Reasoning About Confidentiality Requirements

Simon N. Foley,
Department of Computer Science,
University College,
Cork, Ireland.

Abstract

Reflexive flow policies provide abstract characterizations of certain multilevel confidentiality requirements. This paper describes how reflexive flow policies can be used to construct and reason about large/complex multilevel policies. In particular, we describe how reflexive policies can be used to develop and reason about security policies for multilevel relational databases. Our approach facilitates a study of the relationship between security policy design and database design.

1 Introduction

Multilevel security is often associated with the military style classification ordering `unclass` < `secret` < `topsec`. However, multilevel or lattice based orderings have a wider application than simple military oriented policies. For example, lattices can be used to describe commercial Chinese wall policies[7, 14, 18] and a range of other general confidentiality policies[8, 15, 18].

We believe there are a number of potential problems when developing such large/complex lattice based confidentiality policies. These include: how does an individual responsible for specifying the security policy actually develop/arrive at a suitable lattice; how does a specifier relate a lattice policy to system components (bindings for users, objects, etc.), and how does a specifier reason about the appropriateness of a particular policy.

In this paper we regard the specification of a lattice based policy as a process that evolves, as its specifier considers the security requirements for a particular system. Thus, at some stage the specifier will want to determine if a revised policy R' is a suitable(secure) replacement for an earlier policy R. A policy may be developed as a number of independent or related components which the specifier will want to replace by a policy that reflects the requirements of the original components. While it is relatively straightforward to ascertain appropriate user clearances, from a lattice `unclass` < `secret` < `topsec`, it may be difficult to determine clearances if the lattice policy is large/complex. Additionally, how does one determine the most appropriate classifications for data items such as database tables and attributes. How does changing a relation table classification in a database affect the overall confidentiality policy?

In this paper we describe how representing a confidentiality policy as a reflexive relation provides a basis for addressing the issues outlined above. This reflexive relation, or reflexive information flow policy[6], defines all the different kinds, or classes, of information that are expected to exist in a system, along with a flow relation that defines whether information at one class may flow to another. Our interpretation of an information flow policy is different to [3] in that we also include details about system components in the information flow policy. In doing this, a reflexive policy provides an abstract formulation of a lattice policy plus its relationship with system components (users, objects, etc.). The theory of reflexive relations provides the policy specifier with a number of operators including policy refinement and composition operators.

Section 2 reviews the theory of reflexive flow policies and provides some new results about reflexive flow policies. In Section 3 we define what is meant by a system that is secure with respect to a reflexive flow policy. Section 3 also shows how this specification for a secure system relates to a noninterference based definition of a multilevel secure system. The reflexive flow policy formalism is used to develop/reason about large/complex policies which can then be converted into an implementation lattice policy and clearances for users.

In Section 4 we describe how reflexive flow policies can be used with multilevel relational databases. By including database information in the reflexive policy we provide a way to reason about the relationship between flow policy design and database design. Having arrived at a suitable database design and security policy, we can transform, in a manner similar to Section 3, the reflexive policy into an implementation lattice plus security levels for database tables and attributes.

Throughout the paper we use the Z notation[19] to provide a consistent syntax for presenting the mathematics in this paper. In using Z, it has been possible to syntax- and type-check definitions using the *f*UZZ tool.

2 Confidentiality Policies

An information flow policy defines the different classes of information that can exist in a system and whether information may flow between these classes. We let *class* be the set of all information classes; we are not further interested in its structure, except that it is non-empty. In [3], Denning argues that an information flow policy forms a lattice structure. In [6, 8] it is argued that information flow policies need not form lattices, and examples are given of useful policies that are not transitive and do not have consistent lowest upper bound operators. Our interpretation of an information flow policy is different to [3] in that we also include details about system components in the information flow policy. In addition to including the usual security classes `secret`, `topsec`, etc., we also include unique classes to represent significant system components, such as users, objects, programs, etc. For example, the class `admiral` represents the class of information an admiral may handle, and class `mission` represents the class of information that may be stored in a *Mission* database table. The information flow policy must specify the exact flow relationship between all system components (classes) and the other classes of information in the system.

In this paper a flow policy is represented as a reflexive relation between classes $class \leftrightarrow class$. This is a specialization of the more general conglomerate flow policy in [8], which can be used to define aggregation policies. The remainder of this section presents a general theory about reflexive relations. A more detailed account of part this theory can be found in [6, 8]. Sections 2.3 and 2.4 include new results not previously published.

2.1 A Theory of Reflexive Relations

The set of all reflexive relations between values of (generic) type X is defined as $\mathcal{R}[X]$, where

$$\mathcal{R}[X] == \{R : X \leftrightarrow X \mid \forall a : (\operatorname{dom} R \cup \operatorname{ran} R) \bullet a \mapsto a \in R\}$$

If $R \in \mathcal{R}[X]$ and $a \mapsto b \in R$, then we say that a is less than b in R. Normally we will use an infix form, $a \leadsto_R b$ to abbreviate $a \mapsto b \in R$. Let the alphabet αR of a reflexive relation R denote the set of values it is defined in terms of $(\operatorname{dom} R)$. We assume the existence of definitions for (subsets of $\mathcal{R}[X]$): $\mathcal{T}[X]$, the set of all reflexive relations that are transitive; $\mathcal{P}[X]$, the set of all partially ordered relations, and $\mathcal{L}[X]$, the set of all relations that form lattices.

A reflexive relation may be constructed using the $\leadsto$ operator: $A \leadsto B$ defines a relation such that all elements of A are less than all elements of B. $\bot A$ and $\top A$ define the least and most restrictive reflexive relations with alphabet A.

$$\begin{array}{l}
[X] \\
\hline
\bot, \top : (\mathbb{P}\, X) \to \mathcal{R}[X] \\
_ \leadsto _ : ((\mathbb{P}\, X) \times (\mathbb{P}\, X)) \to \mathcal{R}[X] \\
\hline
\bot A = A \times A \\
\top A = \operatorname{id} A \\
A \leadsto B = \top(A \cup B) \cup (A \times B)
\end{array}$$

Example 1 As argued earlier, we view an information flow policy as a reflexive relation between information classes. A very simple military-style policy can be specified as

$$\mathtt{MLS} == \{\mathtt{unclass}, \mathtt{secret}\} \leadsto \{\mathtt{secret}, \mathtt{topsec}\}$$

$\triangle$

The projection operator projects a relation R into relation $R@A$ with alphabet $\alpha R \cap A$ such that the relationships of R are preserved, for example, $\mathtt{MLS}@\{\mathtt{secret}, \mathtt{topsec}\} = \{\mathtt{secret}\} \leadsto \{\mathtt{topsec}\}$. The extension operator extends a relation R into a relation $R \uparrow A$ with alphabet $\alpha R \cup A$, such that all relationships are permitted so long as the restrictions on relationships in R are preserved.

$$\begin{array}{l}
[X] \\
\hline
_@_, _ \uparrow _ : (\mathcal{R}[X] \times \mathbb{P}\, X) \to \mathcal{R}[X] \\
\hline
R@A = \{a, b : (A \cap \alpha R) \mid a \leadsto_R b\} \\
R \uparrow A = \{a, b : (A \cup \alpha R) \mid \{a, b\} \subseteq \alpha R \Rightarrow a \leadsto_R b\}
\end{array}$$

