Minimal Data Upgrading to Prevent Inference and Association Attacks*

Steven Dawson[1] **Sabrina De Capitani di Vimercati**[2†] **Patrick Lincoln**[1] **Pierangela Samarati**[1‡]

(1) Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{dawson|lincoln|samarati}@csl.sri.com

(2) Dip. di Scienze dell'Informazione
Università di Milano
20135 Milano, Italy
decapita@dsi.unimi.it

Abstract

Despite advances in recent years in the area of mandatory access control in database systems, today's information repositories remain vulnerable to inference and data association attacks that can result in serious information leakage. Such information leakage can be prevented by properly classifying information according to constraints that express relationships among the security levels of data objects. In this paper we address the problem of classifying information by enforcing explicit data classification as well as inference and association constraints. We formulate the problem of determining a classification that ensures satisfaction of the constraints, while at the same time guaranteeing that information will not be unnecessarily overclassified. We present an approach to the solution of this problem and give an algorithm implementing it which is linear in simple cases, and low-order polynomial (n^2) in the general case. We also analyze a variation of the problem which is NP-hard.

1 Introduction

Mandatory policies control access to information on the basis of classifications, taken from a partially ordered set, assigned to data objects and subjects requesting access to them. Classifications assigned to information reflect the sensitivity of that information, while classifications assigned to subjects reflect their trustworthiness not to disclose the information they access to subjects not cleared to see it. By controlling read and write operations accordingly — allowing subjects to read information whose classification is dominated by their level and write information only at a level that dominates theirs — mandatory policies provide a simple and effective way to enforce information protection [1]. In particular, the use of classifications and the access restrictions enforced upon them ensure that information will be released neither directly, through a read access, nor indirectly, through an improper flow into objects accessible by lower level subjects. This provides an advantage with respect to authorization-based control, which suffers from this last vulnerability.

The relatively recent application of mandatory security policies to database systems has resulted in a vast amount of research and the proposal of several models for multilevel database systems [8, 10, 13, 14, 19]. Despite this, the lack of support for expressing and combating inference and data association channels that improperly leak protected information remains a major limitation [7, 9, 11]. Without such a capability, the protection requirements of the information are clearly open to compromise. Proper classification of data is crucial for classification-based control to effectively protect information secrecy.

We address the problem of computing security classifications to be assigned to information in a database system, while reflecting both explicit classification requirements and necessary classification upgrading to prevent exploitation of data associations and inference channels that leak sensitive information to lower levels. One of the major challenges in the determination of a data classification involving classification upgrading is the need to minimize the resulting loss of information visibility. Previous proposals in this direction are based on the application of optimality cost measures, such as upgrading the minimum number of attributes or executing the minimum number of upgrading steps [16, 15], or explicit constraints allowing the specification of different preference criteria [3]. Determining such optimal classifications is often an NP-hard problem, and existing approaches typically perform exhaustive examination of all possible solutions [16, 3]. Moreover, these proposals are limited to the consideration of totally ordered sets of classifications [16, 15, 3] and intra-relation constraints due to functional and multivalued dependencies [16]. While these cost-based approaches afford a high degree of control over how objects are classified, the computational cost of computing optimal solutions may be prohibitive. Moreover, it is generally far from obvious how to manipulate costs to achieve the desired classification behavior and optimality measures based on it can be debated. For the similar problem of computing data classifications from classification constraints on views, Qian [12] provides a polynomial time algorithm, but the approach does not guarantee minimality and, in fact, tends to overclassify information unnecessarily.

*This work was supported in part by National Science Foundation under grants ECS-94-22688 and CCR-9509931, and by DARPA/Rome Laboratory under contract F30602-96-C-0337.

†This work was performed while the author was visiting SRI International, Computer Science Laboratory, supported in part by the National Science Foundation under grant ECS-94-22688.

‡On leave from Università di Milano. Author's permanent address: Università di Milano, Polo Didattico e di Ricerca di Crema, Via Bramante 65, 26013 Crema - Italy; e-mail: samarati@dsi.unimi.it.

We propose an efficient (low-order polynomial) approach that, given a set of classification constraints, computes a classification to be assigned to data objects that satisfies the constraints while minimizing the loss of information visibility. The constraints we consider express lower bounds on the classifications of single objects (explicit requirements) or sets of objects (association constraints), as well as relationships that must hold between the classifications of different objects (inference constraints).

The contributions of this paper can be summarized as follows. First, we introduce a notion of minimality that captures the property of a classification satisfying the protection requirements without overclassifying data. Second, we describe an efficient approach for computing minimal classifications and present an algorithm implementing our approach that executes in (low-order) polynomial time. We further identify an important class of constraints, termed *acyclic* constraints, for which the algorithm executes in time linear in the size of the constraints. Third, we extend the results to allow classification constraints that specify also *upper* bounds on the levels that may be assigned to objects (which explicitly require visibility of information) and show that polynomial-time complexity is preserved. Fourth, we show that the approach is applicable also to security lattices that are not complete lattices (i.e., may be lacking top or bottom elements), but that for non-lattices (arbitary partial orders), the problem of computing a minimal classification is NP-complete.

The technique we describe can form the basis of a practical tool for efficiently analyzing and enforcing classification constraints. For concreteness we frame our work in the context of relational database systems. We note, however, that our approach does not depend in any way on this assumption and can be generally applied in any context where information may need to be classified, such as file systems, object-oriented databases, or component-based system designs.

2 Problem definition

Mandatory policies are based on assignment of access classes to objects and requesting subjects. Access classes L are related by a partial order, called the *dominance relation*, denoted $\succeq$, that governs the visibility of information, where a subject has access only to information classified at the subject's level or below[1]. The partially ordered set $(L, \succeq)$ is generally assumed to be a lattice, and often, access classes are assumed to be pairs of the form (s, C), where s is a classification *level* taken from a totally ordered set and C is a set of *categories* (or *compartments*) taken from an unordered set. In this context, an access class dominates another iff the classification level of the former is at least as high in the total order as that of the latter, and the set of categories is a superset of that of the latter. Figure 1(a) illustrates an example with two levels and two categories. For generality, we do not restrict our approach to specific forms of lattices, but assume access classes, to which we refer alternately as security levels or classifications, to be taken from a generic lattice.

The security level $\lambda(A)$ to be assigned to an attribute A may depend on several factors, which we categorize as: *basic* classification constraints, *inference* and *association* constraints, and *integrity* constraints. Basic constraints specify a minimum level to be assigned to an attribute, for example, $\lambda(\texttt{name})=\texttt{Unclassified}$ and $\lambda(\texttt{salary})=\texttt{Confidential}$. Inference and association constraints are used to prevent bypassing of basic constraints through data inference and to place stronger restrictions on the combined visibility of different attributes. Examples of this type include $\mathsf{lub}\{\lambda(\texttt{name}), \lambda(\texttt{salary})\} \succeq \texttt{Secret}$ and $\mathsf{lub}\{\lambda(\texttt{rank}), \lambda(\texttt{department})\} \succeq \lambda(\texttt{salary})$, where lub denotes the least upper bound of a set of security levels. Integrity constraints are imposed by the security model itself and typically include *primary key* constraints and *referential integrity* constraints [19]. Primary key constraints require that key attributes be uniformly classified and that their classification be dominated by that of the corresponding non-key attributes. Referential integrity constraints require that the classification of an attribute representing a foreign key must dominate the classification of the attribute for which it is foreign key. All these categories of classification constraints are captured in a single general form as follows.

