Supporting Secure Canonical Upgrade Policies in Multilevel Secure Object Stores*

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

Abstract

Secure canonical upgrade policies are multilevel relabel policies that, under certain conditions, allow high-level subjects to update low-level security labels. This paper describes a scheme whereby these policies can be supported within the Message Filter Model for multilevel secure object-oriented database management systems.

1 Introduction

Dynamic labeling policies support the modification of security labels associated with system entities. For example, in the high-water-mark model [17] an object's security level can rise to reflect the sensitivity of the data written to it. Various security requirements can be captured by augmenting multilevel security with special dynamic labeling rules [5, 11, 14].

Rather than using a fixed set of relabel rules supporting just one type of policy, the mandatory access control model described in [7] allows arbitrary relabel functions to be specified as part of a re-configurable security policy. A variety of security policies have been encoded using this approach, including the Chinese Wall Policy and dynamic segregation of duties [6, 7].

A class of these relabel policies, called *secure canonical upgrade policies* (SCUP), allows high-level subjects to upgrade the security level of low-level objects (upgrade from above). Based on an extended Bell-LaPadula model [1], the underlying access model is multilevel secure [7]. However, it requires a form of security level filtering, whereby a subject at one security level cannot determine any difference between certain other security levels. Given this, a high-level subject may change the security level of a low-level object, and this change is unrecognizable at a low-level.

*This work was supported by Forbairt Basic Research Grant SC/96/611.

This paper investigates how such relabel policies can be supported in the multilevel object store of a message filter [15] based multilevel secure object-oriented database management system (OODBMS). The multilevel object store is composed of single-level stores, one for each security level, and each store containing the objects at the security level of the store. Thus, in simple terms, object relabeling involves migrating an object from one store into another, while preserving referential integrity.

However, the original model for relabeling [7] cannot be directly used in this application. If the security kernel provides the necessary security level filtering, then the result of a high-level user relabeling a low-level object should be the migration of the object from one low-level store to another. But a low-level user, who may not test for a change in the label, can notice a difference in the location of the object. This covert channel could be controlled within the OODBMS itself. This is contrary to the message filter approach, which places the message filter in the trusted computing base, and regards the OODBMS as untrusted.

We solve this problem by revising the model for relabel policies and show how this model can be used to support relabeling in a multilevel object store. We argue that our revised model improves on [7]. It builds directly on a standard Bell-LaPadula model and thus our results are applicable to existing multilevel systems. While the revised model does rely on an additional trusted component to manage security labels, it, unlike [7], does not require modification to the Bell-LaPadula model to support security level filtering.

Section 2 acts as a roadmap for the remainder of the paper. In describing how relabel policies provide a basis for tiered verification, Section 2 also identifies the components of our approach, how they relate, and the necessary verification.

Our proposed approach can use the Bell-LaPadula model as the underlying mandatory access model. Section 3 specifies the revised formalism and necessary properties for relabel policies. The additional trusted component that supports relabel policies is described in Section 4. Section 5 gives a brief outline of the Message Filter model, and Section 6 describes how relabeling can be implemented in multilevel object stores.

The Z notation [16] is used to provide a consistent syntax for structuring and presenting the mathematics in this paper. In using Z, it has been possible to check the mathematics using the Z/EVES tool [13]. Appendix A gives a brief overview of the Z notation.

2 Tiered Verification

The verification of a system supporting relabel policies is done in two tiers [7]. The underlying security kernel, or trusted computing base (TCB), is first verified to be secure according to the multilevel mandatory access model. This tier is extensive and time consuming, possibly forming part of an evaluation process. The second tier involves verifying that the relabel policy can be supported by the kernel. A system that passes the two tiers of verification is considered secure. The advantage of this approach is that re-configuring the relabel policy requires only re-verification of the (easier to reason about) second tier.

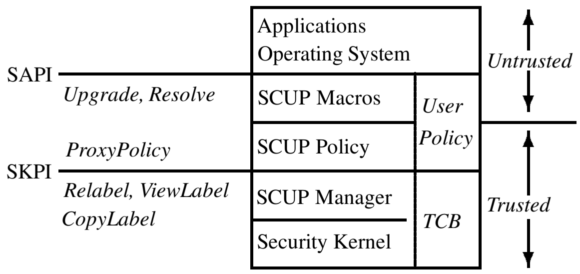


Figure 1. Tiered Verification

Figure 1 gives a conceptual view of tiered verification. In this paper, the first tier corresponds to the verification of a trusted component that manages the relabel policies. This component (SCUP Manager) provides operations for making relabel requests, testing the labels of objects, and so forth. It, together with a security kernel (which we assume is verified) provides a *Security Kernel Programming Interface* (SKPI), on top of which the untrusted operating system and applications are built.

The security policy for a particular enterprise is specified as a relabel policy and must be proven to be a secure canonical upgrade policy. Section 3 gives the properties that such a policy must uphold. Section 6 describes how the requirements for secure relabeling in a multilevel secure object store can be encoded as a SCUP policy. A sample policy for blind updates ($ProxyPolicy$) is specified and verified (second tier) to be a SCUP policy.

The SKPI provides the operating system and applications with a low-level interface to the security policy. The *secure application programming interface* (SAPI), which is constructed from (untrusted) SCUP Macros, provides a more abstract programming interface that is tailored to the particular application. These macros, together with the SCUP policy, clearances for users and so forth, make up the reconfigurable user security policy.

3 Secure Canonical Upgrade Policies

An information flow policy is defined in terms of a set of security labels (L) and a partial ordering ($\_ \leq \_$) of security levels (C).

$$
\begin{array}{|l}
\hline
FlowPolicy[C, L] \\
\_ \leq \_ : C \leftrightarrow C \\
\gamma : L \rightarrow C \\
\hline
\forall\, u, v, w : C \bullet \\
\quad (u \leq u) \land \\
\quad (u \leq v \land v \leq u \Rightarrow u = v) \land \\
\quad (u \leq v \land v \leq w \Rightarrow u \leq w) \\
\hline
\end{array}
$$

Security labels are used to specify security relevant characteristics of the entities in a system. For example, a purchase order object could have a label indicating that it has been requested, but not yet authorized. Security (sensitivity) levels have the usual multilevel interpretation. Given a security label a and security level u, then $\gamma(a)$ gives the security level of an entity with label a. A traditional multilevel flow policy can be viewed as a $FlowPolicy$ with $L = C$ and identity relation γ.

An entity label may change according to relabel functions which form part of the policy.

$$
\begin{array}{|l}
\hline
RelabelPolicy[C, L, F] \\
FlowPolicy[C, L] \\
\mathcal{F_R} : (F \times C \times L) \rightarrow L \\
\hline
\end{array}
$$

Given a set of relabel function identifiers F, then $\mathcal{F_R}(f, s, a) = b$ means that an entity at level s may use function $f \in F$ to change label a to b.