Example 2 When specifying an information flow policy, we identify a unique security class for each user of the system. Each class represents the class of information the user (or class of user) is trusted to handle. The information flow policy must then specify the exact relationship between users (classes) and the other classes of information in the system. Let `admiral` denote the class of information an admiral is trusted to handle. An admiral has unlimited trust and thus there are no flow restrictions between class `admiral` and the classes of α`MLS`. This policy may be specified as

$$\mathtt{AdPol} == \mathtt{MLS} \uparrow \{\mathtt{admiral}\}$$

Observe that if we project this new policy through α`MLS`, we get the original policy, i.e., $\mathtt{AdPol}@\alpha\mathtt{MLS} = \mathtt{MLS}$. $\triangle$

In [6] it is shown that the set $\mathcal{R}[X]$ forms a lattice with partial order $\sqsubseteq$, and a lowest upper bound operator $\oplus$. The sublattice of reflexive relations with the same alphabet form an algebra with a complement operator not.

$$\begin{array}{l}
[X] \\
\hline
_ \sqsubseteq _ : \mathcal{R}[X] \leftrightarrow \mathcal{R}[X] \\
_ \oplus _ : (\mathcal{R}[X] \times \mathcal{R}[X]) \to \mathcal{R}[X] \\
\operatorname{not} : \mathcal{R}[X] \to \mathcal{R}[X] \\
\hline
R \sqsubseteq Q \Leftrightarrow (\alpha R \subseteq \alpha Q) \wedge (Q@\alpha R) \subseteq R \\
R \oplus Q = (R \uparrow \alpha Q) \cap (Q \uparrow \alpha R) \\
\operatorname{not} R = (\top(\alpha R)) \cup ((\bot(\alpha R)) \setminus R)
\end{array}$$

The ordering relation $\sqsubseteq$ allows us to compare flow policies. If $R \sqsubseteq Q$, then we say that Q is no less restrictive than R in that any flow that is not allowed by R will also not be allowed by Q. We view a $R \sqsubseteq Q$ relation as a refinement relation in the sense of [12]: the policy defined by Q is, in a security sense, an acceptable replacement for the policy R. Intuitively, this means that a system secure by policy Q is also secure by policy R.

Example 3 The policy complement operator is useful for constructing policies in terms of the flows that are *not* permitted. Let `captain` and `lieut` denote the class of information a captain and lieutenant may handle, respectively. A lieutenant is not trusted to handle `topsec` or `secret` information; a captain is not trusted to handle `topsec` information. We can define two policies to characterize these particular constraints:

$$\begin{array}{l} \mathtt{LieutPol} == \mathrm{not}\ (\{\mathtt{topsec}, \mathtt{secret}\} \rightsquigarrow \{\mathtt{lieut}\}) \\ \mathtt{CaptPol} == \mathrm{not}\ (\{\mathtt{topsec}\} \rightsquigarrow \{\mathtt{captain}\}) \end{array}$$

The lowest upper bound operator is useful for constructing complex policies from simpler policies: $R \oplus Q$ is a flow policy that enforces the flow restrictions of R and Q. Since $R \oplus Q$ is a lowest upper bound on R and Q, then it is, in a security sense, an acceptable replacement for R and Q. The overall flow policy defining the flow constraints on admirals, captains and lieutenants is specified as:

$$\mathtt{SysPol} == \mathtt{MLS} \oplus \mathtt{AdPol} \oplus \mathtt{LieutPol} \oplus \mathtt{CaptPol}$$

△

2.2 Mapping Transitive Orders to Lattices

A relation that is reflexive and transitive can be mapped to a partially ordered set (poset), while preserving the orderings of the original relation. This mapping may be done using either of the functions:

$$\begin{array}{l} [X] \\ \hline \lfloor \_ \rfloor \_, \lceil \_ \rceil \_ : \mathcal{R}[X] \to X \to \mathbb{P}\, X \\ \hline \lceil a \rceil_R = \{b : \alpha R \mid a \rightsquigarrow_R b\} \\ \lfloor a \rfloor_R = \{b : \alpha R \mid b \rightsquigarrow_R a\} \end{array}$$

The poset created by mapping $\lfloor \_ \rfloor_R$ has alphabet $\{a : \alpha R \bullet \lfloor a \rfloor_R\}$ and an ordering relation defined by subset. Birkoff[1] proves that given any relation $Q \in \mathcal{T}[X]$, then $a \rightsquigarrow_Q b \Leftrightarrow (\lfloor a \rfloor_Q \subseteq \lfloor b \rfloor_Q)$, and by duality that $a \rightsquigarrow_Q b \Leftrightarrow (\lceil b \rceil_Q \subseteq \lceil a \rceil_Q)$. What we call the Birkoff mappings (above), can be used to map, in an order preserving manner, a transitive relation $R \in \mathcal{T}[X]$ to the powerset lattice of αR, that is the (lattice) relation $\{A, B : \mathbb{P}\, \alpha R \mid A \subseteq B\}$. Denning[4] optimizes this lattice mapping by taking the poset (as generated by $\lfloor \_ \rfloor_R$), and gives an algorithm which adds 'missing' values (for lower and upper bounds) until it forms a lattice.

2.3 Reflexive Lattices

Given a reflexive relation R, two transitive orderings can be defined on R:

$$\begin{array}{l} [X] \\ \hline \_ \le \_ \_, \\ \_ \subseteq \_ \_ : \mathcal{R}[X] \to \mathcal{T}[X] \\ \hline a \le_R b \Leftrightarrow \lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R \wedge \lceil b \rceil_R \subseteq \lceil a \rceil_R \\ a \subseteq_R b \Leftrightarrow \lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R \wedge \lceil a \rceil_R \subseteq \lceil b \rceil_R \end{array}$$

That the above relations are transitive and reflexive follow from the transitivity and reflexivity of $\subseteq$. In the next section we will see how these relations are used. We have, $a \le_R b$ iff every value that is less than a is also less than b and that a is less than every value b is less than. For example, we have transitive ordering $\mathtt{secret} \le_{\mathtt{SysPol}} \mathtt{topsec}$, but $\neg\ \mathtt{admiral} \le_{\mathtt{SysPol}} \mathtt{captain}$. The interpretation we give to $a \subseteq_R b$ is that the relationships possible for a are contained within the relationships possible for b—everything that is less than, or greater than b is also less than or greater than a. For example, we have $\mathtt{secret} \subseteq_{\mathtt{SysPol}} \mathtt{captain}$ but $\neg\, \mathtt{topsec} \subseteq_{\mathtt{SysPol}} \mathtt{captain}$

A reflexive relation R forms a *reflexive lattice* if its transitive ordering relation $\le_R$ is partial and there exist valid lowest upper ($\oplus_R$) and greatest lower bound ($\otimes_R$) operators for $\le_R$ The policy `AdPol` forms a reflexive lattice, and thus we may use the operators $\oplus_R$ and $\otimes_R$ as we would if the policy was a lattice. For example, the class ($\mathtt{admiral} \oplus_{\mathtt{AdPol}} \mathtt{unclass}$) represents the class of information formed as a result of a composition of information at classes `admiral` and `unclass`. Finally, note that if the reflexive relation R is transitive, then the relation $\_ \le_R \_$ is equal to R (since the Birkoff mappings are order preserving).

2.4 Mapping Reflexive Orders to Lattices

Lattices form the basis of most systems that enforce multilevel security policies. This motivates the usefulness of the Birkoff mappings described above (and also Denning's optimization), where given a transitive reflexive relation that describes a flow policy we can build a lattice based equivalent of that policy. Building a lattice equivalent provides us with class combining operators, lowest upper bound and greatest lower bound. This section describes how an arbitrary reflexive policy can be mapped to a lattice policy.