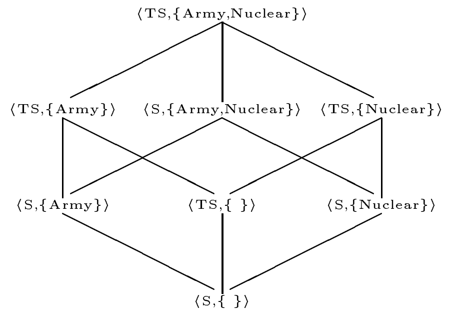
Definition 2.1 (Classification Constraint) *Let $\mathcal{A}$ be a set of attributes and $\mathcal{L} = (L, \succeq)$ be a security lattice. A* classification constraint *over $\mathcal{A}$ and $\mathcal{L}$ is an expression of the form $\mathsf{lub}\{\lambda(A_1), \ldots, \lambda(A_n)\} \succeq X$, where $n > 0$, $A_i \in \mathcal{A}$, $i = 1, \ldots, n$, and X is either a security level $l \in L$ or is of the form $\lambda(A)$, with $A \in \mathcal{A}$. If $n = 1$, the expression may be abbreviated as $\lambda(A_1) \succeq X$.*

For simplicity, we frequently denote classification constraints as pairs (*lhs*,*rhs*), where *lhs* is the set of attributes appearing on the left hand side of the constraint, and *rhs* is the attribute or security level appearing on the right hand side of the constraint. We refer to classification constraints whose left hand side is singleton as *simple* constraints, and to constraints with multiple elements in the left hand side as *complex* constraints. Any set of classification constraints can be viewed as a directed graph containing a node for each attribute $A \in \mathcal{A}$ and security level $l \in L$. Each constraint (*lhs*,*rhs*), with *lhs*=$\{A_1, \ldots, A_n\}$, is represented by a directed edge from node A_1, if $n = 1$, or hypernode containing $A_1, \ldots, A_n$, if $n > 1$, to node *rhs*. Figure 2(a) illustrates an example of a classification constraint graph. Circle nodes represent attributes, square nodes represent security levels, and dashed ellipses represent hypernodes. In the remainder of the paper we refer to the constraints and to their graphical representation interchangeably, and we often refer to a constraint (*lhs*,*rhs*) as the existence of an edge between *lhs* and *rhs*. Constraints whose graph representation is acyclic (i.e., is a dag) are called *acyclic* constraints, while constraints involved in a cycle, including cycles through hypernodes[2], are called *cyclic* constraints. A cycle involving only simple constraints is called a *simple cycle*. For example, in Figure 2(a) constraints $(\{E,F\},M)$, (M,G), $(\{D,G\},C)$, (C,E), (C,F), and $(\{F,I\},B)$ are cyclic; constraints (I,O), (O,N), and (N,I) constitute a simple cycle; and all other constraints are acyclic.

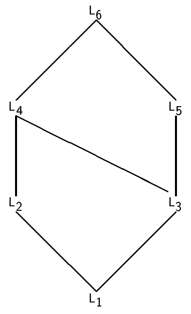
A classification $\lambda : \mathcal{A} \mapsto L$ is an assignment of security levels in L to objects (attributes) in $\mathcal{A}$. A classification λ *satisfies* a set C of constraints, denoted $\lambda \models C$, iff for each constraint, the expression obtained by substituting every $\lambda(A)$ with its corresponding level holds in the lattice. For a given a set of classification constraints, multiple level assignments may exist that satisfy the constraints. However,

[1]The expression $a \succeq b$ is read as, "a dominates b", and $a \succ b$ as, "a strictly dominates b" (i.e., $a \succeq b$ and $a \neq b$).

[2]For the purpose of determining cycles, the attribute on the right hand side of a constraint is considered reachable from *every* attribute on the left hand side. Note that hypernodes never have incoming arcs, but the attribute nodes they contain may.



(a)



(b)

Figure 1: Examples of classification lattices.

not all them are equally good. For instance, the mapping $\lambda : \mathcal{A} \mapsto \{\top\}$ classifying all data at the highest possible level satisfies any set of classification constraints. Such a strong classification is clearly undesirable unless required by the classification constraints, as it results in unnecessary information loss (by preventing release of information that could be safely released). Although the notion of information loss is difficult to make both sufficiently general and precise, it is clear that a first requirement in minimizing information loss is to prevent *over classification* of data. That is, the set of attributes should not be assigned security levels higher than necessary to satisfy the classification constraints. A classification mapping that meets this requirement is said to be *minimal*. To be more precise, we first extend the notion of dominance to classification assignments. For a given set $\mathcal{A}$ of attributes, security lattice $(L, \succeq)$, and mappings $\lambda_1 : \mathcal{A} \mapsto L$ and $\lambda_2 : \mathcal{A} \mapsto L$, we say that $\lambda_1 \succeq \lambda_2$ iff $\forall A \in \mathcal{A} : \lambda_1(A) \succeq \lambda_2(A)$. The notion of minimal classification can now be defined as follows.

Definition 2.2 (Minimal classification) *Given a set $\mathcal{A}$ of attributes, security lattice $\mathcal{L} = (L, \succeq)$, and a set C of classification constraints over $\mathcal{A}$ and $\mathcal{L}$, a classification $\lambda : \mathcal{A} \mapsto L$ is* minimal *with respect to C iff (1) $\lambda \models C$; and (2) for all $\lambda' : \mathcal{A} \mapsto L$ such that $\lambda' \models C$, $\lambda \succeq \lambda' \Rightarrow \lambda = \lambda'$.*

The main problem now is to compute a minimal classification from a given set of classification constraints.

Problem 2.1 (MIN-LATTICE-ASSIGNMENT) *Given a set $\mathcal{A}$ of attributes to be classified, a security lattice $\mathcal{L} = (L, \succeq)$, and a set C of classification constraints over $\mathcal{A}$ and $\mathcal{L}$, determine a classification assignment $\lambda : \mathcal{A} \mapsto L$ that is minimal with respect to C.*

In general, a set of constraints may have more than one minimal solution. The following sections describe an approach for efficiently computing one such minimal solution and a (low-order) polynomial-time algorithm that implements the approach.

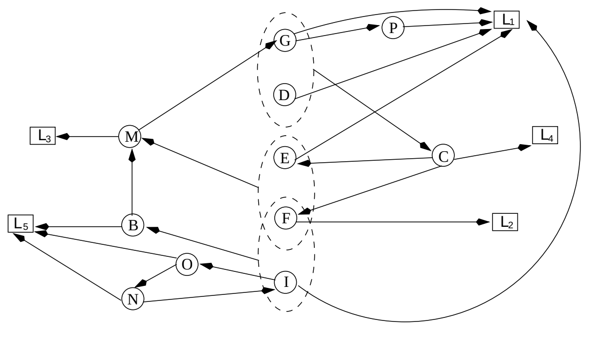
3 Sketch of the Approach

A basic requirement that must be satisfied to ensure the existence of a classification λ is that the set of classification constraints provided as input be *complete* and *consistent*. A set of classification constraints is complete if it defines a classification for each attribute in the database. It is consistent if there exists an assignment of levels to the attributes, that is, a definition of λ, that simultaneously satisfies all classification constraints. Completeness is easily guaranteed by providing a *default* classification constraint of the form $\lambda(A) \succeq \bot$ for every attribute $A \in \mathcal{A}$. In addition, any set of constraints of the form specified by Definition 2.1, which use only the dominance relationship $\succeq$ and security levels (constants) only on the right hand side, is consistent, since mapping every attribute to $\top$ trivially satisfies all such constraints. We assume then, without loss of generality, that any input set of classification constraints is complete and consistent. We further assume the left and right hand sides of each constraint to be disjoint, since constraints not satisfying this condition are trivially satisfied.

3.1 Acyclic Constraints

A straightforward approach to computing a minimal classification involves performing a backward propagation of security levels to the attributes. Consider an acyclic constraint graph with no hypernodes (simple constraints only) and assume all attributes are initially assigned level $\bot$. Starting from the leaves, we traverse the graph backwards (opposite the direction of the edges) and propagate levels according to the constraints. Intuitively, propagating a level to an attribute node A according to a constraint edge (A, X) means assigning to A the least upper bound of its current level, $\lambda(A)$, and the level of X (l, if X is a security level l; $\lambda(X)$ otherwise). As long as X has been assigned its final level, propagating in this way ensures that A is assigned the lowest level that satisfies all constraints on it. Thus, for acyclic simple constraints the unique, minimal solution can be computed simply by propagating levels back from the leaves, visiting all the nodes in (reverse) topological order. This process is clearly the most efficient one can apply, since each edge is traversed exactly once. In terms of the constraints, this corresponds to evaluating the constraints in a specific order, evaluating each constraint only once, when the level of its right hand side becomes definitely known, and upgrading the left hand side accordingly.

