]

With relabeling, only \( \beta \) changes, and since \( R_1 = R_0 \), it follows from UnixRPol that grp and mbr are also unchanged. The initial Unix configuration for RPolicy[UID] is gotten by constructing grp and mbr according to UnixRPol. Thus, the initial Unix configuration for Segregation (Example 8) is computed as (Segregation \( \land \text{UnixRPol} \)). A Unix implementation of the UnixRLabel transformation modifies the ownership of entities, and thus should be (carefully) implemented as a SUID root program.

\[ \text{5 Shadowed Relabel Policies} \]

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 \( \text{R}_0 \equiv \text{R}[\text{C}] \) and \( \text{g} : \text{C} \rightarrow \text{C} \), then \( \text{R} \circ \text{g} \) is the policy abstraction \( \text{R}_0 \circ (\text{ran} \text{ g}) \), except that each class \( a \in \text{dom} \text{ g} \) is used in \( \text{R} \circ \text{g} \) to represent class \( g(a) \in \alpha(\text{R}_0 \circ (\text{ran} \text{ g})) \).
It follows from this definition that $R \cap g$ preserves the orderings of $R$, in the sense that:

\[
\forall R : \mathcal{R}[C]; \quad g : C \to C \quad \bullet \quad R \cap g = g \cap R \cap g
\]

A number of other results follow immediately from this definition.

\[
\forall R : \mathcal{R}[C]; \quad g : C \to C \quad \bullet \quad \alpha((R \cap g)@g) \subseteq (\alpha R) \cap \text{dom } g
\]

\[
\forall R : \mathcal{R}[C]; \quad g, h : C \to C \quad \bullet \quad \alpha((R \cap (g@h))) = \alpha((R \cap g) \cap (g \cap h))
\]

This last law holds due to the disjointivity of the $\cap$ and $\text{dom}$ operators [19], and given the fact that $(R \cap g)@\text{dom } g \cap h = R \cap (\text{dom } g \cap h \subseteq g)$. 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 $R \cap \{(a \mapsto b)\}$, 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-calculating of orderings for flows involving $b$.

This scheme is generalized as follows. Given a reflexive relation $R_i$, we maintain a (smaller) shadow relation $\text{shadow}_i$, and a representation function $\text{rep}_i$, such that $\text{rep}_i(a)$ gives the class in $\alpha R_i$ that is currently represented by class $a$ in $\text{shadow}_i$. Formally,

\[
\begin{align*}
\text{ShadowPol}[C] & \quad R \text{Policy}_i[C] \\
\text{shadow}_i : \mathcal{R}[C] & \quad \text{rep}_i : C \to C \\
\text{dom } \text{rep}_i & = \text{ran } \beta_i \\
\text{ran } \text{rep}_i & \subseteq \alpha R_i \\
\text{shadow}_i & = R_i \cap \text{rep}_i
\end{align*}
\]

Note that $\text{rep}_i$ is defined for only those classes referenced in $\beta_i$, and thus $\text{shadow}_i \subseteq \alpha R_i$. Informally, one may think of $\text{shadow}_i$ as that part of $R_i$ that is currently 'wrapped'-in. Initially, the shadow of a policy $R_i$ is just those classes that are referenced in the initial binding of a relabel policy, that is $R_i \cap \text{ran } \beta_i$. This can be specified in terms of $\text{rep}_i$ as:

\[
\begin{align*}
\text{InitialShadow}[C] & \quad \text{ShadowPol}[C] \\
\text{rep}_i & = \text{id}(\text{ran } \beta_i)
\end{align*}
\]

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 $R_i$, are preserved by the shadow of $R_i$. That is,

\[
\text{ShadowPol}[C] \quad \text{relabelling}
\]

\[
\forall x, y : \text{ENT}, \quad (\beta_1(x), \beta_1(y)) \in \text{shadow}_i
\]

\[
\Rightarrow (\text{rep}_i(\beta_1(x)), \text{rep}_i(\beta_1(y))) \in R_i
\]

