European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 1998. Springer.

A Kernelized Architecture for Multilevel Secure Application Policies

Simon N. Foley[1,2]

[1] University of Cambridge CCSR, Cambridge CB2 3DS, UK.
[2] Department of Computer Science, University College, Cork, Ireland.
s.foley@cs.ucc.ie

Abstract. Mandatory label-based policies may be used to support a wide-range of application security requirements. Labels encode the security state of system entities and the security policy specifies how these labels may change. Building on previous results, this paper develops a model for a kernelized framework for supporting these policies. The framework provides the basis for, what is essentially, an interpreter of multilevel programs: programs that manipulate multilevel label data-structures. This enables application functionality and security concerns to be developed separately, bringing with it the advantages of a separation of concerns paradigm.

1 Introduction

Conventional multilevel secure systems place all trust in the underlying trusted computing base (TCB), regarding most of the operating system and applications software as untrusted [20]. Clark and Wilson [3] argue that security should be defined across both the operating system and applications. Under their model, a secure system may be viewed as a certified application running on top of a trusted computing base (TCB). Certifying an application is analogous to arguing (to a degree) it's correctness according to the application's security requirements; the TCB is expected to have undergone some sort of security evaluation.

A problem with this approach is that it encourages an intertwining of the functional and security-critical code that makes up an application. For example, interpreting the original Clark-Wilson model, the TCB is expected to support (enforce) static segregation of duties. However it appears that dynamic segregation of duty must be implemented/intertwined within the application itself [4, 12]. Intertwining makes the application complex and reasoning more difficult since both security and functionality concerns have to be dealt with at the same level of abstraction.

We argue that when developing an application the security and functionality concerns of the application must be separated, both at the conceptual level and at the implementation level. At the conceptual level, a loose coupling between concerns facilitates the verification of security, while at the implementation level we want an architecture and programming environment that will support the concerns securely.

As a software engineering paradigm, techniques that support separation of concerns for synchronisation, real-time constraints, failures and others have been studied. The reader is referred to [7] for an introduction. One could view policy neutral architectures such as [13, 17] as a step in the direction of providing support for the separation of (security) concerns paradigm.

We are interested in supporting separation for multilevel security concerns. In this case, an application developer designs and implements an application in terms of separate security and functionality components. Once the security concerns meet certain verification conditions, an underlying TCB is expected to ensure multilevel security.

The dynamic label-based policies proposed in [4, 5] can be used to support this separation. From an object-oriented perspective, labels are used to encode security relevant characteristics of objects and provide relabel functions that specify how the labels may change. Given some functional object, we then argue that it also has an associated security object whose state corresponds to its label and has methods that define how its state (label) may change. This security label object is like a meta-object, providing security details about the information in the associated object.

These label-based policies use the Bell-LaPadula model as their underlying access control model and require a special trusted label manager which mediates and interprets requests to update labels. A variety of application security policies can be encoded in terms of label-based policies, including Chinese walls, dynamic segregation of duties and group access policies [4, 5].

It is desirable to have a framework that can support a wide range of label-based policies. In [5] a model is described that allows *any* subject to relabel *any* object label, where the relabel functions meet certain security non-interference style requirements. These *secure canonical upgrade policies* (SCUP) are supported securely so long as high-level changes may not be detected by low-level subjects. A framework that supports such policies in a message-filter [8] based multilevel secure object store is described in [5]. However, the framework in [5] does not consider how the label manager might be developed in practice. This paper develops a kernelized-based model of the label manager.

The paper is organised as follows. Sections 2 and 3 describe the basic model for relabel policies. They are based on [5], but with some minor modifications that make kernelization possible. Sections 4, 5 and 6 give the noninterference analysis, kernelization, and correctness, respectively, of the label manager. A number of state-invariants for the kernelized manager are explored in Section 7; these would contribute towards an efficient implementation. Section 8 discusses the results and contributions of the paper.

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 [15]. Appendix A gives a brief overview of the Z notation used in the paper.

2 Dynamic Label Policies

An information flow policy is defined in terms of a partial ordering ($\_ \leq \_$) of security levels (C) which have the usual multilevel interpretation.

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

Security labels (L) are datatypes that are used to encode security relevant characteristics of objects. For example, a purchase order object could have a label indicating that it has been requested, but not yet authorised. In [5] the security label is also used to specify the desired security level of an object. An object's label may change according to relabel functions which form part of the policy.