In a set of acyclic constraints, the propagation method described for simple constraints alone requires only minor adaptation to handle complex constraints as well. The key observation is that, if a complex constraint is not already satisfied, it can be solved minimally by upgrading any one of the attributes on the left hand side, provided that neither the level of the right hand side nor the levels of any other attributes on the left hand side are later altered. As long as the constraints are acyclic, there exists an order of constraint evaluation (security level back-propagation) that ensures that the security levels of all attributes involved in a complex constraint are known prior to the selection of one for upgrading, if necessary, to satisfy the constraint. For example, referring to the lattice in Figure 1(b), the constraints $\lambda(\{A, B\}) \succeq \mathsf{L}_4$, $\lambda(A) \succeq \mathsf{L}_1$, and $\lambda(B) \succeq \mathsf{L}_2$ can be solved by upgrading either A to L_3 or B to L_4. Note that either solution is minimal according to Definition 2.2, and thus, minimal solutions for sets that include complex constraints are generally not unique. The particular minimal solution



(a)

Priority Sets

		[4]	[3]						[2]			[1]
		P	B	C	E	F	G	M	I	O	N	D
• **initial levels**		L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6
P	–	$\mathbf{L_1}$	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6	L_6
B	$\mathbf{try}(B, L_5)$		$\mathbf{L_5}$	L_6	L_6	L_6	L_5	L_5	L_6	L_6	L_6	L_6
C	$\mathbf{try}(C, L_4)$			$\mathbf{L_4}$	L_4	L_4	L_3	L_3	L_6	L_6	L_6	L_6
E	$\mathbf{try}(E, L_2)$				L_2	L_4	L_3	L_3	L_6	L_6	L_6	L_6
	$\mathbf{try}(E, L_1)$				$\mathbf{L_1}$	L_4	L_3	L_3	L_6	L_6	L_6	L_6
F	$\mathbf{try}(F, L_2)$ **F**					$\mathbf{L_4}$	L_3	L_3	L_6	L_6	L_6	L_6
G	–						$\mathbf{L_1}$	L_3	L_6	L_6	L_6	L_6
M	–							$\mathbf{L_3}$	L_6	L_6	L_6	L_6
I	$\mathbf{try}(I, L_5)$								$\mathbf{L_5}$	L_5	L_5	L_6
O	–									$\mathbf{L_5}$	L_5	L_6
N	–										$\mathbf{L_5}$	L_6
D	–											$\mathbf{L_4}$
• **final levels**		$\mathbf{L_1}$	$\mathbf{L_5}$	$\mathbf{L_4}$	$\mathbf{L_1}$	$\mathbf{L_4}$	$\mathbf{L_1}$	$\mathbf{L_3}$	$\mathbf{L_5}$	$\mathbf{L_5}$	$\mathbf{L_5}$	$\mathbf{L_4}$

(b)

Figure 2: A classification constraint graph (a) and the corresponding classification process (b).

generated depends on the order of constraint evaluation.

3.2 Cyclic Constraints

For cyclic constraints the simple back-propagation of security levels is not directly applicable, and it is not clear whether the method can be adapted easily to deal with arbitrary sets of cyclic constraints. *Simple* cycles are easily handled, since they imply that all attributes in the cycle must be assigned the same security level — we can simply "replace" the cycle by a single node whose ultimate level is then assigned to each of the original attributes in the cycle. For example, we might imagine replacing the simple cycle involving attributes I, N, and O in Figure 2(a) by a single node labeled "I, N, O" and proceeding as before. However, when complex constraints are involved in a cycle, the problem becomes more challenging. Recall that a complex constraint can be solved minimally by selecting any left-hand-side attribute to be upgraded, provided that the level of no other attribute in the constraint subsequently changes. For cyclic complex constraints, it can be difficult to ensure that this requirement is satisfied. We might upgrade the level of one attribute A on the left hand side of a complex constraint only to find that a higher level is propagated through a cycle to another attribute A' in the same constraint. The constraint remains satisfied, but the resulting classification may not be minimal, since the original upgrading of A may have been unnecessary for satisfaction of the constraint.

In many cases it may be possible to determine *a priori* an order of constraint evaluation and a unique candidate for upgrading in each complex constraint that guarantees a minimal classification using back-propagation of levels through cycles. However, as the cycles become more complicated, the criteria and analysis needed for determining the attributes to be upgraded and a suitable evaluation order become more complex. The problem becomes particularly acute for cyclic complex constraints whose left hand sides are nondisjoint (for example, constraints $(\{E, F\}, M)$ and $(\{F, I\}, B)$ in Figure 2(a)), since the choice of attribute to be upgraded in one constraint may invalidate the choice made for another. Moreover, it is not generally possible to choose a single attribute in the intersection of two or more left hand sides to be upgraded for all intersecting constraints. As an example, consider three constraints whose left hand sides are $\{A, B\}$, $\{B, C\}$, and $\{A, C\}$, respectively. If all three constraints require an attribute to be upgraded, one of the constraints will necessarily have both attributes upgraded. The result in such a case can still be minimal. However, it can be far from clear whether any two attributes will do, and if not, which two should be chosen, when such intersecting constraints are entangled in a complex cycle.

Since it is difficult, at best, to ensure that no upgrading operation performed during back-propagation of levels through complex cycles will ever be invalidated, we appear to be left with essentially two alternatives: (1) augment the back-propagation approach with backtracking capabilities for reconsidering and altering upgrading decisions that result in nonminimal classifications, or (2) develop a different approach for computing minimal classifications from cyclic constraints. We would of course prefer a method that is as close as possible in computational efficiency to the simple level propagation for acyclic constraints. Thus, we reject alternative (1), since the worst-case complexity of a backtracking approach is proportional to the *product* of the sizes of the left hand sides of all constraints in the cycle. Instead, we develop a new solution approach to be applied to sets of cyclic constraints. This new approach begins with all attributes involved in a cycle at high security levels, and then attempts to lower each such attribute incrementally (in the lattice) as long as all affected constraints remain satisfied.

More specifically, assume that we are given a set of cyclic constraints and that every attribute in the cycle is initially assigned the highest classification $\top$. For each attribute A involved in the cycle, we attempt to lower the level of A, one step at a time along an arbitrary path down the lattice. At each step we check whether lowering the level of A would violate any constraints, as follows. For each constraint on A, we check whether the level of left hand side would still dominate that of the right hand side if A were to be assigned the lower level. If the constraint would still be satisfied, we simply continue. Otherwise, we check whether the level of the right hand side can also be lowered so that the constraint is again satisfied. If the right hand side is a level constant, the attempt fails. Otherwise, the right hand side is another attribute A', and we then attempt (recursively) to lower the level of A'. If, finally, the attempted lowering of A from a level l_1 to a level l_2 fails, the lowering is attempted again along a different path down the lattice from l_1. The last level for which lowering A succeeds is its final level. The result at the end of the entire process is a minimal classification for all attributes in the cycle.

Unlike the back-propagation method, which is applicable only to acyclic constraints, the incremental, forward-lowering approach is applicable to all constraints. However, it is not generally as efficient, although its complexity remains low-order polynomial. Thus, it is preferable to apply the simple back-propagation method wherever possible and reserve the forward-lowering approach for sets of cyclic constraints. The following section describes an algorithm that elegantly combines the two approaches for greatest efficiency on arbitrary sets of constraints.

4 Algorithm

At a high level, the algorithm implementing our approach consists of three main parts. In the first part, we identify sets of cyclic constraints to be evaluated with the forward lowering approach and determine the order in which attributes (sets of attributes in the case of cyclic constraints) will be considered for labeling. The second and third parts represent, respectively, the back-propagation method for acyclic constraints and the forward lowering method for cyclic constraints. These two components operate alternately according to whether or not the attribute under consideration is involved in a cycle. The procedures embodying the different parts of the approach are formally presented in Figure 3. Here we describe them informally.

The task of **Main** is to determine an order among the attributes that captures both cyclic relationships and reflects, outside cycles, the order of evaluation for back-propagation. If we interpret each edge leaving from a hypernode as a set of edges each leaving from one of the attributes in the hypernode[3], attributes involved in cyclic constraints correspond to those in strongly connected components (SCCs) of the constraint graph. Constraint cycles can therefore be identified by applying known methods for identification of SCCs. Because of the back-propagation used outside cycles, we need to identify not only the strongly connected components, but also the order in which they should be evaluated. This task is accomplished through a minor variation of known approaches to SCC computation involving two passes of the graph with a depth first search (DFS) traversal [2, 18]. The first pass (**dfs_visit**) executes a DFS on the graph, recording attributes in a stack (*Stack*) as the visit is concluded. The second pass (**dfs_back_visit**) considers attributes in the order in which they appear in *Stack*, assigning each a priority (*max_priority*) and marking it as visited. The counter *max_priority* is incremented as each such attribute is visited. For each new attribute A popped from *Stack*, the process walks the graph backward with a DFS and assigns the same priority as A to all attributes it finds still unvisited, deleting them from *Stack*. Priorities are maintained in an array, *priority*, where $\mathit{priority}[i]$ contains the set of attributes that have been assigned priority i. Priority assignments so computed satisfy the following properties: 1) each attribute has exactly one priority, 2) any two attributes appearing together in a cycle are assigned the same priority, 3) any two attributes have the same priority only if they appear together in a cycle, and 4) each attribute has a priority no greater than that of all attributes reachable from it (i.e., on which it depends). This last property ensures that the consideration of attributes in decreasing order of priorities reflects the backward traversal of the graph. As an example, consider the constraints in Figure 2. The execution of **Main** produces the following priority sets:

$$\begin{aligned}
&\mathit{priority}[1] = \{D\}\\
&\mathit{priority}[2] = \{I, O, N\}\\
&\mathit{priority}[3] = \{B, C, E, F, G, M\}\\
&\mathit{priority}[4] = \{P\}.
\end{aligned}$$

In the following we refer to each $\mathit{priority}[i]$ as priority set. In addition to computing priority assignments, **Main** initializes several variables that are used either during the DFS visits or in the actual classification process, as follows. For each complex constraint c, $\mathit{unlabeled}[c]$, initialized to the cardinality of its left hand side, keeps track of the number of attributes in the left hand side of c that are not yet definitively labeled. For each attribute A, $\mathit{Constr}[A]$ is the set of constraints whose left hand side includes attribute A, $\mathit{visit}[A]$ is used in the graph traversal to denote if A has been visited, and $\mathit{done}[A]$ is set to `true` when A becomes definitively labeled. Finally, each attribute's classification $\lambda(A)$ is initialized to $\top$. The actual computation of classification assignments is performed by **Bigloop**.