A relabel request is implemented as a modification of $\text{shadow}_i$ and $\text{rep}_i$.

\[
\begin{align*}
\text{RelabelShadow}[C] & \quad \Delta \text{ShadowPol}[C] \\
\text{req} : \text{id} & \quad \text{target} : C \\
\text{f} : \text{FID} & \quad \text{req} \in \text{USR} \land \text{f} \in \text{dom } F_i \\
\text{rep}_i(\beta_i(\text{req} )) & \in \text{dom } (F_i, \text{f}) \\
\text{target} \in \text{dom } (F_i, \text{f}) & \Rightarrow (\text{rep}_i(\beta_i(\text{req} )) ) \text{target} \} \\
\text{shadow}_i[R_i] \cap \text{rep}_i & \subseteq \text{target} \} \\
\text{rep}_i[R_i] & = \text{target} \}
\end{align*}
\]

A valid request to relabel all entities bound to class $\text{target}$ as $(F_i, \text{f})$ ($\text{rep}_i(\beta_i(\text{req} ))$) is implemented by updating $\text{rep}_i$, so that class $\text{target}$ represents the new class, and then re-evaluating $\text{shadow}_i$ for this new representation. In contrast to specification $\text{Relabel}$, the entity binding $\beta_i$ here remains fixed. Note that since the definition of $\text{rep}_i$ is given in terms of a function override on $\text{rep}_i$ for the target class, then Law L4 implies that the calculation, $\text{shadow}_i[R_i] \cap \text{rep}_i = \text{target}$, can be implemented in terms of the original $\text{shadow}_i$, and a re-calculation of orderings involving just $\text{target}$. Proof that $\text{RelabelShadow}$ is a valid implementation of $\text{Relabel}$ 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:

\[
\begin{align*}
\text{MLSShadowPol} & \equiv \text{ShadowPol}[P \text{CLASS} \times P \text{CLASS}] \\
\text{MLSShadowPolicy} & \equiv \text{RelabelShadow}[P \text{CLASS} \times P \text{CLASS}] \\
\text{InitMLSShadow} & \equiv \text{InitMLSRPolicy} \land \text{InitialShadow}[P \text{CLASS} \times P \text{CLASS}]
\end{align*}
\]

While we use $\text{InitMLSRPolicy}$ 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 $\text{shadow}_i = R_i \cap \text{ran } \beta_i$, rather than from reflexive relation $R_i$. Thus we define the implementation lattice as:

\[
\begin{align*}
\text{MLSShadowPolImp} & \equiv \text{ShadowPol}[P \text{CLASS}] \\
L_1 & : \text{R}[P \text{CLASS}] \\
L_1 & \supseteq \left\{ A, B : \text{first } \{x \text{shadow}_i\} \cup \text{second } \{x \text{shadow}_i\} \right\}
\end{align*}
\]
5.2 Unix Implementation

Following the Unix implementation approach in Section 3.2 we augment ShadowPol by grp, and mbr, but configured for shadow, rather than for R.

\[
\begin{align*}
\text{UnixShadowPol} & \subseteq \text{ShadowPol}(\text{UID}) \\
\text{UnixPolicy} & \subseteq \text{domain grp} = \text{range } \beta, \\
\text{mbr} & = \text{shadow } \beta, \text{ grp}
\end{align*}
\]

Since mbr, is defined in terms of shadow, then an update to shadow, should result in a corresponding update to mbr.

\[
\text{UnixRelabelShadow} \subseteq \text{RelabelShadow}(\text{UID}) \land \Delta \text{UnixShadowPol}
\]

And the initial configuration for grp, and mbr, can be determined as InitialShadow, that is

\[
\text{InitUnixShadow} \subseteq \text{UnixShadowPol} \land \text{InitialShadow}(\text{UID})
\]

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

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 \Rightarrow B, is a component of P(A \times B). For a \in A and b \in B, then the pair (a, b) is written as a \rightarrow b, and a \rightarrow 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 \texttt{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 \texttt{Ents} within schema Policy is equivalent to the syntactic inclusion of the variables and predicates of \texttt{Ents} within Policy. Schemas may be composed using logical operators. For example, \texttt{UnixPol \land RFuns[UID]} is a schema with variables and predicates from both \texttt{UnixPol} and \texttt{RFuns}.