The function Φ_ι maps a reflexive relation R to a function which maps classes of αR to intervals of a partial ordering $\Phi\ R$.

$$\begin{array}{l} [X] \\ \hline \Phi_\iota : \mathcal{R}[X] \to X \nrightarrow (\mathbb{P}\, X \times \mathbb{P}\, X) \\ \Phi : \mathcal{R}[X] \to \mathcal{P}[\mathbb{P}\, X] \\ \hline \mathrm{dom}(\Phi_\iota\ R) = \alpha R \\ \Phi_\iota\ R\ a = (\{b : X \mid \lceil a \rceil_R \subseteq \lceil b \rceil_R\}, \lfloor a \rfloor_R) \\ \Phi\ R = \{A, B : (\{x : \alpha R \bullet (\Phi_\iota\ R\ x)_1\} \\ \qquad \cup \{x : \alpha R \bullet (\Phi_\iota\ R\ x)_2\}) \mid A \subseteq B\} \end{array}$$

Given $R \in \mathcal{R}[X]$, the partial order generated has alphabet $\alpha(\Phi R)$ and an ordering relation defined by subset.

The constructed partial order ΦR preserves the orderings of the original R, under $\Phi_\iota\ R$. Define the following orderings over intervals of a poset:

$$
\begin{array}{|l}
\hline
[X] \\
\_ \leq_{[\_]} \_ : \mathcal{P}[X] \rightarrow \mathcal{P}[X \times X] \\
\_ \rightsquigarrow_{[\_]} \_ : \mathcal{P}[X] \rightarrow \mathcal{R}[X \times X] \\
\hline
\alpha\ (\_ \leq_{[R]} \_) = R = \alpha\ (\_ \rightsquigarrow_{[R]} \_) \\
A \leq_{[R]} B \Leftrightarrow A_1\ \leq_R\ B_1 \wedge A_2\ \leq_R\ B_2 \\
A \rightsquigarrow_{[R]} B \Leftrightarrow A_1 \leq_R B_2 \\
\hline
\end{array}
$$

where, if (a, b) is a pair, then $(a, b)_1$ gives the first component a, and $(a, b)_2$ component b. Given these orderings, we can prove that for any reflexive relation R, then $\Phi_\iota\ R$ is order-preserving in the sense that:

$$
\begin{array}{l}
a \rightsquigarrow_R b \Leftrightarrow (\Phi_\iota\ R\ a)\ \rightsquigarrow_{[\Phi R]} (\Phi_\iota\ R\ b) \wedge \\
a \leq_R b \Leftrightarrow (\Phi_\iota\ R\ a)\ \leq_{[\Phi R]} (\Phi_\iota\ R\ b)
\end{array}
$$

Thus we can map an arbitrary reflexive relation R to an interval poset which preserves the reflexive and transitive ordering of R.

As with the Birkoff mappings above, we can alternatively map the reflexive relation R to intervals defined on a powerset lattice, or to an optimized lattice generated from $\Phi\ R$ using Denning's algorithm[4]. We call this lattice the implementation lattice for reflexive relation R, because it defines how to implement a reflexive flow policy as a lattice. Furthermore, we call the components of the implementation lattice security *levels* to distinguish them from the security classes of the reflexive policy.

Example 4 The reflexive policy `SysPol` could be mapped by Φ_ι to a powerset lattice of the set α`SysPol`. This lattice will have 2^6 components, however Denning's optimization recognizes that the poset generated by Φ_ι`SysPol` is a lattice and thus we generate the lattice and intervals given in Figure 1. Note that we abbreviate classes to their first character in this figure. △

x	Φ_ι SysPol x
u	({uacl}, {uacl})
s	({usacl}, {usacl})
t	({ustacl}, {ustacl})
a	({uacl}, {ustacl})
c	({uacl}, {usacl})
l	({uacl}, {uacl})

$$
\begin{tikzcd}
\{\mathtt{ustacl}\} \\
\{\mathtt{usacl}\} \arrow[u] \\
\{\mathtt{uacl}\} \arrow[u]
\end{tikzcd}
$$

Figure 1: Intervals and Lattice for Φ_ι `SysPol`

The Φ_ι mapping defined above is a specialization (reflexive policies) of the Φ functions described in [8]. In [6] an earlier version of the Φ_ι function mapped (in an order preserving manner) a class a to the 'larger' interval $(\{a\}, \lfloor a \rfloor_R)$. This interval can be thought of as defining the set of all possible sets of classes that may flow to a. The lower bound of $\Phi_\iota\ R\ a$ above, lies between $\{a\}$ and $\lfloor a \rfloor_R$. Informally, the more 'transitivity' in R the 'higher' up the lower bound of $\Phi_\iota\ R\ a$ is in the interval $(\{a\}, \lfloor a \rfloor_R)$. A degenerate case occurs when the policy R is transitive. In this case each interval $\Phi_\iota\ R\ a$ defines a single point (for example, Φ`MLS` generates intervals ({u}, {u}), ({us}, {us}), ({ust}, {ust}) for `unclass`, `secret`, `topsec`, respectively). In general the new mapping defined by Φ_ι above tends to generate 'smaller' intervals (resulting in smaller implementation lattices) than the mapping in [6].

The order preserving mapping Φ_ι is useful in two ways. Firstly, even if the original reflexive flow policy does not form a reflexive lattice, we can generate an equivalent interval lattice which provides us with class (level) combing operators (lowest upper and greatest lower bound on interval pairs). Secondly, it can be used to map the reflexive relation into a lattice based structure which forms the basis of many existing multilevel security policies (as we shall see in later sections).

It is possible to define an order preserving inverse mapping for Φ which surjectively maps any poset plus set of intervals on that poset to a reflexive relation.

$$
\begin{array}{|l}
\hline
[X] \\
\Phi^{-1} : \mathcal{P}[X] \rightarrow\!\!\!\!\!\!+\ \mathbb{P}(X \times X) \twoheadrightarrow \mathcal{R}[X \times X] \\
\hline
I \subseteq P \Rightarrow \\
\qquad \Phi^{-1}\ P\ I = \{a, b : I \mid a_1 \mapsto b_2 \in P \bullet a \mapsto b\} \\
\qquad\qquad \cup \top(\mathrm{id}(\alpha\ P)) \\
\hline
\end{array}
$$

If I is a set of intervals from an arbitrary poset P then $R = \Phi^{-1}\ P\ I$ gives a reflexive policy which preserves the orderings of P in that,

$$
\begin{array}{l}
\forall\, a, b : I \bullet \\
\qquad a \leq_{[P]} b \Leftrightarrow a \leq_R b \wedge \\
\qquad a \rightsquigarrow_{[P]} b \Leftrightarrow a \rightsquigarrow_R b
\end{array}
$$

The significance of this mapping is that if we have a multilevel policy that is represented by a set of intervals on a lattice, then that policy can be described as a reflexive policy. Therefore, we can use reflexive flow policies to specify and reason about such multilevel policies, without any loss of generality.

3 Reasoning About Secure Systems

This section defines, at an abstract level, what it means for a system to be secure with respect to a reflexive flow policy. To simplify our exposition, we choose quite an abstract system model.

The environment of a system is made up of users. These users engage in actions with the system. This context perspective of a system and its environment is similar to that in CSP[11]. Let types *user* and *action* represent the sets of all possible users and actions, respectively. We are not further interested in their structure.