Procedure **Bigloop** considers attributes in decreasing order of priority and determines the level to be assigned to each attribute A in a priority set by considering all constraints in $\mathit{Constr}[A]$ as follows. For each constraint with right hand side definitively labeled ($\mathit{done}[\mathit{rhs}]$=`true`), the procedure determines whether the constraint must be enforced upon A and, if so, the level that A must dominate to satisfy the constraint. A constraint must be enforced upon A if it is either a simple constraint (A is the only attribute appearing in *lhs*) or if all other attributes appearing in *lhs* are definitively labeled (*unlabeled* goes to zero once A has been accounted for). The level that A must dominate to satisfy the constraint is the level of the right hand side in the case of a simple constraint. It is a minimal level that A can assume without violating the constraint (i.e., whose lub with the level of other attributes appearing in *lhs* dominates *rhs*) in the case of a complex constraint. Procedure **minlevel** computes such a level by descending the lattice along a path from A's current level and l one level at a time and stopping at the lowest level found whose direct descendants would all violate the constraint if assigned to A.[4] If all the constraints in $\mathit{Constr}[A]$ have the right hand side done, A is simply assigned the level l so computed. Intuitively, this corresponds to enforcing backward propagation. If there are constraints with right hand side not done, then, according to the computation of priorities, we are in the presence of a cycle, and level l computed as described represents only a lower bound for A. Cyclic constraints are enforced by trying to lower A to a level l'' directly below A's current level in the lattice and determine consequent lowering of other attributes necessary to maintain satisfaction of the constraints. This forward propagation of the lowering process is performed by procedure **Try**, which is called with an attribute and a level. It forward traverses the constraints in a cycle, maintaining lowerings found to be necessary in set *Tocheck*, moving them then to set *Tolower* for their later enforcement, if they do not cause any violation. In the event of a constraint violation **Try** fails immediately, returning the empty set. Otherwise, it returns the set *Tolower* containing the lowerings found

[3]Note that this correspondence can be assumed only for computing reachability and traversing the graph, not for actual constraint enforcement.

[4]In the generally assumed case of compartmented lattices (e.g., Figure 1(a)) the minimum level to be assigned to A can be computed directly without the need of walking through the lattice. The whole **else** branch of the **minlevel** procedure can in fact be substituted with the direct computation $\lambda(A) := \langle \max(\mathit{rhs}_l, \mathit{lubothers}_l), \mathit{rhs}_c - \mathit{lubothers}_c\rangle$, where rhs_l ($\mathit{lubothers}_l$ resp.) is the classification level of *rhs* (*lubothers* resp.) and rhs_c ($\mathit{lubothers}_c$ resp.) the corresponding set of categories.

Algorithm 3.1 (Minimal Classification Generation)

MAIN

```
For $A \in \mathcal{A}$ do
    $Constr[A] := \emptyset$;
    $done[A] :=$ FALSE; $visit[A] := 0$
For $l \in L$ do $done[l] :=$ TRUE; $visit[l] := 1$
For $c=(lhs,rhs) \in C$
    If $|lhs| > 1$ then $unlabeled[c] := |lhs|$
    For $A \in lhs$ do
        $Constr[A] := Constr[A] \cup \{c\}$
$Stack := \emptyset$
For $A \in \mathcal{A}$ do
    If $visit[A] = 0$ then dfs_visit($A$)
$max\_priority := 0$
For $i = 1, \ldots, |\mathcal{A}|$ do $priority[i] := \emptyset$
For $A \in \mathcal{A}$ do $visit[A] := 0$
While NOTEMPTY($Stack$) do
      $A :=$ POP($Stack$)
      If $visit$[A] = 0 then
         $max\_priority := max\_priority + 1$
         $priority[max\_priority] := \{A\}$
         dfs_back_visit($A$)
For $A \in \mathcal{A}$ do $\lambda(A) := \top$;
bigloop
```

DFS_VISIT(A)

/* Executes DFS starting from A recording in $Stack$ attribute as it finishes its visit */

```
$visit[A] := 1$
For $(lhs, rhs) \in Constr[A]$ do
    If $visit[rhs] = 0$ then dfs_visit($rhs$)
PUSH($A$,$Stack$)
```

DFS_BACK_VISIT(A)

/* Traverses the constraints backward and inserts all attributes found in the same priority set as A */

```
$visit[A] := 1$
For $(lhs, A) \in C$ do
    For $A' \in lhs$ do
        If $visit[A'] = 0$ then
            $priority[max\_priority] := priority[max\_priority] \cup \{A'\}$
            dfs_back_visit($A'$)
```

MINLEVEL(A,lhs,rhs)

/* Returns a minimal level that A can assume without violating constraint (lhs,rhs) */

```
$last := \lambda(A)$; $lubothers := \mathsf{lub}\{A' | A' \in lhs, A' \neq A\}$
If $lubothers \succeq \lambda(rhs)$ then $last := -$
else $Trylevels := \{l \mid l$ is a maximal level s. t. $last \succ l\}$
      While $Trylevels \neq \emptyset$ do
                Choose $l$ in $Trylevels$
                $Trylevels := Trylevels - l$
                if $(l \sqcup lubothers) \succeq \lambda(rhs)$ then
                   $last := l$
                   $Trylevels := \{l \mid l$ is a maximal level s. t. $last \succ l\}$
return $last$
```

BIGLOOP

/* Considers components in decreasing order of priorities and computes a minimal level for each attribute in them. A node A is done ($done[A] :=$ TRUE), when its assignment $\lambda(A)$ is set and will not change. The set of immediate descendents in a lattice is recorded in variable DSet. */

```
For $p := max\_priority, \ldots, 1$ do
   For $A \in priority[p]$ do
      $done[A] :=$ TRUE
      $l := -$
      For $c=(lhs,rhs) \in Constr[A]$ do
         If $|lhs| > 1$ then $unlabeled[c] := unlabeled[c] - 1$
         If $done[rhs]$ then
            case $|lhs|$ of
               1:   $l := l \sqcup \lambda(rhs)$
               >1: If $unlabeled[c] = 0$ then
                          $l := l \sqcup$ minlevel($A$,$lhs$,$rhs$)
         else $done[A] :=$ FALSE
      If $done[A]$ then $\lambda(A) := l$
      else $DSet := \{l' \mid l'$ is a maximal level, $\lambda(A) \succ l' \succeq l\}$
            While $DSet \neq \emptyset$
                 Choose $l''$ in $DSet$
                 $DSet := DSet - l''$
                 $Lower :=$ try($A, l''$)
                 If $Lower \neq \emptyset$ then
                    For $(A', l') \in Lower$ do $\lambda(A') := l'$
                    $DSet := \{l' \mid l'$ maximal level, $\lambda(A) \succ l' \succeq l\}$
            $done[A] :=$ TRUE
```

TRY(A,l)

/* Returns a set of attribute-level pairs which together with the current assignment λ forms a (perhaps non-minimal) solution to the constraints, unless there is no such set of pairs, in which case Try returns $\emptyset$. That is, Try returns $\emptyset$ if $\lambda(A) = l$ (transitively) violates the constraints, given the current assignment λ */

```
$Tocheck := \{(A, l)\}$
$Tolower := \emptyset$
Repeat
   Choose $(A', l') \in Tocheck$
   $Tocheck := Tocheck - \{(A', l')\}$
   $Tolower := Tolower \cup \{(A', l')\}$
   For $(lhs, rhs) \in Constr[A']$ do
      $level := -$
      For $A'' \in lhs$ do
           If $\exists (A'', l'') \in Tolower$ then
              $level := level \sqcup l''$
           else $level := level \sqcup \lambda(A'')$
      case $done[rhs]$ of
        TRUE:  If $\neg(level \succeq \lambda(rhs))$ then return $\emptyset$
        FALSE: If $\neg(level \succeq \lambda(rhs))$ then
                  $newlevel := \lambda(rhs) \sqcap level$
                  If $\exists (rhs, l'') \in (Tolower \cup Tocheck)$ then
                     If $\neg(newlevel \succeq l'')$ then
                        $newlevel := l'' \sqcap newlevel$
                        If $(rhs, l'') \in Tolower$ then
                           $Tolower := Tolower - \{(rhs, l'')\}$
                        else $Tocheck := Tocheck - \{(rhs, l'')\}$
                        $Tocheck := Tocheck \cup \{(rhs, newlevel)\}$
                  else $Tocheck := Tocheck \cup \{(rhs, newlevel)\}$
until $Tocheck = \emptyset$
return $Tolower$
```

Figure 3: Algorithm for computing a minimal classification.

to be necessary. Hence, if the returned set is not empty, **Bigloop** lowers the attributes as determined and restarts the process, trying to lower A to a level just below the last level tried. If, instead **Try** fails, another level directly dominated by the last one that returned success is tried. The process is repeated until all direct descendants of the level to which A has been lowered in the last pass return a failure.