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

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

Example 1 A simple *mark-for-upgrade* policy uses security labels to encode details about future upgrade levels for an object. An object label has a format u`:-`A, where u indicates it's current level and A gives a list of future upgrade levels. For example, the label (`u:-[u,t]`) of an unclassified (`u`) object indicates that it should be upgraded to top-secret (`t:-[t]`) when an upgrade is requested.

A requester at level `r` invokes the function `mark` to tag the label with its level (`r`). Function `up` upgrades the object's label u`:-`A to the next level in A. For example, the label (`u:-[u,t]`) becomes (`t:-[t]`), when `up(u,(u:-[u,t]))` is invoked. For the purpose of illustration, Figure 1 gives a fragment of a (Haskell) functional-style prototype of this relabel policy (`project` and `Invisible` will be described later)[1]. Functions are specified in an equational-style and expression `[v|v<-tags, w<=v]` is the list of all `v` that are members of list `tags` and that dominate `w`. $\triangle$

To determine whether the relabel functions `mark` and `up` are secure it is necessary to first 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 $b = (a \restriction v)$. A canonical policy may be

[1] This policy is valid only for flow policies that are total orders. A specification of a more general policy can be found in [5].

```
data Label level = Invisible | level:-[level] --datatype for label

mark(req,(u:-tags))         --level req request marks label (u:-tags)
    | (u<req)               = u:-(req:tags)  --hi may mark lo label
    | otherwise             = u:-tags        --no change in label

up(req,(u:-tags))           --level req request upgrades (u:-tags)
    | req<u && newtags==[]  = (u:-tags)
    | otherwise             = (minimum newtags):-newtags
    where newtags           = [v |v<-tags,u<v]

project((u:-tags),v)        --label (u:-tags), as seen from level v
    | u <= v                = (u:-[w|w<-tags, w<=v])
    | otherwise             = Invisible
```

Fig. 1. Sample Relabel Policy Prototype

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 *invisible*.

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

Example 2 Continuing Example 1, Figure 1 defines the projection operator for the policy. A user may view only those tags on a label that the user's level dominates. Thus, `project(s, u:-[u, t])` returns label (`u:-[u]`). If the label is not visible to the viewer, then it is invisible. For example, `project(u, s:-[s, t])` returns (by label equivalence) the label `Invisible`. △

There are a number of conditions that a canonical policy must uphold in order to be secure. These ensure that a high-level user cannot interfere, in a visible way, with low-level labels. Sections 3 and 5 describe a subsystem for managing labels that is multilevel secure if the canonical policies that it supports uphold these conditions.

$$
\begin{array}{|l}
\hline
\textit{CP\_CView}[C, L, F] \\
\textit{CanonicalPolicy}[C, L, F] \\
\hline
\forall\, v, w : C;\ a : L \bullet \\
\qquad w \leq v \Rightarrow a \upharpoonright w = a \upharpoonright v \upharpoonright w \\
\hline
\end{array}
$$

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.

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

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).

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

This corresponds to the unwound non-interference requirement that a change in a low-level view of a label may not depend on any high-level information in the label (no read up). This requirement is slightly stronger than that originally specified in [5], but was found to be necessary for the kernelization of the label manager. We have,

theorem CP_NRUprev $[C, L, F]$

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

A policy that upholds these three conditions is called a secure canonical upgrade policy (SCUP).

$$
\begin{array}{|l}
\hline
\textit{SCUP}[C, L, F] \\
\textit{CP\_CView}[C, L, F] \\
\textit{CP\_NWD}[C, L, F] \\
\textit{CP\_NRU}[C, L, F] \\
\hline
\end{array}
$$

Example 3 The policy in Figure 1 is overly simplistic and is for illustrative purposes only. The policy described in [5] is an example of a more detailed relabel policy that was designed for the message-filter based model [8]. The scheme supports the relabelling of objects such that, when upgrades are requested, objects are migrated from one (single-level) object store to another. Object migration ensuring referential integrity may be achieved by viewing migration as a multilevel garbage collection [2] problem, or by encoding proxy information [10] in security labels. See [5] for a more detailed explanation. △

3 Label Manager

The label manager [5] provides a trusted interface to a SCUP policy. Its trusted operations make up the TCB extension required for standard multilevel secure systems. While it was intended specifically for message-filter based multilevel OODBMS, we believe it to be sufficiently general to be applicable to other systems.