The relationship between a reflexive flow policy and the users and actions of a system is specified as

$$
\begin{array}{|l}
\hline
\textit{Binding}[\textit{class}] \\
R : \mathcal{R}[\textit{class}] \\
\textit{trust} : \textit{user} \nrightarrow \textit{class} \\
\beta : \textit{action} \nrightarrow \textit{class} \\
\textit{engage} : \textit{action} \nrightarrow \textit{user} \\
\hline
\operatorname{dom} \beta = \operatorname{dom} \textit{engage} \\
\operatorname{ran} \textit{engage} \subseteq \operatorname{dom} \textit{trust} \\
\forall\, x : \textit{action} \bullet \\
\qquad \beta\ x \subseteq_R \textit{trust}\ (\textit{engage}\ x) \\
\hline
\end{array}
$$

Users are bound to single classes from the reflexive policy R. Function $\textit{trust}\ u$ represents the class of information a user u is trusted to handle. Function $\beta\ x$ gives the class at which an action x is engaged. A user u can engage an action x only if $u = \textit{engage}\ x$. In addition to this, the user may engage this action x only if the user is trusted to handle information at the class $\beta\ x$—all flows permitted by the policy for class $\beta\ x$ must also be possible for the class of the user ($\textit{trust}\ u$). For example, a captain, trusted at class `captain` may engage an action at class `secret`, since `secret` $\subseteq_{\texttt{SysPol}}$ `captain`, but may not engage an action at `topsec`, since $\neg$ `topsec` $\subseteq_{\texttt{SysPol}}$ `captain`.

The valid relationship between security bindings and a non-interference relation is specified as:

$$
\begin{array}{|l}
\hline
\textit{SecureSystem}[\textit{class}] \\
\textit{Binding}[\textit{class}] \\
\_\ \textit{ni}\ \_ : \mathbb{P}\ \textit{action} \leftrightarrow \mathbb{P}\ \textit{action} \\
\hline
\forall\, a : \alpha R \bullet \\
\qquad \{x : \textit{action} \mid \neg\ \beta\ x \rightsquigarrow_R a\} \\
\qquad \textit{ni}\ \{x : \textit{action} \mid \beta\ x \rightsquigarrow_R a\} \\
\hline
\end{array}
$$

This defines what is meant by a secure system, with respect to a reflexive flow policy. We are not interested in specific computational models or definitions of non-interference. We choose to abstract these into the relation ($\_\ \textit{ni}\ \_$), where $A\ \textit{ni}\ B$ represents that actions A are non-interfering with actions B.

The Specification *SecureSystem* can be transformed to an equivalent *SecureSystem*$'$, which defines security in terms of intervals over the partially ordered set computed by Φ:

$$
\begin{array}{|l}
\hline
\textit{System}\Phi \\
\textit{SecureSystem}[\textit{class}] \\
\textit{SecureSystem}'[(\mathbb{P}\ \textit{class}) \times (\mathbb{P}\ \textit{class})] \\
\hline
R' = (\_ \rightsquigarrow_{[\Phi R]} \_) \\
\textit{trust}'\ u = \Phi_\iota\ R\ (\textit{trust}\ u) \\
\beta'\ x = \Phi_\iota\ R\ (\beta\ x) \\
\textit{engage}' = \textit{engage} \wedge (\_\ \textit{ni}'\ \_) = (\_\ \textit{ni}\ \_) \\
\hline
\end{array}
$$

If we study *SecureSystem*$'$, as defined by *System*Φ above, we will see that it matches the usual notion of multilevel non-interference security. Consider it's non-interference requirement. Since $R' = (\_ \rightsquigarrow_{[\Phi R]} \_)$, we have

$$
\begin{array}{l}
\forall\, a : \alpha(\_ \rightsquigarrow_{[\Phi R]} \_) \bullet \\
\qquad \{x : \textit{action} \mid \neg\ \beta'\ x \rightsquigarrow_{[\Phi R]} a\} \\
\qquad \textit{ni}\ \{x : \textit{action} \mid \beta'\ x \rightsquigarrow_{[\Phi R]} a\}
\end{array}
$$

which, given the definition of $(\_ \rightsquigarrow_{[\Phi R]} \_)$ is equivalent to

$$
\begin{array}{l}
\forall\, a : \alpha(\_ \rightsquigarrow_{[\Phi R]} \_) \bullet \\
\qquad \{x : \textit{action} \mid \neg\ (\beta'\ x)_1 \subseteq a_2\} \\
\qquad \textit{ni}\ \{x : \textit{action} \mid (\beta'\ x)_1 \subseteq a_2\}
\end{array}
$$

But by definition, $\alpha(\_ \rightsquigarrow_{[\Phi R]} \_) = \Phi\ R$, and since it is a reflexive relation, we have $\alpha(\Phi R) = \{a : \alpha(\_ \rightsquigarrow_{[\Phi R]} \_) \bullet a_2\}$. Thus we have

$$
\begin{array}{l}
\forall\, a : \alpha(\Phi\ R) \bullet \\
\qquad \{x : \textit{action} \mid \neg\ (\beta'\ x)_1 \subseteq a\} \\
\qquad \textit{ni}\ \{x : \textit{action} \mid (\beta'\ x)_1 \subseteq a\}
\end{array}
$$

The non-interference condition above is in terms of the computed poset $\Phi\ R$, and while actions are bound to pairs in β' above, only the lower bound $(\beta' x)_1$ on each pair is used. Therefore, in an implementation it is only necessary to bind each action to a single level from $\Phi\ R$[1]. Thus the non-interference requirement over the implementation lattice is similar to the usual multilevel non-interference requirement (that all actions that are not dominated by a level are non-interfering with those that are dominated by the level).

The trust classification specified for each user by *SecureSystem*$'$ above is given as an interval pair from the implementation poset $\Phi\ R$. This corresponds to a maximum and minimum clearance for each user. The binding requirement states that for each action x engaged by a user u then $\beta' x \subseteq_R \textit{trust}\ u$ This also simplifies to a test that the level (from the implementation poset) of the action lies within the clearance interval range assigned for each user.

Thus we can transform the specification of a secure system (with respect to a reflexive policy) to a specification of a secure system (with respect to a poset/lattice policy, as above). Additionally, we know that from the inverse mapping Φ^{-1}, that a reflexive flow policy is an adequate abstract representation of a multilevel policy, where actions are bound to levels and users bound to interval pairs of levels.

Example 5 Consider the security model for TMach[2]. We can use a reflexive relation to describe its flow policy and then using Φ, derive a lattice implementation. For example, consider the policy `SysPol` (Example 3). The lattice flow policy is Φ `SysPol`. An admiral, whose trust class is `admiral`, would have a clearance range between {`uacl`} and {`ustacl`}. The admiral may launch any process at class $a \subseteq_{\texttt{SysPol}}$ `admiral`. From the implementation lattice, this is a process with *Vmax* equal to $(\Phi_\iota\ \texttt{SysPol}\ a)_2$ and *Amin* equal to $(\Phi_\iota\ \texttt{SysPol}\ a)_1$.

[1] This single level $(\beta' x)_1$ actually corresponds to a class a from the original reflexive policy which maps to a single point, i.e., $\Phi_\iota\ R\ a = ((\beta' x)_1, (\beta' x)_1)$.

If $Amin < Vmax$, we assume that the launch has been appropriately mediated (for example, by an access model[18]) as the process is (partially) trusted. Objects in TMach are single level. Therefore we must ensure that if an admiral creates an object at security level a then the interval defined by Φ_ι `SysPol` a is a single point. It is possible to model objects with an interval range, for example, an object of class `admiral`. However ensuring noninterference is difficult: high water marks can be used, except that covert channels are likely unless strict access regimes are adopted. Database tables, which we study in the next section, are also examples of objects bound to a level interval. $\triangle$

4 Reasoning about Secure Databases

This section provides an abstract model for secure relational databases, with respect to reflexive policies. In particular, we describe how reflexive policies may be used to describe (multilevel) security policies for relational databases. We also look at the relationship between security policy and (secure) database design.