Note that in the forward-lowering process, the level to be pushed forward may change and become either higher or lower because of complex constraints. The level can increase when traversing a complex constraint, because in this case we require only that the right hand side is dominated by (i.e, lowered to) the level of the *lub* of all the attributes in the left hand side. The level can also decrease when, traversing a complex constraint, we would require *rhs* to be dominated by (lowered to) a level incomparable to its current level or the level recorded for it in either *Tocheck* or *Tolower*. In this case, the process can succeed only if the attribute is dominated by both levels, that is, if it can be lowered to their greatest lower bound. We therefore lower the attribute to this level and propagate it forward.

Example 4.1 Figure 2(b) illustrates the execution of the approach on the constraints of Figure 2(a). The left column lists attributes in the order in which they are considered and illustrates how their levels (and those of attributes in the same priority set) change. An **F** on the side of a **Try** call indicates a failure. Traversing down a lattice is assumed to be performed by considering direct descendants in left-to-right order. Levels indicated in bold face are the levels of attributes at the time they become *done*. The bottom line reports the final (minimal) levels computed. Note that the table in Figure 2(b) is only for illustration and does not correspond to any data structure maintained by the algorithm.

5 Correctness and complexity analysis

In this section we state the correctness of our approach and discuss its complexity. Proof sketches of the theorems appear in in the Appendix.

Theorem 5.1 (Correctness) *Algorithm 3.1 solves* MIN-LATTICE-ASSIGNMENT. *That is, given a set C of classification constraints over a set $\mathcal{A}$ of attributes and a security lattice $\mathcal{L} = (L, \succeq)$, Algorithm 3.1 generates a minimal classification mapping $\lambda : \mathcal{A} \mapsto L$ that satisfies C.*

Complexity In the complexity analysis we adopt the following notational conventions with respect to a given instance $(\mathcal{A}, \mathcal{L}, C)$ of MIN-LATTICE-ASSIGNMENT: $N_{\mathcal{A}}$ $(= |\mathcal{A}|)$ denotes the number of attributes in $\mathcal{A}$; N_L $(= |L|)$ denotes the number of security levels in $\mathcal{L}$; N_C $(= |C|)$ denotes the number of constraints in C; $S = \sum_{(lhs,rhs)\in C}(|lhs| + 1)$ denotes the total size of all constraints in C; H denotes the height of $\mathcal{L}$; B denotes the maximum number of immediate predecessors ("branching factor") of any element in $\mathcal{L}$; c denotes the maximum cost of computing the lub of any two elements in $\mathcal{L}$. Note that, for any lattice $\mathcal{L}$, BH is no greater than the size of $\mathcal{L}$ (number of elements + size of the immediate successor relation).

Theorem 5.2 (Complexity) *Algorithm 3.1 solves any instance $(\mathcal{A}, \mathcal{L}, C)$ of* MIN-LATTICE-ASSIGNMENT *in $O(N_{\mathcal{A}}SH^2Bc)$ time, and, if C is acyclic, in $O((S+N_CHB)c)$ time. Therefore,* MIN-LATTICE-ASSIGNMENT *is solvable in polynomial time.*

Note, in particular, that the time taken by Algorithm 3.1 is linear in the size of the constraints for acyclic constraints, and no worse than quadratic for cyclic constraints. Assuming a fixed lattice, a trivial lower bound for MIN-LATTICE-ASSIGNMENT is $\Omega(S)$, and thus, Algorithm 3.1 is optimal for acyclic constraints. Whether the complexity for the cyclic case can be improved to linear in the size of the constraints remains an open question. However, the complexity for the cyclic case is truly worst case — it assumes that the entire constraint set forms a single cyclic component, which should not occur in practice. For any instance of the problem, the acyclic complexity analysis applies to all acyclic portions of the constraint set. The higher price is paid only for cyclic constraints, which will typically include only a small portion of the input constraint set.

The cost of lattice operations An important practical consideration is the efficiency of lattice computations. Recent work [17] has shown that constant-time testing of partial orders can be accomplished through a data structure requiring $O(n\sqrt{n})$ space and $O(n^2)$ time to construct, where n is the number of elements in the poset. Encoding techniques [5, 6] are known that, enable near constant-time computation of lubs/glbs, so that c in the above analysis can be taken as constant, at the expense of additional preprocessing time. In practice, one would expect to use the same security lattice over many different instances of MIN-LATTICE-ASSIGNMENT, so that the additional preprocessing cost for lattice encoding is less of a concern. Finally, we note that the generally considered security lattices with access classes represented by pairs *classification* and a set of *categories* can be efficiently encoded as bit vectors that enable fast testing of the dominance relation and lub and glb computations. The limited number of levels (16) and categories (64) required by the standard [4] allows the encoding of any security level in a small number of machine words, effectively yielding constant-time lattice operations.

6 Upper bound constraints and arbitrary partial orders

The results presented thus far are based on the consideration of lower-bound constraints and the assumption of classification levels forming a lattice. Here we show how the results can be extended to include upper-bound constraints and incomplete lattices. We then show that relaxing the assumption to allow arbitrary partial orders leads to intractability.

Upper bound Constraints The form of classification constraints allowed by Definition 2.1 permits specification only of *lower* bounds on the classifications of attributes or collections of attributes, and thus, are geared toward *restricting* the visibility of information. It may also be desirable to specify *upper* bounds as well, to *guarantee* visibility of some information to certain classes of users. Thus, we extend the definition of classification constraint to allow constraints of the form $l \succeq \lambda(A)$, where l is a security level (constant) and A is an attribute.

The most obvious effect of allowing upper bound constraints is that they introduce the potential for inconsistency in the constraint set, the most trivial example being $\{A \succeq \top, \bot \succeq A\}$ (assuming that $\top$ and $\bot$ are distinct). Such inconsistencies can be detected easily by "pushing" upper bounds through the constraint graph until a violation of the partial order relation is found. If no such violation is discovered, we can, through this same process, determine a firm upper bound on every attribute in the constraint set, which can serve as a starting point for a slightly modified version of Algorithm 3.1. Here we briefly outline both this new preprocessing phase and the algorithm modification.

Let C be a set of classification constraints containing possibly both upper and lower bound constraints. (Observe that, in the graph of C, security level nodes are no longer necessarily leaves.) Initially, each attribute is assigned level $\top$ as before. Then, for each security level involved in an upper bound constraint, we propagate the level through the graph. Where multiple upper bounds arrive at a node, their glb is taken. When propagating upper bounds through a node involved in a complex constraint, the lub of the lhs of the constraint is propagated. Inconsistencies are detected upon arriving at security level nodes. If the level of the incoming upper bound does not dominate the level of such a node, there is an inconsistency. If no inconsistencies are found, each attribute will be labeled at its maximum allowed level. This preprocessing phase can be accomplished in $O(Sc)$ time, where c now represents the cost of one lub or glb operation.

Now, if no inconsistencies are discovered, we can compute a correct and minimal solution for the (lower bound) constraints starting from the upper bounds derived in the preprocessing phase. However, this computation requires a modification to **BigLoop**. In the absence of upper bound constraints, we are able to delay the solving of complex constraints (via **Minlevel**), since, as long as at least one attribute on the lhs is known to be labeled at $\top$, any other attribute in the lhs could assume any level without violating that constraint. But when upper bound constraints are processed, the initial level of any attribute may be lower than $\top$, and therefore the satisfaction of complex constraints cannot be assumed. The solution to this problem is to invoke **Minlevel** for each attribute in each of its complex constraints. For acyclic constraints, this has the effect of increasing the time complexity to $O(SHBc)$, but for the more general (cyclic) case, the complexity remains $O(N_{\mathcal{A}}SH^2Bc)$.