Object identifiers are used to uniquely identify objects within an object-oriented database. 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

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

LabelStore defines the state of the label manager. It is accessed via the operations that make up its programming interface. Each object $o : OID$ has an associated security label $\delta(o)$.

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

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, where a request is made at level *req*? to apply the relabel function *rfun*? to the label of object *oid*?.

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

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

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

4 Security Analysis

Since the actions of the label manager are not mediated by the security kernel we must prove that it is multilevel secure. This is done by using an unwound version of non-interference [6, 14] to prove that no series of high-level requests to the manager can interfere with what a low-level, or disjoint-level, user can view.

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{VEquiv}[C, L, F]} \\
\hline
\textit{LabelStore}[C, L] \\
\textit{LabelStore}'[C, L] \\
\textit{SCUP}[C, L, F] \\
\textit{vl} : C \\
\hline
\forall\, o : \textit{OID}[C] \mid \textit{first}(o) \leq \textit{vl} \bullet \\
\qquad (o \in \mathrm{dom}(\delta) \Leftrightarrow o \in \mathrm{dom}(\delta')) \wedge \\
\qquad (o \in \mathrm{dom}\,\delta \cap \mathrm{dom}\,\delta' \Rightarrow \delta(o) \upharpoonright \textit{vl} = \delta'(o) \upharpoonright \textit{vl}) \\
\hline
\end{array}
$$

The first unwinding condition requires that each operation, when requested at a high-level, cannot interfere with a low-level view of the state (No Write Down). We have

theorem Relabel_NWD $[C, L, F]$
$\forall\, \textit{Relabel}[C, L, F];\ \textit{vl} : C \bullet$
$\qquad \neg\, (\textit{req?} \leq \textit{vl}) \Rightarrow \textit{VEquiv}[C, L, F]$

theorem ViewLabel_NWD $[C, L, F]$
$\forall\, \textit{ViewLabel}[C, L, F];\ \textit{vl} : C \bullet$
$\qquad \neg\, (\textit{req?} \leq \textit{vl}) \Rightarrow \textit{VEquiv}[C, L, F]$

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 (No Read Up). We have

theorem Relabel_NRU $[C, L, F]$
$\forall\, \textit{Relabel}[C, L, F];$
$\qquad \textit{Relabel}[C, L, F][\delta''/\delta, \delta'''/\delta'];\ \textit{vl} : C$
$\qquad \bullet\ \textit{VEquiv}[C, L, F][\delta''/\delta'] \Rightarrow \textit{VEquiv}[C, L, F][\delta'''/\delta]$

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

5 A Kernelized Label Manager

The message-filter model [8] supports multilevel security in object-oriented database systems according to the Bell and LaPadula (BLP) model [1]. 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 both as 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 (multilevel) persistent object store is partitioned into a collection of single-level stores (see Figure 2). The underlying security kernel, upholding the usual BLP axioms, ensures that it is not possible for an (untrusted) OODBMS to violate the multilevel policy.

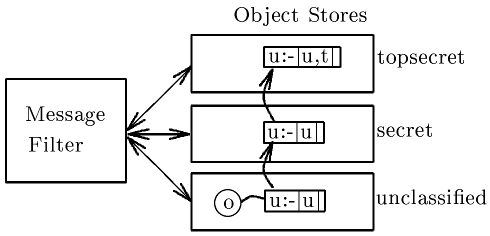


Fig. 2. Message Filter & Single Level Stores

In [5], the label manager runs as a separate trusted-subject servicing requests from objects in the the multilevel object store. Under this approach the entire label store (δ) forms part of the state of the manager, and since it is multilevel, it must be protected from direct access by any untrusted object. While its high-level specification (above) has been proven to be secure, an implementation should be constructed in terms of untrusted components (kernelized), if at all possible. A refinement of the label manager is now specified which, given a suitable message filter, can be kernelized in its entirety.

The basic strategy is to replicate each label for every level. A relabel broadcasts the request to every replicated label at levels that dominate the level of the request. A request to view returns the value of the label, at the level of the requester. The scheme works so long as broadcasts are implemented in a serializable and secure manner. Section 7 will outline an optimisation to the scheme: in practice it is not actually necessary to replicate at every level. Figure 2 illustrates this replication. In the abstract state the object o : *OID* has label $\delta(o) = (\mathtt{u:-}[\mathtt{u},\mathtt{t}])$. In the concrete state (implementation), the label is replicated at every level whereby the replicated value gives that level's view of the label.