Example 1 Given the datatype definitions

$HLlevs ::= lo \mid hi$
$HLlabs ::= \mathtt{Lo} \mid \mathtt{Mlo} \mid \mathtt{Hi} \mid \mathtt{Invisible}$
$HLfuns ::= mark \mid upgrade$

define a simple flow policy *HLFlow* as

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HLFlow}} \\
\textit{FlowPolicy}[\textit{HLlevs}, \textit{HLlabs}] \\
\hline
\textit{lo} \le \textit{hi} \\
\gamma = \{\texttt{Lo} \mapsto \textit{lo}, \texttt{Mlo} \mapsto \textit{lo}, \texttt{Hi} \mapsto \textit{hi}, \texttt{Invisible} \mapsto \textit{hi}\}
\end{array}
$$

There are *lo* level labels (`Lo` and `Mlo`) and *hi* level label (`Hi`). Label `Invisible` will be considered in the next example. A selective upgrade function (from *lo* to *hi*) is defined as

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HLUpgrade}} \\
\textit{HLFlow};\ \textit{RelabelPolicy}[\textit{HLlevs}, \textit{HLlabs}, \textit{HLfuns}] \\
\hline
\forall\, a : \textit{HLlabs};\ s : \textit{HLlevs} \bullet \\
\quad \textbf{if } (s = \textit{lo} \wedge a = \texttt{Mlo}) \\
\quad\ \textbf{then } (\mathcal{F}_{\mathcal{R}}(\textit{upgrade}, s, a) = \texttt{Hi}) \\
\quad\ \textbf{else } (\textbf{if } (s = \textit{lo} \wedge a = \texttt{Lo}) \\
\quad\quad \textbf{then } (\mathcal{F}_{\mathcal{R}}(\textit{upgrade}, s, a) = \texttt{Invisible}) \\
\quad\quad \textbf{else } (\mathcal{F}_{\mathcal{R}}(\textit{upgrade}, s, a) = a))
\end{array}
$$

If the *lo* level label is `Mlo` then the label becomes `Hi`; if it is `Lo` then it becomes `Invisible`. It is straightforward to define function *mark* to permit a *lo* level entity to mark a `Lo` level label as `Mlo`. △

These relabel policies differ from [7] in that we distinguish between security labels and security levels: in [7] a security label is a security level. It is straightforward to re-specify relabel policies such as the Chinese Wall policy and dynamic segregation of duties [6, 7] using our revised formalism. In this paper we focus on just one class of relabel policy. It is introduced in the next example, but the reader should note that the results in this paper are applicable to any valid SCUP policy.

Example 2 The functions *mark* and *upgrade* permit upgrades that are requested from below (*lo*), and may, therefore, be viewed as secure in the sense that it is possible to build a security mechanism to support such upgrading. Schemes that support, what can be thought of as, upgrade from below policies include [15, 17, 18]. Define an alternative mark (from above) function that permits a *hi* level user to perform *lo* level label marking to be

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HLMark}} \\
\textit{HLFlow};\ \textit{RelabelPolicy}[\textit{HLlevs}, \textit{HLlabs}, \textit{HLfuns}] \\
\hline
\forall\, a : \textit{HLlabs};\ s : \textit{HLlevs} \bullet \\
\quad \textbf{if } (s = \textit{hi} \wedge a = \texttt{Lo}) \\
\quad\ \textbf{then } (\mathcal{F}_{\mathcal{R}}(\textit{mark}, s, a) = \texttt{Mlo}) \\
\quad\ \textbf{else } (\mathcal{F}_{\mathcal{R}}(\textit{mark}, s, a) = a)
\end{array}
$$

If a *lo* level user cannot distinguish between a `Lo` and `Mlo` label then it is possible that this relabel function is secure. △

To determine whether functions such as $HLMark$ are secure it is necessary to define, as part of the policy, how users at different levels view labels. This is defined in terms of label projection, where a user at level v, inspecting label a, actually sees the label $b = (a \upharpoonright v)$. Formally,

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CanonicalPolicy}[C, L, F]} \\
\textit{RelabelPolicy}[C, L, F] \\
\_ \upharpoonright \_ : L \times C \to L \\
\textit{invisible} : L \\
\hline
\forall\, u : C;\ f : F \bullet \\
\quad \textit{invisible} \upharpoonright u = \textit{invisible} \wedge \\
\quad \mathcal{F}_{\mathcal{R}}(f, u, \textit{invisible}) = \textit{invisible}
\end{array}
$$

A canonical policy may be thought of as a relabel policy that has a view-equivalence relation $a \upharpoonright v = b \upharpoonright v$ (in the non-interference sense) defined over its labels. By default, there is a special label that is used to represent label projections that are $\textit{invisible}$.

Example 3 Continuing with the earlier examples, a *Mark for Blind Upgrade* canonical policy is defined as

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HLBlind}} \\
\textit{CanonicalPolicy}[\textit{HLlevs}, \textit{HLlabs}, \textit{HLfuns}] \\
\textit{HLMark};\ \textit{HLUpgrade} \\
\hline
\textit{invisible} = \texttt{Invisible} \\
\forall\, a : \textit{HLlabs} \bullet \\
\quad a \upharpoonright \textit{hi} = a \wedge \\
\quad \textbf{if } (a \in \{\texttt{Lo}, \texttt{Mlo}\}) \\
\quad\ \textbf{then } (a \upharpoonright \textit{lo} = \texttt{Lo}) \\
\quad\ \textbf{else } (a \upharpoonright \textit{lo} = \texttt{Invisible})
\end{array}
$$

A *hi* level user may view all parts of a label, but a low-level user cannot distinguish between `Lo` and `Mlo`, since $\texttt{Lo} \upharpoonright \textit{lo} = \texttt{Mlo} \upharpoonright \textit{lo}$. Note that a `Hi` label is `Invisible` to a *lo* level user. △

There are a number of conditions that a canonical policy must uphold to be secure. These ensure that a high-level user cannot interfere, in a visible way, with low-level labels. Section 4 proposes a subsystem for managing labels that is multilevel secure if the canonical policies that it supports uphold these conditions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CP\_CView}[C, L, F]} \\
\textit{CanonicalPolicy}[C, L, F] \\
\hline
\forall\, v, w : C;\ a, b : L \bullet \\
\quad (a \upharpoonright v = b \upharpoonright v \wedge w \le v \Rightarrow a \upharpoonright w = b \upharpoonright w)
\end{array}
$$

This condition (consistent view) specifies that a user may not test for differences between two labels that are viewed as the same from the projection of the user.

$$\begin{array}{|l}
\hline
CP_NWD[C, L, F] \\
CanonicalPolicy[C, L, F] \\
\hline
\forall f : F;\ s, v : C;\ a : L \bullet \\
\quad \neg\, s \leq v \Rightarrow a \restriction v = \mathcal{F}_{\mathcal{R}}(f, s, a) \restriction v \\
\hline
\end{array}$$