Semi-lattices It can happen in practice that the partial order of security levels does not form a complete lattice. There may be no top element when it is intended that no user or class of users can have visibility over all information. Similarly, there may be no bottom element in environments where no information is truly unclassified. Such semi- or partial-lattices pose no particular problem for our approach. If a semi-lattice has no top element, we simply add a dummy $\top$, and proceed with the algorithm as before. When the algorithm has completed, if any attribute remains at $\top$, it is an indication that there is no solution to the constraints (or, more precisely, that the constraints require that any such attribute be visible to no one). If a semi-lattice has no bottom element, we can add a dummy $\bot$ and run the algorithm as before. When the algorithm has completed, if any attribute is labeled at $\bot$, it is simply an indication that there was no effective constraint on the attribute (which might be flagged as an error to indicate incompleteness in the input constraint set).

Arbitrary Partial Orders Although lattices need not be complete for our approach to work, it appears to be crucial that the partial order of security levels be at least a partial lattice, where any two levels that have an upper bound must have a least upper bound. If the set of security levels may be an arbitrary poset, the problem of determining a minimal classification that satisfies all constraints appears to become intractable. We define the problem MIN-POSET similarly to MIN-LATTICE-ASSIGNMENT, except that the partial order is not restricted to be a lattice. The following theorems, whose proof is reported in Appendix A states the intractability of the MIN-POSET problem.

Theorem 6.1 MIN-POSET *is* NP*-complete.*

7 Conclusions

We have examined the problem of computing an assignment of security levels to database attributes from a set of classification constraints. The constraints we consider permit specification of relationships between the security levels of a set of one or more attributes and the level of another attribute or an explicit level. In contrast to previous proposals investigating the NP-hard problem of determining optimal solutions (with respect to some cost measure), we provide an efficient algorithm for computing one solution with (pointwise) minimal information loss. Our approach efficiently handles complex cyclic constraints and guarantees a minimal solution in all cases in quadratic time, but also provides linear time performance for the common case of acyclic constraints.

References

[1] S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison-Wesley, 1995.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[3] Steven Dawson, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Specification and Enforcement of classification and Inference Constraints. In *Proc. of the 20th IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[4] Dep. Defense, National Computer Security Center, Standard DOD 5200.28-STD, 1985. *Department of Defense Trusted Computer System Evaluation Criteria.*

[5] D.D. Ganguly, C.K. Mohan, and S. Ranka. A Space-and-Time-Efficient Codeing Algorithm for Lattice Computations. *Transactions on Knowledge and Data Engineering*, 6(5):819–829, 1994.

[6] H.Aït-kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[7] S. Jajodia and C. Meadows. Inference Problems in Multilevel Secure Database Management Systems. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security - An Integrated Collection of Essays*, pages 570–584. IEEE Computer Society Press, 1995.

[8] S. Jajodia and R. Sandhu. Toward a Multilevel Secure Relational Data Model. In *Proc. of the 1991 ACM SIGMOD Conference*, pages 50–59, May 1991.

[9] T.F. Lunt. Aggregation and Inference: Facts and Fallacies. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 102–109, Oakland, May 1989.

[10] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView Security Model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990.

[11] M. Morgenstern. Security and Inference in Multilevel Database and Knowledge-Base Systems. In *Proc. of the 1987 ACM SIGMOD Conference*, pages 357–373, San Francisco, CA, May 1987.

[12] X. Qian. View-Based Access Control with High Assurance. In *Proc. of the 1996 IEEE Symposium on Security and Privacy*, pages 85–93, May 1996.

[13] X. Qian and T.F. Lunt. A MAC Policy Framework for Multilevel Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):1–14, February 1996.

[14] R. Sandhu and F. Chen. The Multilevel Relational (MLR) Data Model. *ACM Transactions on Information and System Security*, 1(1), November 1998.

[15] M.E. Stickel. Elimination of Inference Channels by Optimal Upgrading. In *Proc. of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 168–174, Oakland, CA, May 1994.

[16] T.A. Su and G. Ozsoyoglu. Controlling FD and MVD Inferences in Multilevel Relational Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):474–485, December 1991.

[17] M. Talamo and P. Vocca. A Data Structure for Lattice Representation. *Theoretical Computer Science*, 175:373–392, 1997.

[18] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[19] M. Winslett, K. Smith, and X. Qian. Formal Query Languages for Secure Relational Databases. *ACM Transactions on Database Systems*, 19(4):626–662, December 1994.

A Proof sketches

Correctness of Algorithm 3.1 We first establish several lemmas used in the proof of the main theorem. Lemma A.1 uses the priority ordering of strongly connected components (SCCs) to show that arguments about the satisfaction of generated level assignments can be made locally. That is, it establishes that, if any changes to a solution mapping that are limited to the attributes in an SCC result in satisfaction of the immediate constraints on those attributes, then the modified mapping remains a solution for all constraints.

Lemma A.1 *For a given set C of constraints and priority p, let λ be an assignment of levels to attributes that satisfies C and λ' be an assignment such that $\lambda \succeq \lambda'$ and that differs from λ only on attributes in* priority$[p]$. *Let C_p denote the set of direct constraints on attributes of priority p, that is, $C_p = \{(lhs, rhs) \in C \mid lhs \cap \text{priority}[p] \neq \emptyset\}$. Then, λ' satisfies C if and only if λ' satisfies C_p.*

Proof: (sketch)

(If): Assume that λ' satisfies C_p. Let $c = (lhs, rhs)$ be an arbitrary constraint in C. If $c \in C_p$, then by assumption, λ' satisfies c. Otherwise, $c \notin C_p$, so $lhs \cap priority[p] = \emptyset$, and thus, $\text{lub}\{\lambda'(lhs)\} = \text{lub}\{\lambda(lhs)\}$. Now, $\lambda(rhs) \succeq \lambda'(rhs)$ and $\text{lub}\{\lambda'(lhs)\} = \text{lub}\{\lambda(lhs)\} \succeq \lambda(rhs) \succeq \lambda'(rhs)$, and hence, λ' satisfies c.

(Only if): If λ' satisfies C, λ' satisfies any subset of C.

■

The following lemma shows that any changes to a solution λ resulting from the output of procedure **Try** in Algorithm 3.1 preserves λ as a solution.

Lemma A.2 *Let AS be the set of pairs of the form (A', l') returned by* **Try**(A, l). *If λ satisfies C just before* **Try**(A, l) *is called, then the assignment obtained by replacing $\lambda(A')$ with $\lambda(A') = l'$ for all $(A', l') \in AS$ also satisfies C.*

Proof: (sketch) If **Try** returns $\emptyset$ the lemma is trivially satisfied. Otherwise, consider an arbitrary pair (A', l') in the set *Tolower* returned by **Try**. Since any pair is added to *Tolower* only upon removal from *Tocheck*, it must be that one iteration of the repeat-loop is run with (A', l'). During that run, every constraint on A' is checked. Furthermore, each check must succeed, since otherwise **Try** fails, returning $\emptyset$. Note that every attribute in every pair in *Tolower* (and *Tocheck*) has the same priority, since any attribute of higher priority is already marked *done* (and thus, cannot be added to the set *Tocheck*), and because of priority ordering, no attribute of lower priority is reachable. Now, every constraint on an attribute with the same priority as A' is either explicitly checked (using the levels specified for those attributes in *Tolower* and λ for every other), or is known to be satisfied (by transitivity from a constraint successfully checked). Hence, every constraint on attributes of the priority of A' is satisfied when $\lambda(A')$ is replaced by $\lambda(A') = l'$ for every $(A', l') \in$ *Tolower*, and by Lemma A.1, C is also satisfied. ■

Theorem 5.1 (Correctness) *Algorithm 3.1 solves* MIN-LATTICE-ASSIGNMENT. *That is, given a set C of classification constraints over a set $\mathcal{A}$ of attributes and a security lattice $\mathcal{L} = (L, \succeq)$, Algorithm 3.1 generates a minimal classification mapping $\lambda : \mathcal{A} \mapsto L$ that satisfies C.*

Proof: (sketch)

Satisfaction: To show that **BigLoop** always produces an assignment λ that satisfies C, we use an inductive argument on the outermost loop of **BigLoop**. For the basis, note that λ initially assigns $\top$ to every attribute, which trivially satisfies all constraints of the allowed form (Definition 2.1). For the induction step we need to show that, if λ is a solution at the start of an iteration of the outermost loop, then λ is also a solution at the end of that iteration. By Lemma A.1 it suffices to show that (1) λ at the end of any iteration differs from λ at the start only on attributes of a given priority p, (2) the level assigned by λ to any attribute is never raised, and (3) all direct constraints on attributes of priority p are satisfied at the end of any iteration.