Each replicated label is regarded as a single-level object (of type object label class) contained in a single-level object store. These objects, in turn, provide relabel and view-label methods.

$$\begin{array}{|l}
\hline
\textit{LabelStore}0[C, L] \\
\delta_0 : C \times \textit{OID}[C] \to L \\
\textit{CurrOIDs} : \mathbb{P}\, \textit{OID}[C] \\
\hline
\end{array}$$

For the purposes of modelling this implementation approach, the concrete state is specified in terms of a function δ_0 whereby $\delta_0(u, o)$ gives the level u copy of o's label. The set *CurrOIDs* defines the current set of objects Any state may serve as an initial state as long as replicated views are consistent.

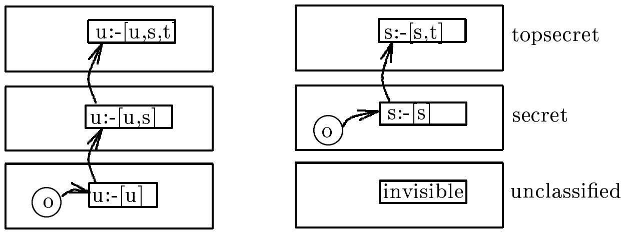
$$\begin{array}{|l}
\hline
\textit{InitialLabelStore}0[C, L, F] \\
\textit{SCUP}[C, L, F] \\
\textit{LabelStore}0[C, L] \\
\hline
\forall\, o : \textit{CurrOIDs};\ u, w : C \mid u \leq w \bullet \\
\qquad \delta_0(u, o) = \delta_0(w, o) \upharpoonright u \\
\hline
\end{array}$$

It turns out that *InitialLabelStore*0 also defines a state invariant that is maintained by the concrete relabel and view-label operations.

To relabel the label of object $oid?$ a relabel request message should be sent to the replicated copy of the label (represented as $\delta_0(req?, oid?)$) at the level of the requester $req?$. The relabel function $rfun?$ is applied to this label and the request is also broadcast to every replicated copy of the label whose level dominates $req?$. This is specified by concrete operation *Relabel*0.

If these label objects are maintained in a multilevel object store then the effect of the relabelling broadcast must be atomic across the single level stores. Thomas and Sandhu [18, 19] describe a message-filter based architecture that support write-up in a secure and serializable manner. This means that the broadcast (a write-up) specified in *Relabel*0 can be supported, in theory. The SINTRA replicated database [9] also supports both replication and write-ups and therefore should be capable of supporting the relabelling manager.

$$\begin{array}{|l}
\hline
\textit{Relabel}0[C, L, F] \\
\textit{SCUP}[C, L, F] \\
\Delta \textit{LabelStore}0[C, L] \\
\\
\textit{req}? : C \\
\textit{oid}? : \textit{OID} \\
\textit{rfun}? : F \\
\hline
\textbf{if } (\textit{oid}? \in \textit{CurrOIDs}) \\
\textbf{then } \delta_0' = \delta_0 \oplus \{\, u : C \mid \textit{req}? \leq u \\
\qquad\qquad\qquad \bullet (u, \textit{oid}?) \mapsto \mathcal{F}_{\mathcal{R}}(\textit{rfun}?, \textit{req}?, \delta_0(u, \textit{oid}?)) \,\} \\
\textbf{else } \ \delta_0' = \delta_0 \\
\\
\textit{CurrOIDs}' = \textit{CurrOIDs} \\
\hline
\end{array}$$



Relabel(mark, s, o) *Relabel*(up, u, o)

Fig. 3. Applying Relabel Functions

Example 4 Figure 3 illustrates the effect of a secret request to `mark` the label of object o in Figure 2, followed by an unclassified request to upgrade the object. In this figure the migration of o from unclassified to secret, as a result of the upgrade, is also illustrated. How this migration is achieved in practice is considered in [5]. △

At any moment, the value $\delta_0(u, o)$ gives the view of label o from level u. Therefore, the concrete version of *ViewLabel* is specified as follows.

$ViewLabel0[C, L, F]$

$SCUP[C, L, F]$
$\Xi LabelStore0[C, L]$

$req? : C$
$oid? : OID$
$lab! : L$

$\textbf{if } (oid? \in CurrOIDs \wedge first(oid?) \leq req?)$
$\textbf{then } lab! = \delta_0(req?, oid?)$
$\textbf{else } \ lab! = invisible$

Thus, to determine the label of an object *oid*?, the requester simply makes the request to the replicated copy of the object's label at the level of the requester.