This corresponds to the unwound non-interference requirement that a high-level user may not interfere with a low-level view of a label (no write down).

$$\begin{array}{|l}
\hline
CP_NRU[C, L, F] \\
CanonicalPolicy[C, L, F] \\
\hline
\forall f : F;\ s, v : C;\ a, b : L \bullet \\
\quad a \restriction v = b \restriction v \Rightarrow \mathcal{F}_{\mathcal{R}}(f, s, a) \restriction v = \mathcal{F}_{\mathcal{R}}(f, s, b) \restriction v \\
\hline
\end{array}$$

This corresponds to the unwound non-interference requirement that a change in a low-level view of a label may not depend on any high-level information in the label (no read up). A policy that upholds these three conditions is called a secure canonical upgrade policy (SCUP).

$$\begin{array}{|l}
\hline
SCUP[C, L, F] \\
CP_CView[C, L, F] \\
CP_NWD[C, L, F] \\
CP_NRU[C, L, F] \\
\hline
\end{array}$$

Example 4 It is straightforward to prove that the blind update policy is a secure canonical upgrade policy, that is, $\forall\, HLBlind \bullet SCUP$. △

Example 5 Suppose that details of ships is maintained using labeled objects. The marking of a low-level ship object label, by a high-level user, could be used to indicate that the ship had been destroyed; this aspect of the security characteristics of this object are not known at low. △

Example 6 Suppose that the *HLBlind* policy is to be designed for a flow policy with disjoint high security levels hi_1 and hi_2. In this case we must make sure that the marking of a `Lo` labeled object by security level hi_1 is not visible to level hi_2. If a `Lo` labeled object is marked by both high security levels, then upon upgrading, two copies must be made; one at hi_1, the other at hi_2. This is similar to the strategy proposed in [2] for garbage collecting objects in low-level stores that are referenced from high-level object stores. Section 6 gives a specification for a general blind update policy that supports any information flow ordering.

In [7], labels are represented by the security levels provided by the security kernel, and object duplication due to relabeling is not considered since it is possible to design the flow policy such that this duplication is not necessary. However, representing labels as security levels leads to a very large number of security levels and requires development of a specialized security kernel that provides security level filtering [7]. △

4 SCUP Manager

This section defines a set of operations that provide a trusted interface to a SCUP policy. These trusted operations make up the TCB extension required for standard multilevel secure systems. While we present these operations within the context of our application—an OODBMS—we believe that they are sufficiently general to be applicable to other systems.

Object identifiers are used to uniquely identify objects within an object-oriented database. Following [9], an object identifier is given as a tuple (u, i), where identifier i uniquely identifies an object at a security level u. Thus, given ID, the set of all identifiers, define

$$OID[C] == (C \times ID)$$

Therefore, the security level of every object is encoded within its identifier. Each object $o : OID$ has an associated security label $\delta(o)$.

$$\begin{array}{|l}
\hline
LabelStore[C, L] \\
\delta : OID[C] \rightarrowtail\!\!\!\!\!\!+\ L \\
\hline
\end{array}$$

$LabelStore$ defines the state of the SCUP manager. It is accessed via the operations that make up its programming interface.

The label of an object may be changed according to the relabel functions defined in a SCUP policy. This is done by invoking the $Relabel$ operation.

$$\begin{array}{|l}
\hline
Relabel[C, L, F] \\
SCUP[C, L, F] \\
\Delta LabelStore[C, L] \\
\\
req? : C \\
oid? : OID \\
rfun? : F \\
\hline
\textbf{if}\ (oid? \in \mathrm{dom}\ \delta) \\
\ \textbf{then}\ \delta' = \delta \oplus \{oid? \mapsto \mathcal{F}_{\mathcal{R}}(rfun?, req?, \delta(oid?))\} \\
\ \textbf{else}\ \delta' = \delta \\
\hline
\end{array}$$

For notational convenience, let $Relabel(req?, oid?, rfun?)$ represent the application of relabel function $rfun?$ to the label of object $oid?$, and requested by a user at level $req?$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ViewLabel[C, L, F]} \\
SCUP[C, L, F] \\
\Xi LabelStore[C, L] \\
\\
req? : C \\
oid? : OID \\
lab! : L \\
\hline
\textbf{if } (oid? \in \mathrm{dom}\ \delta \wedge \mathit{first}(oid?) \leq req?) \\
\ \textbf{then } lab! = (\delta\ oid?) \upharpoonright req? \\
\ \textbf{else } lab! = invisible \\
\hline
\end{array}
$$

Operation $ViewLabel(req?, oid?)$ returns, as $lab!$, the appropriate projection of the label of object $oid?$ when requested by a user with level $req?$. Note that if the object does not exist then label $invisible$ is returned; this prevents a low-level user testing the existence of high-level objects.

Since users may view only their projection of an object label, we provide an operation that permits the copying of the underlying value of an object label.

$$
\begin{array}{|l}
\multicolumn{1}{l}{CopyLabel[C, L, F]} \\
SCUP[C, L, F] \\
\Delta LabelStore[C, L] \\
\\
req? : C \\
oid? : OID \\
newid? : ID \\
\hline
\textbf{if } ((\mathit{first}\ oid?) \leq req? \\
\qquad \wedge\ (\neg\ ((req?, newid?) \in \mathrm{dom}\, \delta)) \\
\wedge\ oid? \in \mathrm{dom}\, \delta) \\
\ \textbf{then } \delta' = \delta \cup \{(req?, newid?) \mapsto \delta\ oid?\} \\
\ \textbf{else } \delta' = \delta \\
\hline
\end{array}
$$

Operation $CopyLabel(req?, oid?, newoid?)$, requested by a user at level $req?$, creates a new object $newoid?$ and assigns it the label of object $oid?$. The security level of the new object (given by its identifier) must be the same as the security level of the requester, and must be consistent with it's intended label.

Note that, unlike [7], the SCUP manager may also be used to manage the security labels of subjects. Further operations, such as delete, should be included within the SKPI, but for reasons of space they are not considered here. However, the three operations proposed above provide a sufficient programming interface for supporting SCUP policies in a multilevel object store.

We have proven that this subsystem is multilevel secure in the sense that no high-level user can use the subsystem to interfere with a low-level user's interactions with the subsystem. Appendix B gives details of what has been proven.

5 Message Filtering OODBMS

The message filter model [9] supports multilevel security in object-oriented database systems according to the Bell and LaPadula (BLP) model. If message passing is the only communication possible between objects, then it forms the basis for all information flow. Every object is assigned a security level, and a *message filter* mediates all message passing between objects such that information may flow according to the information flow relation. These database objects are viewed as both objects and subjects in the Bell-LaPadula model. As objects, they have state, and as subjects, they execute actions by sending messages.