Let p be the priority in the outermost loop of **BigLoop** and S be the set $priority[p]$. There are two cases:

- $|S| = 1$: Let A be the sole attribute in S, and let $c = (lhs, rhs)$ be an arbitrary constraint in

Constr$[A]$. Note that l, which is initially $\bot$, will eventually hold the level to be assigned to A. Now, rhs is either a security level or is an attribute of higher priority than A. In either case, $done[rhs]$ = TRUE, and so there are two cases to consider based on $|lhs|$. If $|lhs| = 1$, l is assigned the lub of its current value and $\lambda(rhs)$, so that $l \succeq \lambda(rhs)$, which, if assigned to A, will satisfy c. Otherwise, $|lhs| > 1$, and we consider the value of $unlabeled[c]$. If $unlabeled[c] > 0$, there is at least one attribute $A'(\neq A) \in lhs$ such that $done[A']$ = FALSE, $\lambda(A') = \top$, and thus, for any value of l assigned to A, c is trivially satisfied. If $unlabeled[c] = 0$, **Minlevel** computes a minimal level l' for A such that c is satisfied, l is assigned the lub of its current value and l', and thus, c is satisfied if l is assigned to A. Note that, since $\mathcal{L}$ is a lattice, the (unique) lub of any two levels always exists.

After processing each constraint on A, l is such that $\lambda(A) = l$ satisfies all constraints processed so far, since l is assigned an upper bound of its current value and the value needed to satisfy the constraint just processed. After all constraints on A are processed, $\lambda(A)$ is set to l, and thus all constraints on A are satisfied (3). Note that (at most) the level of A is modified (1), and since $\lambda(A)$ was initially $\top$, if its assignment changed, it could only have been lowered (2). Thus, by induction hypothesis, λ satisfies C.

- $|S| > 1$: We extend the inductive argument to the second-level loop (**For** $A \in priority[p]$), and show that λ satisfies C at the end of each iteration of this inner loop. Let A be an arbitrary attribute in S. Consider *Constr*$[A]$. If every $(lhs, rhs) \in$ *Constr*$[A]$ is such that $done[rhs]$ = TRUE, the argument for case $|S| = 1$ applies. Otherwise, there is at least one $(lhs, rhs) \in$ *Constr*$[A]$ such that $done[rhs]$ = FALSE. So, after processing each $c \in$ *Constr*$[A]$, $done[A]$ = FALSE, and we proceed from the initialization of *DSet*. Now, by an argument similar to that of case $|S| = 1$, l holds a lower bound on the level that may be assigned to A, and *DSet* is initialized to the set of levels immediately below $\lambda(A)$ and that dominate l. We again extend the inductive argument to the while-loop to show that λ satisfies C at the end of any iteration of the while loop. Observe that, before entering the while-loop, λ satisfies C because no assignments have been modified up to this point in the enclosing for-loop. In the while-loop, either **Try** fails for every $l'' \in DSet$, or it succeeds for one of them. If it fails for all, no assignments in λ are modified, and thus, C remains satisfied. Otherwise, by Lemma A.2, **Try** returns a set of pairs of the form (A', l'), where $A' \in priority[p]$, $\lambda(A') \succeq l'$, and such that replacing $\lambda(A')$ by $\lambda(A') = l'$ for all such A' satisfies all constraints on attributes in $priority[p]$. The while-loop concludes by making this replacement and resetting *DSet* to levels immediately below $\lambda(A)$. Hence, λ satisfies C at the end of the current iteration of the while-loop, and by the extended inductive arguments, λ also satisfies C at the end of the enclosing for-loop and at the end of the outermost loop.

Minimality: To prove minimality of the generated assignments we use a similar inductive argument to show that, at the end of any iteration of the outermost loop, any attribute A for which $done[A]$ = TRUE has been assigned a minimal level that satisfies its constraints. For the basis, observe that for every $l \in L$, $\lambda(l) = l$ and $done[l]$ = TRUE at the start of the outermost loop. For the induction step we need to show that, if any attribute marked *done* at the start of any iteration of the outermost loop has been assigned its minimal satisfying level, then any attribute marked *done* at the end of that iteration has as well.

As before, let p be the priority in the outermost loop of **BigLoop** and S be the set $priority[p]$. We consider two cases:

- $|S| = 1$: Let A be the sole attribute in S. From the satisfaction argument, we know that l is a satisfying assignment for A. To see that $\lambda(A) = l$ is a minimal satisfying assignment, first observe that, for any $c = (lhs, rhs) \in$ *Constr*$[A]$, $done[rhs]$ = TRUE, so by induction hypothesis, $\lambda(rhs)$ is minimal. Second, l was computed as the least upper bound of only those minimal levels needed to make $\mathsf{lub}\{\lambda(lhs)\} \succeq \lambda(rhs)$ true for all constraints on A. By definition of least upper bound, l is the lowest level that does so. At the end of the iteration $\lambda(A) = l$, and $done[A]$ = TRUE.
- $|S| > 1$: Let A be an arbitrary attribute in S. As discussed in the satisfaction argument, if all constraints on A are such that $done[rhs]$ = TRUE, then the argument for case $|S| = 1$ applies, so we assume that there is at least one constraint on A for which $done[rhs]$ = FALSE. Using an inductive argument similar to that of case $|S| = 1$ we know that l is a lower bound on any minimal assignment for A; that is, A must dominate l in any minimal solution. Now, let l' be the level assigned to A when processing of A is completed (marked *done* after the while-loop in **BigLoop**). Suppose that $\lambda(A) = l'$ is not minimal for A, that is, there exists a solution λ' for C such that $\lambda \succ \lambda'$ and $\lambda'(A) = l''$ where $l' \succ l''$. Consider the set *DSet* of levels immediately below l' in the lattice. **Try** must have failed on each of these, resulting in the assignment of l' to A. At least one of these levels must dominate l'', so let $\hat{l}$ be an arbitrary one of these levels such that $\hat{l} \succeq l''$. Consider the run of **Try** that failed when trying to lower A to $\hat{l}$. Since **Try** fails only when a constraint is violated, it follows that there exists some constraint c (on an attribute of the same priority as A) that requires $\lambda(A) \succ \hat{l}$. Since we are dealing with a lattice, and $\lambda' \succ \lambda$ (where λ is the set of assignments at the time **Try** failed on $(A, \hat{l})$), the same constraint c must also require $\lambda'(A) \succ \hat{l}$. Thus, $\lambda'(A) = l''$ is not a solution for A, so λ' is not a solution for C.

Termination: There are two aspects to termination that are not obvious. First, the while-loop at the end of **BigLoop** terminates because *DSet* is finite, and in each iteration every level in *DSet* is strictly dominated by any level in the preceding iteration. Thus, as long as **Try** terminates, the while-loop will terminate, because either the bottom of the lattice is reached or because every level tried in one iteration fails.

Second, it is not immediately obvious that the repeat-loop in **Try** terminates. Note that it continues as long as the set *Tocheck* is not empty. In each iteration of the loop one pair is removed from *Tocheck* and added to *Tolower*. However, for any attribute, there can be at most one pair involving that attribute in either *Tocheck* or *Tolower*. It is possible that, for some pair $(A, l) \in$ *Tolower*, a pair (A, l') will be added to *Tocheck*. If so, l must strictly dominate l', so the number of times a pair involving the same attribute may be entered into *Tocheck* is bounded by the height of the lattice.

■

Complexity analysis In the complexity analysis we adopt the following notational conventions with respect to a given instance $(\mathcal{A}, \mathcal{L}, C)$ of MIN-LATTICE-ASSIGNMENT: $N_{\mathcal{A}}$ $(= |\mathcal{A}|)$ denotes the number of attributes in $\mathcal{A}$; N_L $(= |L|)$ denotes the number of security levels in $\mathcal{L}$; N_C $(= |C|)$ denotes the number of constraints in C; $S = \sum_{(lhs,rhs)\in C}(|lhs| + 1)$ denotes the total size of all constraints in C; H denotes the height of $\mathcal{L}$; B denotes the maximum number of immediate predecessors ("branching factor") of any element in $\mathcal{L}$; c denotes the maximum cost of computing the lub or glb of any two elements in $\mathcal{L}$. Note that, for any lattice $\mathcal{L}$, BH is no greater than the size of $\mathcal{L}$ (number of elements + size of the immediate successor relation).