We adapt the specification of relational databases given in [20] to incorporate reflexive flow policies. Let the basic types $\mathcal{V}$ and $\mathcal{N}$ represent the set of all possible values and names (for attributes), respectively. Specify a record (tuple) of a relation table as

$$
\begin{array}{|l}
\textit{record}[\textit{class}] \\
\hline
\textit{attribs} : \mathbb{F}\,\mathcal{N} \\
\textit{value} : \mathcal{N} ⇻ \mathcal{V} \\
\delta : \mathcal{N} ⇻ \textit{class} \\
\hline
\mathrm{dom}\ \textit{value} = \textit{attribs} = \mathrm{dom}\,\delta \\
\hline
\end{array}
$$

The set *attribs* gives the attribute names of the record, *value* their values and δ their security classification. We do not consider tuple classifications. A database relation (table) is a collection of these records and is specified as

$$
\begin{array}{|l}
\textit{Relation}[\textit{class}] \\
\hline
\textit{attribs} : \mathbb{F}\,\mathcal{N} \\
\textit{records} : \mathbb{P}\,\textit{record}[\textit{class}] \\
\hline
\forall\, r : \textit{records} \bullet \\
\qquad r.\textit{attribs} = \textit{attribs} \\
\hline
\end{array}
$$

A relation scheme defines the key(s) and attributes of a relation. Typical multilevel databases such as Seaview[5] or SWORD[22, 23] associate lattice intervals with relations and their attributes (classification constraints). Given the inverse mapping Φ^{-1}, we know that a suitable reflexive relation representation exists for these intervals on a lattice. Thus, we may use a reflexive flow policy to describe the multilevel security requirements and define a security relational scheme as:

$$
\begin{array}{|l}
\textit{SecureScheme}[\textit{class}] \\
\hline
R : \mathcal{R}[\textit{class}] \\
\textit{tabclass} : \textit{class} \\
\textit{attribs} : \mathbb{F}\,\mathcal{N} \\
\delta : \mathcal{N} ⇻ \textit{class} \\
\textit{keys} : \mathbb{P}\,\mathcal{N} \\
\hline
\mathrm{dom}\ \delta = \textit{attribs} \\
\mathrm{ran}\ \delta \cup \{\textit{tabclass}\} \subseteq \alpha R \\
\textit{keys} \subseteq \textit{attribs} \\
\hline
\end{array}
$$

This defines the security requirements for the relation—we do not consider any other kinds of requirements, such as functional dependencies. The class associated with a table (*tabclass*) or attribute (δ) represents the class of information that may be associated with it.

It is straightforward to translate, using Φ, the components of *SecureScheme* and *Relation* above into their lattice based security form.

Example 6 Consider the database relation $\textit{Mission}(\underline{\textit{ship}}, \textit{destination})$. It defines the usual relationship between spaceships and destinations. Let classes `mission`, `galleon`, and `destination` represent mission, ship and destination information, respectively. Initially, there are no constraints on flows between these classes and the classes of `SysPol`. The security policy is thus defined as `TrekPol1`:

$$
\begin{array}{l}
\mathtt{DBpol1} == \bot\{\mathtt{mission}, \mathtt{galleon}, \mathtt{destination}\} \\
\mathtt{TrekPol1} == \mathtt{SysPol} \oplus \mathtt{DBpol1}
\end{array}
$$

This is a very simple policy; its implementation lattice is effectively a relabeling of Φ `SysPol` given in Figure 1. For example, the table classification *tabclass* is an interval from the implementation lattice, Φ_ι $\mathtt{TrekPol1}\ \mathtt{mission} = (\{\mathtt{uaclmgd}\}, \{\mathtt{ustaclmgd}\}$ (using the first character of each class to represent the class). Its lower bound gives the table's classification level, and the upper bound gives the maximum classification level of data that may be stored in the table. $\triangle$

Whether a relation instance/table satisfies the security constraints (entity integrity[5]) given by a security scheme is specified as:

$$
\begin{array}{|l}
\textit{SecureSat}[\textit{class}] \\
\hline
\textit{SecureScheme}[\textit{class}] \\
\textit{Relation}[\textit{class}] \\
\hline
\forall\, r : \textit{records};\ a : \textit{attribs};\ k : \textit{keys} \bullet \\
\qquad r.\delta\ a \subseteq_R \delta\ a \wedge \\
\qquad r.\delta\ a \subseteq_R \textit{tabclass} \wedge \\
\qquad r.\delta\ k\ \leq_R\ r.\delta\ a \\
\hline
\end{array}
$$

The constraints on classifications for attribute values correspond to the (implementation lattice) constraints—that the classifications must lie in the interval range of the attribute (column) and also of the table itself, and given any record, then the classification of each attribute value dominates the key value classifications.

A relational database is characterized as a collection of relation schemes and functional dependencies. The functional dependencies reflect the real-world nature of the data. To achieve data integrity, a database management system (DBMS) must help ensure that stored data is (externally) consistent with its real-world value. Normalization is a design criteria for relation schemes which is adopted by most RDBMSs. If relation schemes are in a normal form, then integrity problems such as insertion and deletion anomalies will not arise (in these DBMSs). Normalization also provides the basis for a logical view of data in a system—it helps identify what the data entities are, rather than how they are implemented[13].

A security relation scheme defines the security constraints on a relation scheme. The reflexive policy defines the constraints on the possible security classifications for data in a relation instance/table of the scheme (as specified by *SecureSat*). As it is currently specified, a security scheme will permit an attribute classification to be disjoint with the table classification. To enforce this constraint, the (secure) DBMS has to ensure that no value can be stored in that table. However, it is more reasonable to assume that such a security scheme is a poor 'design' and will not arise. Thus we give the first design rule/normal form for security schemes:

$1SNF[class]$

$SecureScheme[class]$

$$\forall\, a : attribs;\ k : keys \bullet \\ \quad \delta\ a\ \subseteq_R\ tabclass \land \\ \quad \delta\ k\ \leq_R\ \delta\ a$$

By converting the above to its implementation lattice equivalent we see that it corresponds to the usual assumptions made by secure DBMSs for multilevel relation schemes: that the interval class pair of every attribute fits 'within' the interval of the table class, and that the key class intervals are dominated by every attribute class interval.

We view this first secure normal form as a design rule which helps to avoid a potential integrity problem, where data values in a table are incorrectly classified (in the sense of *SecureSat*). Of course, there are other ways in which data in a table may be incorrectly classified, but we would treat them as integrity issues that the DBMS or the application system must address.

Debate has surrounded polyinstantiation in multilevel relations[16]—whether it yields an integrity problem or is a natural property of multilevel data. One design approach which views polyinstantiation as a potential integrity problem is the truthful database[22]. The truthful approach adopts a particular database manipulation regime which avoids polyinstantiation. It's essence is the 'insert at low' strategy, which requires that all users who have insert access to a table, perform inserts at the classification (level) of the table. From a security scheme design perspective, this requires that the lower bound on the key classification (level) be equal to the lower bound on the table classification (level). We will consider just this aspect of the truthfulness approach and examine how it influences database design.

We first extend the truthful strategy to a *table insert protocol*, where a particular insertion classification is chosen for a table (not necessarily the lower bound on the table class): inserts may be performed at this classification only. For example, we might chose `secret` as the insertion class for table *Mission*. Under this regime, covert channels do not arise since all insertions are done at this fixed level. With this scheme, it may not be always possible to establish an insert protocol that will permit all users who have access to the table to actually use it. The key classes of the table may be disjoint to the users (who can view it), or the users' classifications may be disjoint in some way. In the example above, using `secret` as the insertion class would mean that a lieutenant cannot read any record from *Mission*; thus we use `unclass` as the insertion class. A security scheme is in second secure normal form if it is possible to establish a insert protocol that will work for all users who have access to the table.