Implementation of the message filter model does not rely on the construction of a special trusted OODBMS: if the message filter lies within the TCB of a multilevel system, then the remainder of the application can be based on existing and untrusted OODBMSs. The left-hand side of Figure 2 gives a representation of the message filter model. The (multilevel) persistent object store is partitioned into a collection

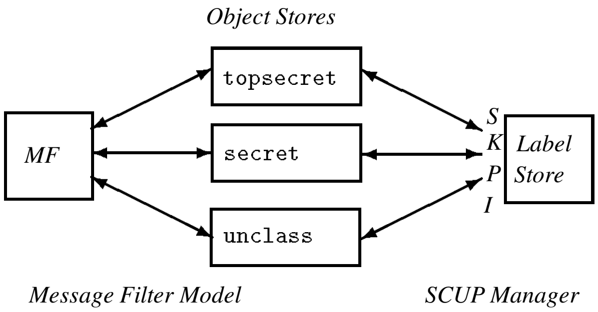


Figure 2. Message Filter & Single Level Stores

of single-level stores. The underlying security kernel, upholding the usual BLP axioms, ensures that it is not possible for an (untrusted) OODBMS to violate the multilevel policy.

This very brief outline of the message filter model is sufficient for the purposes of this paper. To introduce some of the problems that arise with relabeling in object stores, the next example describes how it could be implemented using a multilevel garbage-collection scheme. The example only sketches the approach since a more satisfactory scheme is proposed in Section 6.

Example 7 Consider a multilevel object store and the flow policy *HLFlow*. A simple configuration has objects *B* and *C* in the *lo* level store and a single object *A* in the *hi* level store, with security labels as indicated in Figure 3(a). Suppose that object *B* had been marked by another *lo* level object (mark from below, Example 1). When a request is made at

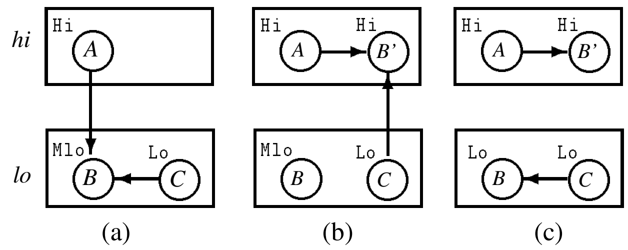


Figure 3. Relabeling and Referential Integrity

lo to upgrade *B*, a copy (B') must be made in the *hi* level store, and references from *A* and *C*, to *B*, must be updated to preserve referential integrity (Figure 3(b)).

However, if *A* had been marked by a request at security level *hi* (Example 2), then the new configuration in Figure 3(b) would lead to a covert channel. Object *C* can test its reference to *B* and a noticeable change in the pointer from a *lo* level reference to a *hi* level reference could be controlled by *hi*. To prevent this channel, the *lo* level references must not be updated (Figure 3(c)).

It is possible to view relabeling as a class of multilevel garbage collection problem. In [2] the authors propose a garbage collection scheme for a message-filter-based multilevel object store that is based on a minimal number of trusted components. A garbage collector running at *lo* retrieves (garbage collects) objects that are not referenced by any other *lo* level object in the store. The *lo* level collector is not permitted to know of *hi* level references to such objects. The essence of the scheme in [2] is that another *hi* level collector copies these *lo* level objects into the *hi* level store (appropriately updating *hi* level references), before the original object is retrieved by the *lo* level garbage collector.

This scheme can be used to carry out relabeling if the criteria for garbage collecting is modified. The SCUP manager maintains individual object labels via the operations described in Section 4. After a series of *Relabel* requests to the SCUP manager, labels may be out of alignment with their object's security level, that is $\mathit{first}(o) = \gamma(\delta(o))$ no longer holds. A low-level object that is out of alignment (tested using *ViewLabel* and γ) should be regarded as an object that is to be retrieved. The same test is used by the high-level collector to check if a copy of the object should be made at this high level (object label aligns at this security level), and references appropriately updated.

Realignment must occur periodically, and, as with garbage collection, must scan the entire database, aligning objects to their labels. The advantage of this relabeling by realignment scheme is that since it is based on the multilevel garbage collection scheme it requires a minimal number of trusted components (in addition to the SCUP manager). However, there are disadvantages that makes it impractical. Relabeling does not occur at the time of the request. Furthermore, it is not practical to frequently run the realignment process and, therefore, objects will be regularly out of alignment. $\triangle$

6 Proxy Enabled Blind Update Policy

Relabeling of objects can be viewed as an object migration problem, where an object must be moved from one single level object store to another, at a different security level. This migration must be done so that it is multilevel secure and referential integrity is preserved. In this section we propose a method of multilevel secure relabeling based on data migration and, given the SCUP manager, can be built from untrusted components.

Proxy objects [3, 4, 10] can be used to support object migration between object stores. Proxies provide a level of indirection for references to objects in remote object stores and are used in systems such as Distributed Smalltalk [3] and SOS [10]. In addition to migration, they have a variety of uses, including remote invocation, physical location transparency and reuse.

Example 8 Figure 4(a) shows two object stores with objects *A*, *B* and *C*. Suppose that object *B* is to be migrated from the *lo* level store to the *hi* level store. Figure 4(b) shows the result of this migration. A copy of *B* is moved to the

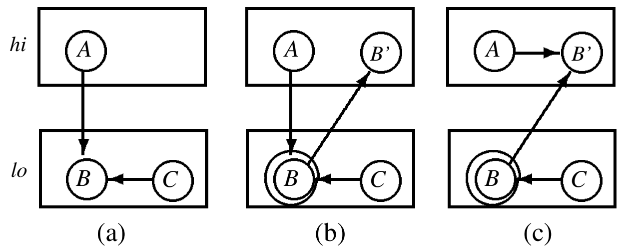


Figure 4. Object Migration using Proxies

hi level store as B', and *B* becomes a proxy object for B'. When a message is passed to *B*, the proxy forwards the message on to its current copy B'[1]. If desired, references can be updated when proxies are referenced, for example, having communicated with proxy *B*, object *A* could update its reference to B' (Figure 4(c)). $\triangle$

If proxies are used to support migration due to relabeling then it must not be possible to use a proxy to violate

[1]Proxy objects provide physical location transparency. For complete transparency in our example we should include a proxy for *A*'s reference to *B*.

multilevel security. Studying Figure 4(b), while the hi level object A should be allowed to follow the proxy to B', if the migration was due to a mark requested from above (Example 2), then object C should not be able use the proxy to test for the existence of B'. Similar mediation must done on references from proxies where the relabeling/migration resulted in the instantiation of a number of copies of the object at different security levels (Example 6).