Note that *LabelStore*0 specifies that an object's label is replicated at *every* level and not just at those levels that dominate the object's level. This may seem surprising, but it is desirable if further restrictions on SCUP policies are to be avoided. Maintaining a low-level version of a high-level object's label means that a low-level user can view the low-level effects that low relabel requests may have on the high-level object.

If this flexibility is not required then low-level views of high-level object labels could, for example, be assumed to be invisible, that is, $\delta(o) \upharpoonright v = invsible$, where $first(o) \not\leq u$. Thus, we always have $\delta_0(u, o) = invisible$ for $first(o) \not\leq u$ and it becomes necessary to replicate only the label of o at levels that dominate $first(o)$.

In [5], a create label operation is specified which is used to enter a label for a new object in the label store. Its specification and refinement is straightforward;

we do not include it here for reasons of space. However, we must address, at least in broad terms, the issue of entering new object label details into the replicated label store. Recording a new object label [5] is effectively a matter of adding tuple $((req?, newid?) \mapsto lab)$ to δ: a requester at level $req?$ has created a new object with identifier $newid?$ (in the object store at level $req?$) and wishes to assign it label lab.

With the concrete state implementation this is achieved by adding replicated entries $\{\, u : C \mid req? \leq u \bullet (u, (req?, oid?)) \mapsto lab \restriction u \,\}$ that correspond to a broadcast write-up from level $req?$. If $lab \restriction u = invisible$ for all other levels $(req? \not\leq u)$ then we are done. However, consider the case where the relabel policy permits low-level information to be encoded in what are, ostensibly, high-level labels. The message-filter will not permit a request at level $req?$ to create a label object at a disjoint or lower level. This problem is easily solved by adapting the *Relabel0* implementation such that the broadcast upwards will replicate missing labels, assigning them default values that can be specified as part of the policy.

6 Correctness of Refinement

Since the kernelized label manager is designed to be implemented in terms of untrusted components, with multilevel security enforced by the underlying TCB, it is not strictly necessary to prove that it is secure. However, it is necessary to prove that it is correct. This corresponds to proving that the behaviour of the concrete label manager is consistent with its abstract specification, that is, it is a refinement in the sense of [16].

Data Refinement In the kernelized label manger, $\delta_0(u, o)$ gives the view of label o from level u. In the abstract specification this corresponds to $\delta(o) \restriction u$. Therefore, given any concrete state we can retrieve its abstract equivalent. The abstraction (retrieve) relation relates the concrete and abstract states.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Abs[C, L, F]} \\
\hline
SCUP[C, L, F] \\
\\
LabelStore[C, L] \\
LabelStore0[C, L] \\
\hline
\mathrm{dom}\ \delta = CurrOIDs \\
\\
\forall\, o : CurrOIDs;\ u : C \bullet \\
\qquad \delta(o) \restriction u = \delta_0(u, o) \\
\hline
\end{array}
$$

Initial States Theorem. Any state of *LabelStore* may serve as a suitable initial abstract state. We can retrieve from any initial concrete state a valid abstract state, that is,

$$
\begin{array}{l}
\textbf{theorem}\ \mathrm{InitialStates}\ [C, L, F] \\
\quad \forall\, InitialLabelStore0[C, L, F] \bullet \\
\qquad \exists\, LabelStore[C, L] \bullet Abs[C, L, F]
\end{array}
$$

Operation Refinement All operations, both abstract and concrete are total, in the sense that they are defined for all possible input values. Thus it is safe to apply a concrete operation whenever it would be safe to apply the same request to its corresponding abstract form. If the label manager is in some concrete state, related to an abstract state by Abs, and the manager moves to a new concrete state as a result of a concrete relabel, then this new concrete state must be related (by Abs) to an abstract that can be reached by an abstract relabel from the original abstract state. This is illustrated in Figure 4, where schema $AbsAfter$ is used to retrieve the abstract after state. Operation $ViewLabel$ has a similar correctness requirement.

$$\begin{tikzcd}
LabelStore \arrow[r, "Relabel"] \arrow[d, leftrightarrow, "Abs"'] & LabelStore' \arrow[d, leftrightarrow, "AbsAfter"] \\
LabelStore0 \arrow[r, "Relabel0"'] & LabelStore0'
\end{tikzcd}$$

Fig. 4. Correctness of $Relabel0$.