The decorated schema Policy is Policy with all variables decorated by the subscript 1. The schema \( \Delta R\text{Policy} \) is a syntactic sugar for \( R\text{Policy} \land R\text{Policy}' \). It is typically used for specifying state transitions, with undecorated variables representing 'before values' and decorated (primed) variables representing 'after values'. \( \theta \text{Ents} \) gives a schema type with variables from \texttt{Ents}. Put simply, predicate \( \theta \text{Ents} = \theta \text{Ents}_1 \) in schema MLSPolicy\( \Phi \) is equivalent to specifying \( \text{USR} = \text{USR}_1 \), \( \text{TP} = \text{TP}_1 \), and so forth.

\begin{align*}
\text{first}(a, b) & : \text{Component } a \text{ of ordered pair } (a, b) \\
\text{second}(a, b) & : \text{Component } b \text{ of ordered pair } (a, b) \\
\mathcal{P}A & : \text{The power set of } A \\
\mathcal{F}A & : \text{The set of finite sets from } P A \\
\mathcal{A} \rightarrow \mathcal{B} & : \text{Relations between } A \text{ and } B \\
\mathcal{A} \rightarrow \mathcal{B} & : \text{Total functions from } A \text{ to } B \\
\mathcal{A} \rightarrow \mathcal{B} & : \text{Partial functions in } A \rightarrow B \\
\mathcal{A} \rightarrow \mathcal{B} & : \text{Partial injective functions in } A \rightarrow B \\
\text{dom } \mathcal{R} & : \text{Domain of relation } \mathcal{R} \\
\text{ran } \mathcal{R} & : \text{Range of relation } \mathcal{R} \\
id A & : \text{Identity relation over values from } A \\
R \mathrm{\mid} S & : \text{Relational composition} \\
R[A] & : \text{Image of set } A \text{ through relation } R. \\
\text{inv } R & : \text{The inverse of relation } R \\
R \circ G & : \text{The relational override of } R \text{ by } G \\
A \sqcap R & : \text{Relation } R \text{ with its domain restricted to values from } A \\
id A & : \text{Identity relation over } A 
\end{align*}

A.2 Constructing Reflexive Relations

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

\[
\begin{align*}
\downarrow : (P X) \rightarrow \mathcal{R}[X] \\
\neg \rightarrow \neg : ((P X) \times (P X)) \rightarrow \mathcal{R}[X] \\
\theta \neg \rightarrow \neg : (\mathcal{R}[X] \times P X) \rightarrow \mathcal{R}[X] \\
\neg \neg \rightarrow \neg : (\mathcal{R}[X] \times \mathcal{R}[X]) \rightarrow \mathcal{R}[X] \\
\not : \mathcal{R}[X] \rightarrow \mathcal{R}[X] \\
\downarrow A = A \times A \\
A \rightarrow B = \text{id}(A \cup B) \cup (A \times B) \\
\mathcal{R}\mathcal{A} = \{ (a, b) : (a \land \alpha R) \mid (a, b) \in R \} \\
\mathcal{R} \cup A = \{ (a, b) : (a \land \alpha R) \mid (a, b) \in R \} \\
\mathcal{R} \cup Q = (\mathcal{R} \cup Q \cap (Q \cup \alpha R)) \\
\not R = (\mathcal{R}(\alpha R)) \cup (\mathcal{R}(\alpha R)) \setminus R 
\end{align*}
\]

A reflexive relation is mapped to a lattice by function \( \Phi \).