Therefore, it is necessary to associate security characteristics with proxy objects and, in turn, control access to objects based on these characteristics. This can be done by encoding the requirements into a SCUP policy; the SCUP manager and existing MLS security kernel will provide the necessary mediation. The reader should note that while the policy presented here is specified for blind updates only, the scheme can form the basis of any consistent SCUP policy.

Let $Level$ represent the set of all security levels of interest. We define the set of security labels to be the datatype

$$
\begin{aligned}
Label ::= &\ \mathtt{Invisible} \\
| &\ \mathtt{Obj}\langle\!\langle Level \times \mathbb{P}\ Level \rangle\!\rangle \\
| &\ \mathtt{Prxy}\langle\!\langle Level \times \mathbb{P}\ Level \times (Level \twoheadrightarrow ID) \rangle\!\rangle
\end{aligned}
$$

A label $\mathtt{Obj}(u, M)$ represents an object at security level u that has been marked by objects at levels in M. The label $\mathtt{Prxy}(u, M, P)$ represents an object at security level u, that was marked by objects at security levels in M, and is acting as a proxy to the set of objects with object identifiers in P. Recall that an object identifier is of the form (v, i), where i uniquely identifies the object within the object store at security level v.

Example 9 Consider Example 8. If object B has initial label $\mathtt{Obj}(lo, \{\})$ then Figure 5(a) gives the configuration resulting from a hi level object making request $Relabel(hi, B, \mathtt{mark})$ to the SCUP manager. The upgrading of B requires a number of SCUP manager invocations and these are illustrated in Figure 5 (b) to (f). Each of these relabel functions will be explained as they are defined below. $\triangle$

Define the relabel function identifiers to be

$$
\begin{aligned}
Fid ::= &\ \mathtt{mrk} \mid \mathtt{unmrk} \mid \mathtt{mkprxy} \\
| &\ \mathtt{ref}\langle\!\langle OID[Level] \rangle\!\rangle \mid \mathtt{up}\langle\!\langle Level \rangle\!\rangle
\end{aligned}
$$

LabelLevels

$$
\begin{array}{|l}
CanonicalPolicy[Level, Label, Fid] \\
\hline
\forall\, u : Level;\ M : \mathbb{P}\ Level;\ P : Level \twoheadrightarrow ID \bullet \\
\quad \gamma(\mathtt{Obj}(u, M)) = u \wedge \\
\quad \gamma(\mathtt{Prxy}(u, M, P)) = u
\end{array}
$$

It is not necessary to define any particular security level ordering $(\_ \leq \_)$—it can be specified at a later stage.

LabelProjection

$$
\begin{array}{|l}
CanonicalPolicy[Level, Label, Fid] \\
bnd : Level \rightarrow \mathbb{P}\ Level \\
\hline
invisible = \mathtt{Invisible} \\
\forall\, x : Level \bullet bnd(x) = \{y : Level \mid y \leq x\} \\
\forall\, u, v : Level;\ M : \mathbb{P}\ Level;\ P : Level \twoheadrightarrow ID \bullet \\
\quad (\mathbf{if}\ (u \leq v)\ \mathbf{then} \\
\qquad (\mathtt{Obj}(u, M) \upharpoonright v = \mathtt{Obj}(u, M \cap bnd(v)) \wedge \\
\qquad \mathtt{Prxy}(u, M, P) \upharpoonright v \\
\qquad\qquad = \mathtt{Prxy}(u, M \cap bnd(v), bnd(v) \lhd P)) \\
\quad \mathbf{else} \\
\qquad (\mathtt{Obj}(u, M) \upharpoonright v = \mathtt{Invisible} \wedge \\
\qquad \mathtt{Prxy}(u, M, P) \upharpoonright v = \mathtt{Invisible}))
\end{array}
$$

A user at class v views a label as invisible if that label cannot be instantiated at a security level dominated by v. If an object is visible to a user at class v then the user may view only those marked levels and proxy references that it dominates. For example, we have $\mathtt{Obj}(lo, \{hi\}) \upharpoonright lo = \mathtt{Obj}(lo, \{\}) \upharpoonright lo$ and $\mathtt{Prxy}(lo, \{hi\}, \{(hi, B')\}) \upharpoonright lo = \mathtt{Prxy}(lo, \{\}, \{\})$.

Mark

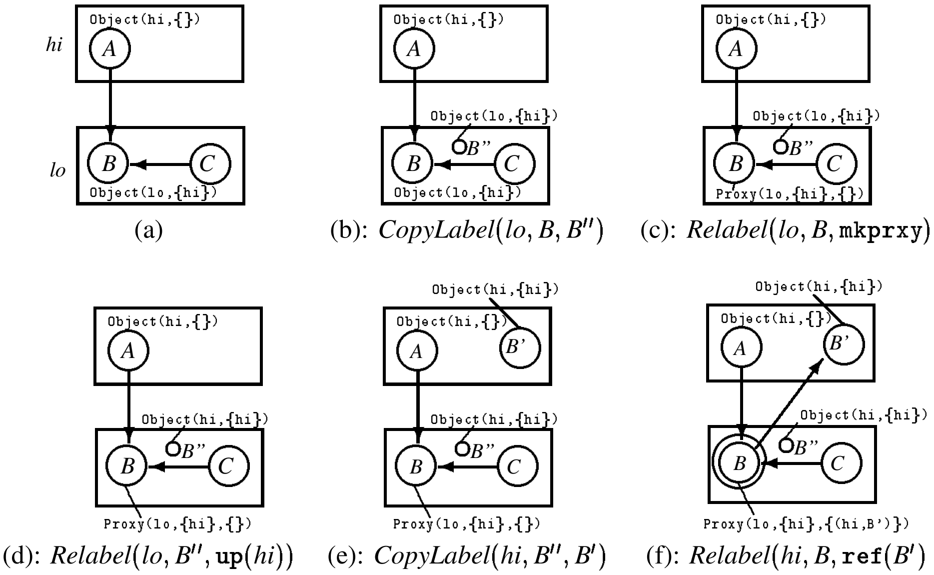
$$
\begin{array}{|l}
LabelLevels;\ LabelProjection \\
\hline
\forall\, u, s : Level;\ M : \mathbb{P}\ Level;\ P : Level \twoheadrightarrow ID \bullet \\
\quad \mathcal{F}_{\mathcal{R}}(\mathtt{mrk}, s, \mathtt{Prxy}(u, M, P)) = \mathtt{Prxy}(u, M, P) \wedge \\
\quad \mathbf{if}\ (u \leq s \wedge u \neq s) \\
\quad\ \mathbf{then}\ (\mathcal{F}_{\mathcal{R}}(\mathtt{mrk}, s, \mathtt{Obj}(u, M)) = \mathtt{Obj}(u, M \cup \{s\})) \\
\quad\ \mathbf{else}\ (\mathcal{F}_{\mathcal{R}}(\mathtt{mrk}, s, \mathtt{Obj}(u, M)) = \mathtt{Obj}(u, M))
\end{array}
$$

The $\mathtt{mrk}$ function may be used by any user whose security level strictly dominates the security level of the object.