$$AbsAfter[C, L, F] \mathrel{\widehat=} Abs[C, L, F][\delta'/\delta, \delta_0'/\delta_0, CurrOIDs'/CurrOIDs]$$

theorem RelabelCorrect $[C, L, F]$

$$\begin{array}{l} \forall\, Relabel0[C, L, F];\ Abs[C, L, F] \bullet \\ \qquad \exists\, \delta' : OID[C] \rightarrow\!\!\!\!\!+ L \bullet AbsAfter[C, L, F] \land Relabel[C, L, F] \end{array}$$

theorem ViewLabelCorrect $[C, L, F]$

$$\begin{array}{l} \forall\, ViewLabel0[C, L, F];\ Abs[C, L, F] \bullet \\ \qquad \exists\, \delta' : OID[C] \rightarrow\!\!\!\!\!+ L \bullet AbsAfter[C, L, F] \land ViewLabel[C, L, F] \end{array}$$

7 Optimisation

We outline an optimisation to the realization of the concrete label manager which, wherever possible, avoids replicating labels. First, we extend the definition of $SCUP$ with the addition of a label join operator $(\_ \odot \_)$, where $a \odot b$ gives a label representing the join of labels a and b. For consistency we assume that joining a lower-level view to a label's higher-level view makes no difference, that is, $u \le v \Rightarrow a \upharpoonright v = (a \upharpoonright v) \odot (a \upharpoonright u)$. Assuming a generalised form of the join operator, $\bigodot A$, which joins the set of labels A, then the state invariant

$$\begin{array}{l} \forall\, LabelStore0[C, L];\ v : C;\ o : OID[C] \bullet \\ \qquad \delta_0(v, o) = \bigodot\{\, u : C \mid u \le v \bullet \delta_0(u, o) \,\} \end{array}$$

follows, that is, we have a consistency between a view of a label and its lower views. Under certain circumstances it may be possible to compute the view, at level vl, of object vo's label by simply joining those views that are *strictly* dominated by vl. Define this view-invariant property as

$$
\begin{array}{|l}
\hline
\textit{ViewInv}[C, L, F] \\
\textit{SCUP}[C, L, F] \\
\textit{LabelStore0}[C, L] \\
vl : C \\
vo : \textit{OID}[C] \\
\hline
\delta_0(vl, vo) = \bigodot\{\, u : C \mid u \leq vl \wedge u \neq vl \bullet \delta_0(u, vo) \,\} \\
\hline
\end{array}
$$

If this property holds for some vl and vo, the result is that it is not necessary to store a replicated version of object vo's label at level vl, as it can be computed by joining its lower-level views. Thus, when a requester at level $req?$ stores a new label for a new object with id $(req?, id?)$, just one entry $(req?, (req?, id?)) \mapsto lab)$ is stored (assuming that $lab = lab \restriction req?$).

Relabelling maintains this view-invariant for views that cannot be altered by the request, that is,

$$
\begin{array}{l}
\forall\, \textit{Relabel0}[C, L, F];\ \textit{ViewInv}[C, L, F] \bullet \\
\qquad \neg\,(req? \leq vl) \Rightarrow \textit{ViewInv}[C, L, F][\delta_0'/\delta_0]
\end{array}
$$

Thus, if vo's label is not replicated at level vl ($req? \not\leq vl$) before the relabelling, it is not necessary to replicate it after the operation (this trivially follows from the definition of *Relabel*0). However, vo must be replicated at all levels that dominate $req?$: this allows the relabel to be broadcast correctly. If vo is not currently replicated at level $req?$ then its value is computed from the joins of lower-level views (that exist) and the relabel function applied. Since write-ups are allowed by the relabel policy, a relabelling may modify higher-level views differently to lower-level views. Therefore, the label computed for vo at level $req?$ must be broadcast (with the relabel function) to all levels that dominate $req?$.