$2SNF[class]$

$1SNF[class]$
$trust : user \pfun class$

$$\forall\, V, I : \mathbb{P}\, class \bullet \\ \quad (V = \{u : \operatorname{ran}\ trust \mid tabclass \rightsquigarrow_R u \bullet u\} \land \\ \quad I = \{u : V;\ k : keys \mid \delta\ k\ \rightsquigarrow_R\ u\ \rightsquigarrow_R\ \delta\ k \bullet u\}) \\ \qquad \Rightarrow I \times V \subseteq R$$

The above can be interpreted as follows. From Section 3, a user can engage an action at any class so long as it 'lies within' the user's trust range (in the sense of the implementation lattice). Thus user u may view a table if it is possible for the user to engage a (view) action x whose classification dominates the table classification, i.e., if $tabclass \rightsquigarrow_R \beta x$ and $x \subseteq_R trust\ u$. By definition, this is equivalent to $tabclass \rightsquigarrow_R trust\ u$. The set V gives the set of classes of all users who may view the table. Similarly, the set I gives the set of classifications of users who may insert records into the table. A user may insert a new record into the table only if it is possible to engage an (insert) action at a classification which 'lies within' the range of classifications for the key attributes of the relation (assuming that the scheme is already in $1SNF$, then all attributes that contribute towards the key have the same classification).

Second secure normal form requires that the classes of all inserters may flow to the classes of all viewers, of a table. Given this, in the implementation lattice we have

$$\forall\, i : I;\ v : V \bullet (\Phi_\iota\ R\ i)_1 \subseteq (\Phi_\iota\ R\ v)_2$$

and since the implementation lattice is a lattice we know there exists a class l such that

$$\bigcup_{i:I} (\Phi_\iota\ R\ i)_1 \subseteq l \subseteq \bigcap_{v:V} (\Phi_\iota\ R\ v)_2$$

and we can use l as class for the insertion protocol. Thus, if a security scheme is in $2SNF$ then it is possible to adopt a insertion protocol on the table that will not require polyinstantiation and can be used by all accessers of the table.

Example 7 The security scheme for table *Mission* is in $1SNF$. If we consider only lieutenant, captain and admiral users then and the scheme is also in $2SNF$.

A user with class `ensign` works on the bridge of a spaceship and is privy to information at class `secret`. However, an ensign is unsophisticated and is not able to handle (in a trusted way) both secret and unclassified information. Secret information could be forced through this user down to unclassified, from a Trojan horse operating at secret. For example, the Trojan horse might attempt to influence the order in which the ensign enters unclassified information[21]. Thus, the ensign is trusted to handle secret information, but not unclassified information. Define the ensign policy to be

$$\begin{aligned}\texttt{EnsPol1} == {} & \text{not } (\{\texttt{topsec}\} \rightsquigarrow \{\texttt{ensign}\}) \oplus \\ & \text{not } (\{\texttt{ensign}\} \rightsquigarrow \{\texttt{unclass}, \texttt{lieut}\})\end{aligned}$$

and the overall database policy is now

$$\texttt{TrekPol2} == \texttt{SysPol} \oplus \texttt{EnsPol1} \oplus \texttt{DBpol1}$$

The resulting implementation lattice for policy `TrekPol2` is a simple relabeling of Φ `TrekPol1` and is given in Figure 2. Note that classes `mission`, `galleon` and `destination` have the same implementation intervals as `admiral`. While the relation scheme is still

x	Φ_ι `TrekPol2` x
u	$(\{\texttt{uaclmdg}\}, \{\texttt{uaclmdg}\})$
s	$(\{\texttt{usaclemdg}\}, \{\texttt{usaclemdg}\})$
t	$(\{\texttt{ustaclemdg}\}, \{\texttt{ustaclemdg}\})$
a	$(\{\texttt{uaclmdg}\}, \{\texttt{ustaclemdg}\})$
c	$(\{\texttt{uaclmdg}\}, \{\texttt{usaclemdg}\})$
l	$(\{\texttt{uaclmdg}\}, \{\texttt{uaclmdg}\})$
e	$(\{\texttt{usaclemdg}\}, \{\texttt{usaclemdg}\})$

$$\begin{tikzcd} \{\texttt{ustaclemdg}\} \\ \{\texttt{usaclemdg}\} \arrow[u] \\ \{\texttt{uaclmdg}\} \arrow[u] \end{tikzcd}$$

Figure 2: Intervals and Lattice for Φ_ι `TrekPol2`

$1SNF$, it is no longer in $2SNF$: if we establish an insert at `unclass` protocol, then an ensign cannot enter (secret) information, and if we establish an insert at `secret` protocol, then a lieutenant cannot query the table.

We might argue that the functionality of our particular application system requires that the *Mission* table is to be used for storing and sharing information about different missions (as entered by those trusted to source missions: admirals, captains or lieutenants). Further to this, we might decide that the unsophisticated ensign may only be aware of missions, but is not trusted to create them. In this case the policy is

$$\begin{aligned}\texttt{TrekPol3} == {} & \texttt{TrekPol2} \oplus \\ & \text{not } (\{\texttt{ensign}\} \rightsquigarrow \{\texttt{galleon}\})\end{aligned}$$

This revised policy results in a $2SNF$ security scheme, and permits an insert at `unclass` protocol. The new policy is a secure replacement for policy `TrekPol2`, since $\texttt{TrekPol2} \sqsubseteq \texttt{TrekPol3}$. The implementation lattice and order preserving mappings are given in Figure 3. In this case, not permitting the ensign to create new missions reflects that the existence of missions is an unclassified fact in the sense of [22]. Other $2SNF$ redesigns for the scheme include denying an ensign complete read or write access to the table. △

Example 8 Assume there are two classes of mission objectives: `battle` and `freight`. These classes are disjoint and the revised security policy is

$$\begin{aligned}\texttt{DBpol2} == {} & \texttt{DBpol1} \oplus \top\{\texttt{battle}, \texttt{freight}\} \oplus \\ & \text{not } (\{\texttt{topsec}\} \rightsquigarrow \{\texttt{freight}\}) \oplus \\ & \text{not } (\{\texttt{battle}\} \rightsquigarrow \{\texttt{unclass}\})\end{aligned}$$

$$\texttt{TrekPol4} == \texttt{SysPol} \oplus \texttt{DBpol2}$$

Note the additional constraints that `topsec` may not flow to `freight`, and `battle` may not flow to `unclass`. Figure 4 gives the mapping and lattice for this policy. Note that in constructing the implementation lattice, an additional level $\{\texttt{usaclmdgb}\}$ had to be added to make it a lattice. The classes `battle` and `freight` can be likened to compartments in the sense of a traditional MLS policy: a captain can process secret battle or secret freight information (classes $\texttt{captain} \oplus_{\texttt{TrekPol4}} \texttt{battle}$ or $\texttt{captain} \oplus_{\texttt{TrekPol4}} \texttt{freight}$), while a lieutenant may processes *only* unclassified or unclassified freight. The advantage of a reflexive policy is that it is easy to specify the relationship between these 'compartments' and the other classes in the policy.