UnMark

$$
\begin{array}{|l}
LabelLevels;\ LabelProjection \\
\hline
\forall\, u, s : Level;\ M : \mathbb{P}\ Level;\ P : Level \twoheadrightarrow ID \bullet \\
\quad \mathcal{F}_{\mathcal{R}}(\mathtt{unmrk}, s, \mathtt{Prxy}(u, M, P)) = \mathtt{Prxy}(u, M, P) \wedge \\
\quad \mathbf{if}\ (u \leq s) \\
\quad\ \mathbf{then}\ (\mathcal{F}_{\mathcal{R}}(\mathtt{unmrk}, s, \mathtt{Obj}(u, M)) = \mathtt{Obj}(u, M \setminus \{s\})) \\
\quad\ \mathbf{else}\ (\mathcal{F}_{\mathcal{R}}(\mathtt{unmrk}, s, \mathtt{Obj}(u, M)) = \mathtt{Obj}(u, M))
\end{array}
$$

The $\mathtt{unmrk}$ function may be used to reverse the result of a $\mathtt{mrk}$ request.



(a) (b): $CopyLabel(lo, B, B'')$ (c): $Relabel(lo, B, \texttt{mkprxy})$

(d): $Relabel(lo, B'', \texttt{up}(hi))$ (e): $CopyLabel(hi, B'', B')$ (f): $Relabel(hi, B, \texttt{ref}(B'))$

Figure 5. Relabeling using Proxy Labels

MkProxy

$LabelLevels;\ LabelProjection$

$$\forall\, u, s : Level;\ M : \mathbb{P}\, Level;\ P : Level \twoheadrightarrow ID \bullet$$
$$\mathcal{F}_{\mathcal{R}}(\texttt{mkprxy}, s, \texttt{Prxy}(u, M, P)) = \texttt{Prxy}(u, M, P) \wedge$$
$$\textbf{if } (s = u)$$
$$\textbf{then } (\mathcal{F}_{\mathcal{R}}(\texttt{mkprxy}, s, \texttt{Obj}(u, M)) = \texttt{Prxy}(u, M, \{\}))$$
$$\textbf{else } (\mathcal{F}_{\mathcal{R}}(\texttt{mkprxy}, s, \texttt{Obj}(u, M)) = \texttt{Obj}(u, M))$$

When an object is upgraded it first becomes a proxy object. The relabel is requested at the security level of the object, and thus the set of proxy references is empty since, it is not possible, at this this point, to specify the set of objects for which it will act as a proxy (the references are high-level data). Figure 5(c) illustrates this step in upgrading. Note that before applying function `mkprxy`, a copy of the original object label is made (Figure 5(b)).

To avoid excessive object duplication an object is upgraded to the lowest bounds of the security levels by which it was marked, rather than to all levels by which it was marked. Thus, if an unclassified object was marked by both secret and topsecret, then when upgraded by unclassified it becomes a secret object (marked by topsecret).

Once an object o has been relabeled as a proxy object, copies of it must be made at each security level s at which it might have been marked. It is not possible for the marked object to be aware of the levels by which it has been marked. Therefore, each copy must be created at the security level s of the new object. This creation is done by the OODBMS, which also generates a suitable identifier for the object.

Upgrade

$LabelLevels;\ LabelProjection$

$$\forall\, u, v, s : Level;\ M : \mathbb{P}\, Level;\ P : Level \twoheadrightarrow ID \bullet$$
$$\mathcal{F}_{\mathcal{R}}(\texttt{up}(v), s, \texttt{Prxy}(u, M, P)) = \texttt{Prxy}(u, M, P) \wedge$$
$$\textbf{if } ((s = u) \wedge (u \leq v))$$
$$\textbf{then } (\mathcal{F}_{\mathcal{R}}(\texttt{up}(v), s, \texttt{Obj}(u, M)) = \texttt{Obj}(v, \operatorname{ran}(\{v\} \lhd (\_ \leq \_) \rhd M)))$$
$$\textbf{else } (\mathcal{F}_{\mathcal{R}}(\texttt{up}(v), s, \texttt{Obj}(u, M)) = \texttt{Obj}(u, M))$$

If the original label of the object was $\texttt{Obj}(u, M)$, then function $\texttt{up}(v)$ upgrades the label so that its level v corresponds to the level of the new copy. Note that, in order to prove this function secure, this relabeling request must be made at the level of the original object (u), while the copying (if it occurs) must be requested at the level of its copy. Figure 5(d) and Figure 5(e) illustrate this migration of the original object. As noted above, this relabeling and copying must be done for every level—the temporary objects that are used to give the labels for these copies (B'' in Figure 5(d)) can be garbage-collected at a later stage.

$$
\begin{array}{l}
\textit{Ref} \\
\hline
\textit{LabelLevels};\ \textit{LabelProjection} \\
\hline
\forall\, o : \textit{OID}[\textit{Level}];\ u, s : \textit{Level}; \\
\quad M : \mathbb{P}\ \textit{Level};\ P : \textit{Level} \nrightarrow \textit{ID} \bullet \\
\quad \mathcal{F}_{\mathcal{R}}(\mathtt{ref}(o), s, \mathtt{Obj}(u, M)) = \mathtt{Obj}(u, M) \land \\
\quad \textbf{if } ((u \leq s) \land (s \in M) \land (\textit{first}(o) = s)) \\
\quad\ \textbf{then } (\mathcal{F}_{\mathcal{R}}(\mathtt{ref}(o), s, \mathtt{Prxy}(u, M, P)) \\
\qquad\qquad\qquad = \mathtt{Prxy}(u, M, P \oplus \{o\})) \\
\quad\ \textbf{else } (\mathcal{F}_{\mathcal{R}}(\mathtt{ref}(o), s, \mathtt{Prxy}(u, M, P)) \\
\qquad\qquad\qquad = \mathtt{Prxy}(u, M, P))
\end{array}
$$

If the identifier of this object is o' then relabel function $\mathtt{ref}(o')$ is used to update the label of the proxy of the original object o (an update requested from above). For example, in Figure 5(e) the copy B' is created at level hi. Its (hi) object-identifier is then used to update the references in the proxy label of B (Figure 5(f)).