Example 5 Consider the policy from Example 1. An unclassified object o has label `(u:-[u])`, which is stored in the unclassified object store (no replication). When a top-secret user requests a `mark`, its top-secret view is computed as `(u:-[u])`, function `mark` is applied, giving `(u:-[u,t])`, which is saved in the top-secret store. A secret `mark` request to object o results in the computation of a secret view of the label `(u:-[u])`, which is relabelled as `(u:-[u,s])` and stored. This relabel is also broadcast to the top-secret replicated label, changing it to `(u:-[u,s,t])`. △

The drawback of this scheme is that even with a modest number of relabel requests, it is likely that an object's label will end up being fully replicated. If we limit the relabel policy such that write-ups are not permitted then it is not necessary to replicate when the broadcast is complete.

$$\begin{array}{|l}
\hline
\textit{SCUP_NWU}[C,L,F] \\
\textit{SCUP}[C,L,F] \\
\hline
\forall f : F;\ s, v : C;\ a : L \bullet \\
\quad s \neq v \Rightarrow a \upharpoonright v = \mathcal{F}_{\mathcal{R}}(f, s, a) \upharpoonright v \\
\hline
\end{array}$$

In our opinion, this does not seem to be overly restrictive and we have,

$$\begin{array}{l}
\forall\, \textit{SCUP_NWU}[C,L,F];\ \textit{Relabel}0[C,L,F];\ \textit{ViewInv}[C,L,F] \bullet \\
\quad (\textit{req}? \neq \textit{vl}) \Rightarrow \textit{ViewInv}[C,L,F][\delta_0'/\delta_0]
\end{array}$$

In this case, if a label is not replicated at level vl ($req? \neq vl$) before the relabelling, it is not necessary to replicate it after the operation. A relabelling operation must, if necessary, replicate the label at level $req?$, and then broadcast the relabel function onto those existing replicated copies.

With this scheme, if a object label is not replicated at level $req?$ then the *ViewLabel*0 operation must compute it by joining the object's replicated labels from lower views. If the flow policy forms a lattice then it is possible to define the policy so that the label can be computed by joining labels whose lowest upper bound equals that of the label being calculated ($req?$).

8 Discussion and Conclusion

The label-based framework may be used to support the separation of concerns paradigm. An application may be developed, at a conceptual level, in terms of functional and security components. The security concerns are modelled in terms of objects or abstract data types: relabel functions (methods) which define how labels (state) may change. These security and functionality concerns may also be separately implemented. In addition to ensuring multilevel security of the functionality concerns, the framework ensures multilevel security and integrity of the security concerns.

This framework is based on the Message-Filter model for secure multilevel secure OODBMSs, extended to incorporate a trusted label manager. Section 5 gave a model for its kernelization, relying on a secure and consistent write-up mechanism such as those described in [9, 18]. But our results are not limited to the Message-Filter model, we believe that they could be used in any multilevel secure architecture that supports write-up.

The label-manager provides the basis for a multilevel program interpreter. While the policy in Figure 1 is a very simple example of a multilevel program, the framework could be used to support more elaborate security concerns such as those in [4, 5]. An interesting area for future study is to consider how this framework could be combined with recent work on compile-time information flow analysis of programs [21, 11] so that multilevel data-structures could be supported. These are topics for future study.

In one sense, the label manager is like a 'universal' trusted subject: an interpreter of trusted programs (relabel policies). A result of Section 4 is that if the relabel policy is SCUP then multilevel security (confidentiality) is ensured. However, an implication of Section 6 is that *any* relabel policy can be securely supported by the kernelized manger, but to maintain integrity, it is necessary to prove that the policy is SCUP.

We make two interesting observations about this relationship. Firstly, to prove integrity for this system it is necessary to perform what is effectively a non-interference analysis of the policy program. Secondly, if integrity is as critical a requirement as confidentiality then, whether we kernelize or not, we must nevertheless perform the non-interference analysis. We believe that similar observations can be made about *any* 'trusted' subject.

Acknowledgement

This work was done while I was a member of the Centre for Communications Systems Research, on leave from University College, Cork. I'd like to express my gratitude to Stewart Lee for inviting me and thank him, and the members of the Cambridge Computer Security Group, for a most enjoyable visit. Thanks also to the anonymous reviewers for their useful comments. This work was supported, in part, by a basic research grant from Forbairt (Ireland).

References

1. D. E. Bell and L. J. La Padula. Secure computer system: unified exposition and MULTICS interpretation. Report ESD-TR-75-306, The MITRE Corporation, March 1976.
2. E. Bertino, L.V. 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. D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security models. In *Proceedings Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, April 1987.
4. S.N. Foley. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *4th ACM Conference on Computer and Communications Security*, 1997.
5. S.N. Foley. Supporting secure cannonical upgrade policies in multilevel secure object stores. In *Proceedings of the 13th. Annual Computer Security Applications Conference*, pages 69–80, San Diego, CA, December 1997.
6. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86, 1984.
7. W.L. Hursch and C.V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA 02115, USA., 1995.