The new security scheme for *Mission* is in $1SNF$ and $2SNF$. Suppose there are additional users of the system: Klingons and Ferengi, who are trusted to handle `battle` and `freight` information, respectively. Given the revised user bindings *trust*, *Mission* is no longer in $2SNF$: a table insert protocol cannot be established because there are two users, who may both access the table but whose classifications are disjoint. We could redesign the policy by constraining access to the table by these users, in a manner similar to Example 7 above. However, the requirements of the application system being developed might be that both Klingons and Ferengi should be allowed to create/access missions. An alternative might be to redesign the relation scheme to give two different entities: $\mathit{BattlePlans}(\underline{\mathit{ship}}, \mathit{destination})$ for battle objectives and $\mathit{Consignments}(\underline{\mathit{ship}}, \mathit{destination})$ for consignment details. In this case, the redesigned security schemes will be in $2SNF$. △

Normal forms (functional dependency or security) should not be taken as absolute. A System's Analyst must take an overall enterprise view of a system. An Analyst's objective is to provide an overall system design that will meet specific enterprise requirements. These requirements include functionality, security, cost, integrity etc. Consider functional dependency normal forms. A System's Analyst would view normalized data

x	Φ_ι TrekPol3 x
u	({uaclgmd}, {uaclgmd})
s	({usaclgmd}, {usaclegmd})
t	({ustaclgmd}, {ustaclegmd})
a	({uaclgmd}, {ustaclegmd})
c	({uaclgmd}, {usaclegmd})
l	({uaclgmd}, {uaclgmd})
e	({usaclegmd}, {usaclegmd})
g	({uaclgmd}, {ustaclgmd})
m	({uaclgmd}, {ustaclegmd})
d	({uaclgmd}, {ustaclegmd})

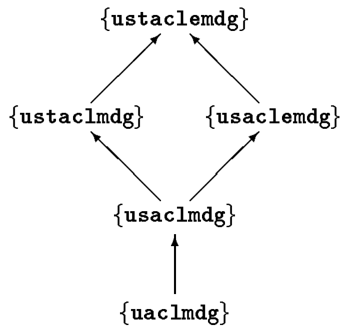

Figure 3: Intervals and Implementation Lattice for Φ_ι TrekPol3

u	({uaclmdg}, {uaclmdgf})
s	({usaclmdg}, {usaclmdgbf})
t	({ustaclmdgb}, {ustaclmdgbf})
a	({uaclmdg}, {ustaclmdgbf})
c	({uaclmdg}, {usaclmdgbf})
l	({uaclmdg}, {uaclmdgbf})
m	({uaclmdg}, {ustaclmdgbf})
d	({uaclmdg}, {ustaclmdgbf})
g	({uaclmdg}, {ustaclmdgbf})
b	({uaclmdgb}, {ustaclmdgb})
f	({uaclmdgf}, {usaclmdgf})

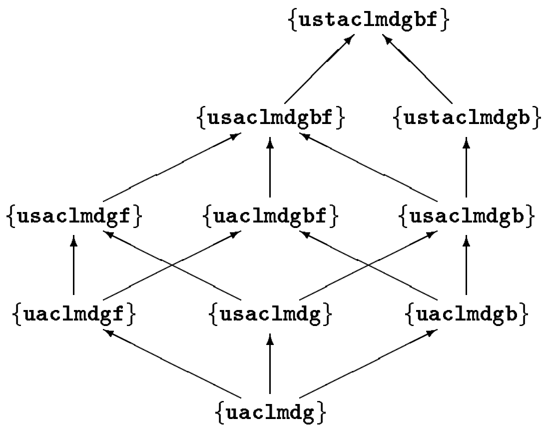

Figure 4: Intervals and Implementation Lattice for Φ_ι TrekPol4

as a specification of what data entities exist in the system; they need not necessarily correspond to how the entities are implemented. Thus normal forms need not necessarily be followed so long as external consistency is adequately addressed. Alternate normal forms (specifically, non 1NF forms) are also evidence that traditional normal forms do not always reflect the entities in the real world.

In the last two examples above we illustrated how secure normal forms could be used as a design guideline to ensure that the security scheme is in a form that is straightforward to maintain by a truthful DBMS. We should point out that it is possible to implement a non 2*SNF* scheme on a truthful database: the result is that not all users get complete access (as defined by the security scheme) to the database. For example, the policy TrekPol2 above can be implemented in SWORD (with an insert at low). But the result is that the ensign cannot insert into the table—in reality the policy is TrekPol3. Thus, given an intention to use a truthful implementation, a non 2*SNF* scheme is considered a poor design in the same sense that a non 1*SNF* scheme is a poor design: the expected permitted flows of the security scheme are at variance with the design of the database scheme.

However, 2*SNF* should not be viewed as an 'absolute' requirement. The absolute requirements are those for the enterprise in which the system exists. In Example 7 the system requirements might have been that an ensign is permitted to create missions for the *Mission* table. In this situation the Analyst must ensure that the overall system design provides the required functionality, but maintains confidentiality and integrity. The developer might decide that an ensign who wishes to insert a new mission, liaise with some trusted mediator who can insert (at unclass) a secret record into the *Mission* table for an ensign to populate with values. Alternatively, mission objectives in reality, may require cover stories and so polyinstantiating mission objectives is a more appropriate design in that it more closely reflects the real-world nature of the data.

5 Conclusion

This paper describes a framework for constructing and reasoning about complex multilevel policies. This framework forms the basis of an approach to specifying confidentiality policies, whereby a policy specifier can state the precise relationship between a multilevel policy and the system entities. This approach can be used to design complex multilevel policies that can in turn be transformed into lattice based policies and clearances/classifications for users, objects, etc. We believe that developing a complex multilevel policy using our approach is easier than it is to directly develop the lattice plus bindings: arriving at the relatively simple policy `TrekPol4` is straightforward, however we would consider the direct development (i.e., without using Φ_ι) of an implementation lattice (Figure 4) an arduous task.

The theory of reflexive relations provides a meaning for policy composition. It is possible to define: whether it is valid to, and what is meant by, the composition of two systems which enfore different multilevel policies[9]. We hope to report on this topic in the future.

Section 4 describes how reflexive policies can be used to describe and reason about multilevel policies for relational databases. We include database entities as part of the information flow policy. This approach facilitates a study of the relationship between security policy design and database design. Two design rules are given: $1SNF$ which corresponds to entity integrity, and $2SNF$ which defines the conditions under which a security policy can be consistently enforced using a truthful implementation. Expressing these rules in terms of reflexive policies allow policies to be specified and reasoned about in a consistent manner. Once an acceptable policy and database design has been established, the implementation lattice and table and attribute classification constraints can be computed from the reflexive policy.

We chose to limit our study to the relational model, because we were interested in how multilevel DBMS implementations influenced security policy and relation scheme design. Our conclusion is that 'polyinstantiation' and 'truthfulness' approaches can be viewed as implementation strategies for multilevel relational databases. The design rules described in this paper are limited to polyinstantiating and truthful databases. A topic for future research is the investigation of how our approach and design rules can be used/adapted for other multilevel data models such as [10] and other implementation strategies such as [17].

The development of any application system involves trade-offs: we provide a framework for a System Developer to study and reason about aspects of the designs of a multilevel (database) system before proceeding (forced or otherwise) towards a particular implementation.

References

[1] G. Birkhoff. Lattice Theory. Volume XXV of *American Mathematical Society Colloquium Publications*, American Mathematical Society, 3 edition, 1967.

[2] M. Branstad et. al. Trusted Mach Design Issues. in *Proceedings Third Aerospace Computer Security Conference*. December 1987.

[3] D.E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[4] D.E. Denning. *On the Derivation of Lattice Structured Information Flow Policies.* Technical Report CSD TR180, Purdue University, 1976.