\[
\begin{align*}
\Phi : \mathcal{R}[X] \rightarrow X \rightarrow (P X \times \mathcal{R}[X]) \\
\Phi : \mathcal{R}[X] \rightarrow \mathcal{R}[P X \times P X] \\
\Phi, R a = \{ (b : X \mid \text{dom}(a R) \subseteq \text{dom}(\{b \} \subseteq R)) \} \\
\Phi = \{ (a, b) : a R \bullet (\Phi, R a \bullet \Phi, R b) \}
\end{align*}
\]

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

\[
\downarrow \forall a, b : a R \bullet (a, b) \in R \Leftrightarrow \text{first}(\Phi, R a) \subseteq \text{second}(\Phi, R b)
\]

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 \( \text{Abs} \).

\[
\text{Abs} C \equiv
\begin{align*}
\text{Abs} C & : R\text{Policy}[C] \\
\text{Shadow Pol} & : C
\end{align*}
\]

\[
\text{Abs} C \equiv \text{Abs} C
\]

The policy components are the same except that we retrieve \( \beta \) (which changes with relabeling) from \( \beta \) (remains static) and \( \text{rep}_1 \) (changes with relabeling).

The \text{Initial States Theorem} since \text{Shadow Pol} is defined in terms of \text{Relabel}, then it follows that we can retrieve from an initial shadow policy, using \text{Abs}, its abstract relabel policy. That is,

\[
\text{InitialShadow}[C] \land \text{Abs} C \vdash \text{InitRPolicy[C]}
\]

B.2 Operation Refinement

The \text{Relabel} transition updates \( \beta \) by relabeling all entities bound to \text{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 \( \text{req}? \) and \( \text{target}? \), then the transition is applicable. Thus we can compute the precondition of \( \text{Relabel} \) to be

\[
\text{PreRelabel}[C] = RPolicy[C] \\
\text{ req}? : ID \\
\text{ target}? : C \\
\text{ f}? : FID \\
\text{ req}? \in \text{USR} \land f? \in \text{dom} F \\
\beta \text{ req}? \in \text{dom}(F, f?) \\
\text{ target}? \in \text{dom}(F, f? (\beta \text{ req}__))
\]

The \( \text{RelabelShadow} \) does not modify \( \beta \), but updates \( \text{rep}_1 \) and \( \text{shadow} \). No other variables are modified, and since we have \( \text{shadow}' = \text{R}\_1 \text{; rep}_1 \), which is equivalent to \( \text{shadow}' = \text{R}\_1 \text{; rep}_1' \), then the invariant holds on \( \text{ShadowPol}' \). Thus it follows that we can compute the precondition of \( \text{RelabelShadow} \) to be

\[
\text{PreRelabelShadow}[C] = \text{ShadowPol}[C] \\
\text{ req}? : ID \\
\text{ target}? : C \\
\text{ f}? : FID \\
\text{ req}? \in \text{USR} \land f? \in \text{dom} F \\
\text{ rep}_1(\beta \text{ req}?) \in \text{dom}(F, f?) \\
\text{ target}? \in \text{dom}(F, f? (\text{rep}_1(\beta \text{ req}__)))
\]

\textbf{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 \( \beta = \beta \_1 \text{; rep}_1 \) and thus we have \( \beta(\text{req}?) = \text{rep}_1(\beta(\text{req}__)) \). Thus it follows that

\[
\text{PreRelabel}[C] \land \text{Abs}[C] \vdash \text{PreRelabelShadow}[C]
\]

\textbf{Correctness Theorem.} \( \text{RelabelShadow} \) must update a shadow policy correctly. Compare the definition of \( \text{rep}_1 \) in schema \( \text{RelabelShadow} \) with \( \beta' \) in \( \text{RelabelPolicy} \). They perform the same function in sense that we have \( \beta' = \beta (\text{rep}_1) \). Thus we have

\[
\text{PreRelabel}[C] \land \text{Abs}[C] \\
\land \text{RelabelShadow}[C] \land \text{Abs}'[C] \vdash \text{Relabel}[C]
\]