8. S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 1990.
9. M. Kang et al. Achieving database security through data replication: The SINTRA prototype. In *Proceedings of the 17th National Computer Security Conference*, pages 77–87, Baltimore, MD, USA, 1994.
10. M. Makpangou and M. Shapiro. The SOS object-oriented communication service. In *9th International Conference on Computer Communications*, Israel, 1988.
11. A.C. Myers and B. Liskov. A decentralized model for information flow control. In *16th Annual Symposium on Operating Systems Principles.* ACM, 1997.
12. M.J. Nash and K.R. Poland. Some conundrums concerning separation of duty. In *Proceedings of the Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990. IEEE Computer Society Press.
13. D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a policy neutral access control policy. In *Proceedings of the New Security Paradigms Workshop.* IEEE Computer Society Press, 1996.
14. J. Rushby. Noninterference, transitivity and channel-control security policies. Technical Report SRI-CSL-92-02, SRI International, Menlo Park, CA., December 1992.
15. M. Saaltink. The Z/EVES system. In *ZUM'97 (10th International Conference of Z Users)*, pages 72–85. Springer Verlag LNCS 1212, 1997.
16. J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, second edition, 1992.
17. D.F. Sterne, G.S. Benson, and H Tajalli. Redrawing the security perimeter of a trusted system. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 162–174, Franconia, NH, 1994.
18. R.K. Thomas and R.S. Sandhu. A kernelized architecture for multilevel secure object-oriented databases supporting write-up. *Journal of Computer Security*, 2(3):231–275, 1993.
19. R.K. Thomas and R.S. Sandhu. Supporting object-based high assurance write-up in multilevel databases for the replicated architecture. In *Proceedings of European Symposium on Research in Computer Security*, pages 403–428, UK, 1994.
20. U. S. Department of Defense. Trusted computer system criteria. Technical Report CSC-STD-001-83, U. S. National Computer Security Center, August 1983.
21. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3), 1995.

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. For example, the set of squares of all even natural numbers is defined as $\{ n : \mathbb{N} \mid (n \mathbin{\mathsf{mod}} 2) = 0 \bullet n^2 \}$. 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. For example, the set of all even numbers may be written as $\{ n : \mathbb{N} \mid (n \mathbin{\mathsf{mod}} 2) = 0 \}$. An equivalent way to define this set is as $\{ n : \mathbb{N} \bullet (2 * n) \}$—the predicate may be dropped if it equals true.

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 $\mathit{FlowPolicy}[C, L, F]$ 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 *RelabelPolicy* is equivalent to the syntactic inclusion of the variables and predicates of *FlowPolicy* within *RelabelPolicy*. Schema predicates are useful for writing theorems: in Section 6 the Initial States Theorem is a universal quantification over all the variables of $\mathit{InitialLabelStore0}$ such that it's predicate part implies the existence a δ such that the predicate part of Abs holds.

The decorated schema $\mathit{LabelStore}'[C, L]$ is $\mathit{LabelStore}[C, L]$ with all variables primed. Schema variables may be renamed in the usual way: $\mathit{LabelStore}[C, L][\delta'/\delta]$ an alternative way of writing $\mathit{LabelStore}'[C, L]$. The schema $\Delta \mathit{LabelStore}$ is a syntactic sugar for $\mathit{LabelStore} \wedge \mathit{LabelStore}'$. It is typically used for specifying state transitions, with undecorated variables representing 'before values' and decorated (primed) variables representing 'after values'. Schema $\Xi SCUP$ is the schema $\Delta SCUP$, but with the constraint that variable values are unchanged.

$\mathit{first}(a, b)$	Component a 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 $A \rightarrow B$
$\mathrm{dom}\ R, \mathrm{ran}\ R$	Domain and Range of relation R
$f \oplus g$	The functional override of f by g

Fig. 5. Some Operators from the Z Toolkit

B Theorem Proving with Z/EVES

This paper was typeset using LaTeX with the z-eves style. Thus, the LaTeX source of the paper acts as the input specification to the Z/EVES system [15]. In addition to using the system to syntax-, type- and domain- check the specifications in this paper, Z/EVES was used to verify the security and correctness of the label manager. The specification source, along with the Z/EVES proof scripts for all theorems are available from the author or under the author's WWW page at `<URL:http://www.cs.ucc.ie/sfoley.html>`.