[5] D.E. Denning et al. A Multilevel Relational Data Model, In *1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Oakland, April, 1987.

[6] S.N. Foley. A Model for Secure Information Flow. In *1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Oakland, May, 1989.

[7] S.N. Foley Secure Information Flow Using Security Groups In *Computer Security Foundations Workshop III*, IEEE Computer Society Press, 1990.

[8] S.N. Foley Aggregation and Separation as Non-Interference Properties. *Journal of Computer Security*, Vol. 1, Nr. 2, 1992.

[9] S.N. Foley Composing Systems with Heterogeneous Multilevel Policies *In preparation.*

[10] G.E. Gajnak Some Results from the Entity/Relationship Multilevel Secure DBMS Project. In *Research Directions in Database Security*, Menlo Park, CA, 1988. Springer-Verlag.

[11] C.A.R. Hoare *Communicating Sequential Processes.* Prentice Hall, 1985.

[12] J.L. Jacob The Variety of Refinements In *Proceedings of the 1991 Refinement Workshop*, Springer Verlag.

[13] S.M. McMenamin and J.F. Palmer *Essential Systems Analysis* Prentice Hall, 1984.

[14] C. Meadows Extending the Brewer Nash Model to a Multilevel Context. In *1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990.

[15] S.B. Lipner Nondiscretionary Controls for Commercial Applications *IEEE Symposium on Security and Privacy*, Oakland CA., April 1982.

[16] Panel Discussion on the Polyinstantiation Problem. *Computer Security Foundations Workshop IV*, IEEE Computer Society Press, 1991.

[17] R.S. Sandhu and Sushil Jajodia Honest Databases that can Keep Secrets In *Proceedings of the 14th National Computer Seurity Conference*, Washington, DC, 1991.

[18] R.S. Sandhu Lattice-Based Access Control Models IEEE Computer, November 1993, pp9–19.

[19] J.M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[20] B. Sufrin and J. Hughes *A Tutorial Introduction to Relational Algebra* Draft, Programming Research Group, Oxford. July, 1985.

[21] S. Wiseman. *Private Communication*, 1990.

[22] S.R. Wiseman On the Problem of Security in Databases. In *Database Security III: Status and Prospects*, Monterey CA, 1989, Elsevier Science Publishers.

[23] A.W. Wood et. al. The SWORD Multilevel Secure DBMS. Report 92005, Royal Signals and Radar Establishment, U.K., February 1992.

Appendix

Lemma 1 Mapping Φ_ι is a dual of the Birkoff mappings, in the sense that for any reflexive ordering R and $a, b \in \alpha R$, then

$$\lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R \quad \Leftrightarrow \quad (\Phi_\iota\ R\ a)_2 \subseteq (\Phi_\iota\ R\ b)_2 \qquad (1)$$
$$\lceil b \rceil_R \subseteq \lceil a \rceil_R \quad \Leftrightarrow \quad (\Phi_\iota\ R\ a)_1 \subseteq (\Phi_\iota\ R\ b)_1 \qquad (2)$$

Proof [Eqn. (1)] Follows from the definition of $\Phi_\iota\ R$.

Proof [Eqn.(1); $\Rightarrow$ part] Assume $\lceil b \rceil_R \subseteq \lceil a \rceil_R$. We have by definition that for all $x \in (\Phi_\iota\ R\ a)_1$ then $\lceil a \rceil_R \subseteq \lceil x \rceil_R$. Given the assumption, this implies $\lceil b \rceil_R \subseteq \lceil x \rceil_R$, which implies (by the definition of $\Phi_\iota\ R$), that $x \in (\Phi_\iota\ R\ b)_1$. Thus we have $(\Phi_\iota\ R\ a)_1 \subseteq (\Phi_\iota\ R\ b)_1$

Proof [Eqn. (2); $\Leftarrow$ part] Assume $(\Phi_\iota\ R\ a)_1 \subseteq (\Phi_\iota\ R\ b)_1$. Since $a \in (\Phi_\iota\ R\ a)_1$, then by assumption $a \in (\Phi_\iota\ R\ b)_1$, and the definition of $(\Phi_\iota\ R\ b)$ implies that $\lceil b \rceil_R \subseteq \lceil a \rceil_R$. □

Theorem 1 Mapping Φ_ι preserves the orderings of a reflexive relation, in that, for any reflexive relation R and $a, b \in \alpha R$, then

$$a \rightsquigarrow_R b \quad \Leftrightarrow \quad (\Phi_\iota\ R\ a) \rightsquigarrow_{[\Phi R]} (\Phi_\iota\ R\ b) \qquad (3)$$
$$a \leq_R b \quad \Leftrightarrow \quad (\Phi_\iota\ R\ a) \leq_{[\Phi R]} (\Phi_\iota\ R\ b) \qquad (4)$$

Proof [Eqn. (3)] Follows from the more general proof for conglomerate policies in [8].

Proof [Eqn. (4)] Follows from lemma 1 above. □

Theorem 2 The mapping Φ^{-1} preserves the orderings the partial order in that, given a partial order P and a set of intervals I on P, where $I \subseteq P$, then for all $a, b \in I$ we have

$$a \leq_{[P]} b \quad \Leftrightarrow \quad a \leq_R b \qquad (5)$$
$$a \rightsquigarrow_{[P]} b \quad \Leftrightarrow \quad a \rightsquigarrow_R b \qquad (6)$$

where $R = (\Phi^{-1}\ P\ I)$.

Proof [Eqn.(6)] We have by the definitions of Φ^{-1} and $(\_ \rightsquigarrow_{[P]} \_)$, that

$$\Phi^{-1}\ P\ I = \{a, b : I \mid a \rightsquigarrow_{[P]} b\} \cup \top(\mathrm{id}(\alpha P))$$

and thus equation (6) follows.

Proof [Eqn.(5)] By definition we have

$$a \leq_R b \Leftrightarrow \lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R \wedge \lceil b \rceil_R \subseteq \lceil a \rceil_R$$

Consider $\lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R$. Since R is equivalent to $(\_ \rightsquigarrow_{[P]} \_)$ (equation 6 above) we have

$$\lfloor a \rfloor_{(\_ \leq_{[P]} \_)} \subseteq \lfloor b \rfloor_{(\_ \leq_{[P]} \_)}$$

which is

$$\{x : \alpha(\_ \leq_{[P]} \_) \mid x_1 \rightsquigarrow_P a_2\} \subseteq \{x : \alpha(\_ \leq_{[P]} \_) \mid x_1 \rightsquigarrow_P b_2\}$$

But, since $(\_ \leq_{[P]} \_)$ is reflexive, we have $\operatorname{dom} \alpha(\_ \leq_{[P]} \_) = \alpha P$, and thus we have

$$\{x : \alpha P \mid x \rightsquigarrow_P a_2\} \subseteq \{x : \alpha P \mid x \rightsquigarrow_P b_2\}$$

And since P is transitive, Birkoff's mappings give $\lfloor a_2 \rfloor_P \subseteq \lfloor b_2 \rfloor_P$ i.e., $a_2 \rightsquigarrow_P b_2$. Thus we have $\lfloor a \rfloor_R \subseteq \lfloor b \rfloor_R \Leftrightarrow a_2 \rightsquigarrow_P b_2$.

By symmetry, we can also prove that $\lceil b \rceil_R \subseteq \lceil a \rceil_R \Leftrightarrow \lceil b_1 \rceil_P \subseteq \lceil a_1 \rceil_P$ and thus we have $\lceil b \rceil_R \subseteq \lceil a \rceil_R \Leftrightarrow a_1 \rightsquigarrow_P b_1$ and the theorem is proven. □