The complete proxy based blind update policy may be specified as

$$
\begin{array}{l}
\textit{ProxyPolicy} \mathrel{\widehat{=}} \textit{Mark} \land \textit{UnMark} \land \textit{MkProxy} \\
\qquad\qquad\qquad \land\ \textit{Upgrade} \land \textit{Ref}
\end{array}
$$

and it has been proven that

$$\forall\, \textit{ProxyPolicy} \bullet \textit{SCUP}[\textit{Level}, \textit{Label}, \textit{Fid}]$$

The operation of marking an object is done by invoking operation $\textit{Relabel}$. However, the upgrade operation is implemented by a series of relabels performed at different security levels. Relabel macro $\textit{Upgrade}$ implements these relabels

```
Upgrade(s : Level, o : OID){
      a ← ViewLabel(s, o);
      if (a = Obj(s, M)) then {
        ol ← NewOid();
        CopyLabel(s, o, ol);
        Relabel(s, o, mkprxy);
        Migrate(s, o, ol);
      }
}
```

If $\textit{Upgrade}$ is regarded as an operation of the (untrusted) object manager, and there is a copy of it available at each security level, then it need not form part of the TCB. An object, whose current security level ($\textit{rlevel}$ in [9]) is given as s, invokes (message passing) $\textit{Upgrade}(s, o)$ to perform an upgrade on object o. Invocation of this operation corresponds to an untrusted subject running at security level s, making requests to the TCB (SCUP manager).

At security level s, operation $\textit{Upgrade}$ has no way of determining what copies are to be made: invoking $\textit{ViewLabel}(s, o)$ will return only the view that a user at security level s may have on the object's label. $\textit{Upgrade}(s, o)$ invokes (message passing) $\textit{Migrate}$.

```
Migrate(s : Level; po, ol : OID){
      a ← ViewLabel(s, ol);
      if (a = Obj(s, M) ∧ s ∈ M) then {
        oc ← CopyObject(po);
        CopyLabel(s, ol, oc);
        Relabel(s, oc, unmark(s));
        Relabel(s, po, ref(oc));
      }else
        for each v ∈ mins{u : Level | s < u} then {
           ol' ← NewOid();
           CopyLabel(s, ol, ol');
           Relabel(s, ol', up(v));
           Migrate(v, po, ol');
        }
}
```

Operation $\textit{Migrate}(s, po, ol)$ determines whether or not a copy of the object po should be migrated to the level s. If migration is required, then the label for the copy of po is provided by ol. The object po should be migrated to level s if the label of ol includes a marking by level s.

If migration is not required at this level then it may be required at levels that dominate s, that is, ol may be marked at these levels. But there is no way of testing this by requesting $\textit{ViewLabel}$ at level s. Thus, $\textit{Migrate}(v, po, ol')$ is recursively called for each level that is a lowest bound on the levels that strictly dominate s.

Operation $\textit{Migrate}$ may also be regarded as untrusted. Under the message filter model, when $\textit{Migrate}(s, po, ol)$ passes a message with a request to invoke $\textit{Migrate}(v, po, ol')$, where v strictly dominates s, the return message ($\textit{null}$) is sent immediately. This prevents downward signalling channels that might arise from higher level objects varying their response time.

When a message is sent to an object that has a proxy label, the object manager must determine whether the message is to be forwarded or not. This is achieved by inspecting the object's label (at the security level of the requester). If the proxy reference is empty, then this is the object, otherwise a proxy reference must be followed.

```
OID Resolve(s : Level, o : OID){
      a ← ViewLabel(s, o);
      if (a = Prxy(u, M, P) ∧ P ≠ ∅)
      then return(Resolve(s, (μ x : P • x))) ;
      else return(o);
}
```

Example 10 The SCUP policy *ProxyPolicy* can be used with any underlying flow policy. Consider the ordering with security levels x, y, z, and w, where $x < y < w$, $x < z < w$, with disjoint y and z. An object A at security level x that was marked by security levels y, z, and w, has label $\mathtt{Obj}(x, \{y, z, w\})$. An object at security level y referencing A has no way of determining whether it has been marked by disjoint z or higher w, since $\mathit{Viewlabel}(y, A)$ returns $\mathtt{Obj}(x, \{y\})$.

Operation $\mathit{Upgrade}(x, A)$ gives object A a proxy label $\mathtt{Prxy}(x, \{w\}, \{(y, A'), (z, A'')\})$, with references to copies A' at level y and A'' at security level z. These objects inherit the marking from w: the label for A' is $\mathtt{Obj}(y, \{w\})$, so that a subsequent upgrade requested at y on A' will upgrade it to w. A reference to A at security level y will be resolved to $\mathit{Resolve}(y, A) = A'$, while $\mathit{Resolve}(z, A) = A''$. A reference to A by a w security level object can be resolved to either A' or A'', it does not matter, as far as security is concerned, which is used. $\triangle$

A number of variations may be made to $\mathit{ProxyPolicy}$. For example, it is straightforward to also support the marking of an object at levels that are dominated by the object's security level (mark requested from below). In this case, when the object is upgraded, then it is possible for low-level references to proxies to be resolved to high-level references if the marking was originally requested from the low-level.

7 Conclusion

Tiered verification spreads the verification of security across two tiers. The first tier corresponds to the verification of the system's trusted computing base. The second tier involves the verification of the application or site-specific security policy. Modification of the application security policy requires verification of just the easier to reason about second tier.

Relabel policies are used to provide the basis for tiered verification and the SCUP manager specified in Section 4 gives the additional TCB component required to support these policies. In this paper we focused on a particular class of policy, a flexible relabel policy for multilevel object stores. It is interesting to note that the security requirements for this application are captured, in their entirety, as the relabel policy $\mathit{ProxyPolicy}$. Other relabel policies, such as those in [7] can be similarly adapted and enforced in a multilevel object store.

A dual perspective may be taken on an object. An object has a state which encodes its functional characteristics. An object also has a security label (state), which encodes its corresponding security characteristics. Methods are used to update the object state, while relabel functions update the security state. Given this, method invocation may be interpreted as message passing to the object concerned plus a relabeling request. Designing object-oriented application systems where functionality and security is partitioned in this way is a topic worth future investigation.

The abstract SCUP manager specified in Section 4 has been verified multilevel secure. This allows us to argue that our overall approach is sound. Further research must be done on the practical implementation and verification of this manager. We are currently exploring a kernelized approach, where we view the SCUP manager as a separate database system and subset it into a number of untrusted single level components. We hope to report on this work at a future date.

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] E. Bertino, L. Mancini, and S. Jajodia. Collecting garbage in multilevel secure object stores. In *Proceedings of the Symposium on Security and Privacy*, pages 106–120, Oakland, CA, May 1994. IEEE Computer Society Press.

[3] A. Corradi, L. Leonardi, and M. Zannini. Distributed environments based on objects: Upgrading smalltalk toward distribution. In *Ninth International Phoenix Conference on Computers and Communications*, 1990.

[4] D. Decouchant. A distributed object manager for the smalltalk-80 system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 487–520. ACM Press, 1989.

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

[6] S. Foley. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *4th ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[7] 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.

[8] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[9] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 1990. IEEE Computer Society Press.

[10] M. Makpangou and M. Shapiro. The SOS object-oriented communication service. In *Ninth International Conference on Computer Communications*, Tel Aviv, Israel, 1988.

[11] C. Meadows. Extending the Brewer Nash model to a multilevel context. In *Proceedings of the Symposium on Security and Privacy*, pages 95–102, Oakland, CA, 1990. IEEE Computer Society Press.

[12] J. Rushby. Noninterference, transitivity and channel-control security policies. Technical Report SRI-CSL-92-02, SRI International, Menlo Park, CA., Dec. 1992.

[13] M. Saaltink. The Z/EVES system. In *ZUM'97 (10th International Conference of Z Users)*, pages 72–85. Springer Verlag LNCS 1212, 1997.

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

[15] R. Sandhu and S. Jajodia. Honest databases that can keep secrets. In *Proceedings of the 14th National Computer Security Conference*, Washington, DC, 1991.

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

[17] C. Weissman. Security controls in the ADEPT-50 time-sharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133. FJCC, 1969.

[18] S. Wiseman. On the problem of security in databases. In *Database Security III: Status and Prospects*. Springer, 1989.

A 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.

The Schema notation is used to structure specifications in Z. A schema such as *FlowPolicy* defines a collection of variables (limited to the scope of the schema), and specifies how they are related. Schema $FlowPolicy[C, L, F]$ is defined in terms of generic types $[C, L, F]$, which must be instantiated when the schema is used. Schemas may be defined in terms of other schemas. For example, the inclusion of *FlowPolicy* within schema *RelabelPolicy* is equivalent to the syntactic inclusion of the variables and predicates of *FlowPolicy* within *RelabelPolicy*. Schemas may be composed using logical operators. For example, $HLFlow \wedge RelabelPolicy$ is a schema with variables and predicates from both *HLFlow* and *RelabelPolicy*. Schema predicates are useful for writing theorems: in Example 2 the notation $\forall HLBind \bullet SCUP$ is a universal quantification over all the variables of *HLBind* such that the predicate part of *HLBind* implies the predicate part of schema *SCUP*.

The decorated schema $LabelStore'$ is *LabelStore* with all variables primed. The schema $\Delta LabelStore$ is a syntactic sugar for $LabelStore \wedge LabelStore'$. It is typically used for specifying state transitions, with undecorated variables representing 'before values' and decorated (primed) variables representing 'after values'. $\theta SCUP$ gives a schema type with variables from schema *SCUP*. Schema $\Xi SCUP$ is the schema $\Delta SCUP$, but with the constraint that variable values are unchanged, that is, $\theta SCUP = \theta SCUP'$.

$first(a, b)$	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
$A \leftrightarrow B$	Relations between A and B
$A \rightarrow B$	Total functions from A to B
$A \nrightarrow B$	Partial functions in $A \rightarrow B$
dom R, ran R	Domain and Range 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
$R \rhd A$	Relation R with its range restricted to values from A
$\mu\, x : S \bullet x$	a value from non-empty set S.

B Security Analysis of SCUP Manager: Summary

In this appendix we give a summary of a non-interference analysis [8, 12] of the SCUP Manager. In particular, we use an unwound version of non-interference, noting that the manager operations are deterministic and input-total.

State *LabelStore* looks the same as state $LabelStore'$, when viewed from security level vl, when the label projections of the objects, whose levels are dominated by vl, are equal. This is formally specified as follows.

$$
\begin{array}{|l}
\hline
VEquiv[C, L, F] \\
LabelStore[C, L] \\
LabelStore'[C, L] \\
SCUP[C, L, F] \\
vl : C \\
\hline
\forall o : OID[C] \mid first(o) \leq vl \bullet \\
\quad (o \in \mathrm{dom}(\delta) \Leftrightarrow o \in \mathrm{dom}(\delta')) \wedge \\
\quad (o \in \mathrm{dom}\,\delta \cap \mathrm{dom}\,\delta' \Rightarrow \delta(o) \upharpoonright vl = \delta'(o) \upharpoonright vl) \\
\hline
\end{array}
$$

Given this relationship, the first unwinding condition requires that each operation, requested at a high-level, cannot interfere with a low-level view of the state. This corresponds to a no-write-up (NWD) rule and three theorems have been proven.

Theorem Relabel_NWD $[C, L, F]$:
$\forall Relabel[C, L, F];\ vl : C \bullet$
$\quad \neg\,(req? \leq vl) \Rightarrow VEquiv[C, L, F]$

Theorem ViewLabel_NWD $[C, L, F]$:
$\forall\ ViewLabel[C, L, F];\ vl : C \bullet$
$\quad \neg\ (req? \leq vl) \Rightarrow VEquiv[C, L, F]$

Theorem CopyLabel_NWD $[C, L, F]$:
$\forall\ CopyLabel[C, L, F];\ vl : C \bullet$
$\quad \neg\ (req? \leq vl) \Rightarrow VEquiv[C, L, F]$

The second unwinding condition requires that the outcome of an operation, requested at a low-level, cannot be based in any way on the high-level part of the state.

Theorem Relabel_NRU $[C, L, F]$:
$\forall\ Relabel[C, L, F];$
$\quad Relabel[C, L, F][\delta''/\delta, \delta'''/\delta'];\ vl : C$
$\quad \bullet\ VEquiv[C, L, F][\delta''/\delta'] \Rightarrow VEquiv[C, L, F][\delta'''/\delta]$

Theorem ViewLabel_NRU $[C, L, F]$:
$\forall\ ViewLabel[C, L, F];$
$\quad ViewLabel[C, L, F][\delta''/\delta, \delta'''/\delta', lab!''/lab!];\ vl : C$
$\quad \bullet\ (VEquiv[C, L, F][\delta''/\delta'] \wedge (req? \leq vl))$
$\quad\quad \Rightarrow lab! = lab!''$

Theorem CopyLabel_NRU $[C, L, F]$:
$\forall\ CopyLabel[C, L, F];$
$\quad CopyLabel[C, L, F][\delta''/\delta, \delta'''/\delta'];\ vl : C$
$\quad \bullet\ VEquiv[C, L, F][\delta''/\delta']$
$\quad\quad \Rightarrow VEquiv[C, L, F][\delta'/\delta, \delta'''/\delta']$

C Theorem Proving with Z/EVES

This paper was typeset using LaTeX with the fuzz and z-eves styles. Thus, the LaTeX source of the paper acts as the input specification to the Z/EVES system [13]. In addition to using the system to syntax-, type- and domain-check the specifications in this paper, Z/EVES was also used for the security analysis of the SCUP Manager and verification of *ProxyPolicy*. The specification source, along with the Z/EVES proof scripts for all theorems are available from the author or under the author's home page at `<URL:http://www.cs.ucc.ie/sfoley.html>`.