Theorem 5.2 (Complexity) *Algorithm 3.1 solves any instance $(\mathcal{A}, \mathcal{L}, C)$ of* MIN-LATTICE-ASSIGNMENT *in $O(N_{\mathcal{A}}HBSc)$ time, and, if C is acyclic, in $O((S+N_CHB)c)$ time. Therefore,* MIN-LATTICE-ASSIGNMENT *is solvable in polynomial time.*

Proof: For the analysis, we consider two cases: (1) C is acyclic, and (2) C is cyclic. We begin by noting that the preprocessing steps (common to both cases) in **Main**, apart from **DFS_Visit** and **DFS_Back_Visit**, require (in total) time proportional to $S + N_L$. **DFS_Visit** and **DFS_Back_Visit** themselves are simply a minor adaptation of Tarjan's linear-time SCC computing algorithm [18], and require time proportional to S. Thus, the time complexity of the preprocessing phase is $O(S + N_L)$. It remains to determine the complexity of **BigLoop**. For **BigLoop** note that the effect of the three nested for-loops is to consider every attribute in each of its constraints, which requires no more than S iterations of the innermost loop, while the containing loop iterates $N_{\mathcal{A}}$ times.

In the acyclic case, note that every attribute is its own SCC. When considering any attribute A in **BigLoop**, then, the computation of the level of any attribute appearing on the rhs of any constraint on A will have been completed (*done*[*rhs*] is always true), and the *DSet* computation and while-loop are never performed. Thus, apart from constant-time initializations in the second for-loop, the only cost to consider for the acyclic case is that of the innermost for-loop. For each constraint, either a lub operation is performed, or possibly a lub operation and a call to **Minlevel**. Note, however, that **Minlevel** is called only once for each complex constraint (when its *unlabeled* count reaches zero). Overall, no more than S iterations of the innermost for-loop compute a lub, and no more than N_C iterations involve **Minlevel**. The naive algorithm given for **Minlevel** first performs a number of lub operations proportional to the size of the lhs of the given constraint. The remainder of **Minlevel** considers overall at most HB security levels, each involving a lub operation. The time complexity of **Minlevel**, then, is $O((|lhs| + HB)c)$. For the N_C iterations involving **Minlevel**, the total cost is $O((S + HBN_C)c)$. The total cost of the remaining iterations is $O(Sc)$, and hence, the overall time complexity of **BigLoop** in the acyclic case is $O((S + HBN_C)c)$.

For cyclic constraints we take the worst case, where all attributes are in the same SCC. The cost due to the innermost for-loop of **BigLoop** cannot be greater than that of the acyclic case. In the containing loop (the loop over attributes), the while-loop may execute for every attribute in the SCC. Like **Minlevel**, the while-loop considers at most HB security levels, each involving the **Try** computation. In the worst case, **Try** processes the constraints for all attributes in the SCC. More precisely, it processes the constraints of every attribute in the SCC not marked *done*. The number of such attributes decreases by one after each invocation of **Try**, but on average, **Try** may process as many as half the constraints involved in the SCC. Now, it can happen that, for some pair $(A, l) \in$ *Tolower* and level l', (A, l) is removed from *Tolower* and (A, l') added to *Tocheck*, implying the preprocessing of constraints on A. For any attribute, this reintroduction into *Tocheck* can happen at most H times[5], since l' must be strictly lower than l. For each constraint considered, the lub of all attributes in the lhs is computed, requiring time proportional to $|lhs| \cdot c$. Assuming suitable data structures for constant-time operations involving *Tolower* and *Tocheck*, the only remaining nonconstant cost comes from at most two glb operations. The time complexity of **Try**, then, is $O(HSc)$, and that of the while-loop in **BigLoop** is $O(H^2BSc)$. Over all attributes in the SCC, the time complexity of **BigLoop** due to the while-loop is $O(N_{\mathcal{A}}H^2BSc)$, which dominates the cost due to the innermost for-loop of **BigLoop**. ■

Minimal assignment in a POset We define the problem MIN-POSET similarly to MIN-LATTICE-ASSIGNMENT, except that the partial order is not restricted to be a lattice and it is stated as a decision problem. Given a partial order $(P, \geq)$ and a set of constraints C, each constraint taking one of three forms: $A \geq A'$, $A \geq l$, $\mathsf{lub}\{A_1, \cdots, A_k\} \geq A$, where the As are variables, and l is a constant drawn from P, is there is an assignment from variables to members of P that satisfies all the constraints C, and which is minimal?

Theorem 6.1 [MIN-POSET *is* NP-*complete.*]
It is easy to see that this problem is in NP, since one may simply guess an assignment of levels (lattice values) to variables, and check that every constraint in C is satisfied.

To show that this problem is NP-hard, we give a reduction from 3-SAT. We begin with the empty set C, and for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, we add the element named C_i to C, and further add 7 more elements to A, one for each truth assignment which satisfies the clause. For convenience, we name these 7 elements by simply concatenating the names of the clauses with the names of the variables they contain, using overbars to denote negation: "$C_iP_{i1}P_{i2}P_{i3}$", "$C_iP_{i1}P_{i2}\overline{P_{i3}}$", "$C_iP_{i1}\overline{P_{i2}}P_{i3}$", etc. For each propositional variable P_j, we add three elements to C, named "P_j", "P_j^+", and "P_j^-". Intuitively, these stand for the j-th proposition being undecided, true, and false, respectively.

[5]More precisely, it can happen at most $\min(H, R)$ times, where R is the maximum number of constraints with a common attribute on the rhs.

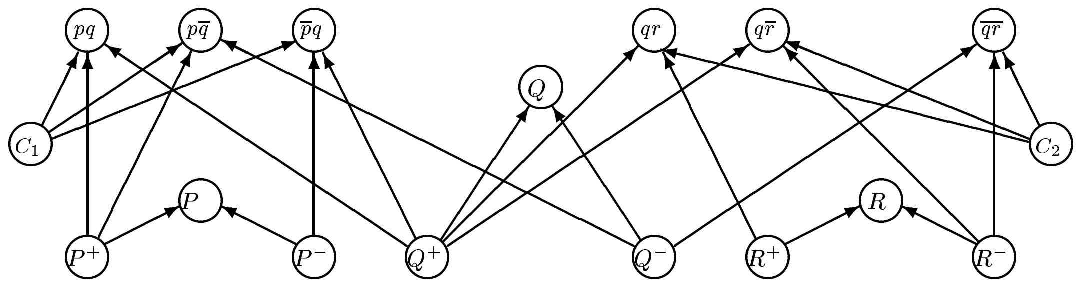


Figure 4: Poset for $(P \vee Q) \wedge (Q \vee \neg R)$.

With the above set of constants, we define a partial order relation $\geq$ on them as follows. We define the relation R_{prop} to include, for each proposition P_i, $P_i^+ \geq P_i$ and $P_i^- \geq P_i$. We define the relation R_{clause} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$ occurring in the 3-SAT problem, and each truth assignment which satisfies the clause, $C_i \geq C_iP_{i1}P_{i2}P_{i3}$. We also define the relation R_{true} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, and each proposition in that clause P_{ij}, a relation $P_{ij}^+ \geq C_iP_{i1}P_{i2}P_{i3}$ for each of the 3 or 4 clause elements which correspond to P_{ij} being true. Similarly, we define the relation R_{false} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, and each proposition in that clause P_{ij}, a relation $P_{ij}^- \geq C_i\overline{P_{i1}P_{i2}P_{i3}}$ for each of the 3 or 4 clause types which correspond to P_{ij} being false. The final partial order of interest will be $(A, R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false})$. The partial order has height one, and contains 8 $(= 2^3)$ elements for each 3-SAT clause, plus three elements for each proposition. Figure 4 displays the partial order produced for the SAT problem $(P \vee Q) \wedge (Q \vee \neg R)$. Clauses of length two were used, and the name pq was used in place of C_1pq, for example, in Figure 4 to improve readability.

We use a set of variables, one wp_j and one wu_j for each proposition P_j, and one wc_j for each clause $Clause_j$. We define a set of inequations C_{clause} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, the constraint $C_i \geq wc_i$, and for each proposition P_{ij} in that clause, $wp_{ij} \geq wc_i$. We also define a set of inequations C_{prop} to include, for each proposition P_i, $wp_i \preceq wu_i$ and $P_i \preceq wu_i$. Thus there are four constraints in C_{clause} per 3-SAT clause, and two constraints in C_{prop} for each proposition. Continuing with our simple example, $(P \vee Q) \wedge (Q \vee \neg R)$, the inequations $C_{clause} = \{C_1 \preceq wc_1, wp_p \preceq wc_1, wp_q \preceq wc_1, C_2 \preceq wc_2, wp_q \preceq wc_2, wp_r \preceq wc_2\}$, and $C_{prop} = \{wp_p \preceq wu_p, wp_q \preceq wu_q, wp_r \preceq wu_r, P \preceq wu_p, Q \preceq wu_q, R \preceq wu_r\}$.

We claim that the MIN-POSET problem given by the partial order $(A, R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false})$, with the constraints $C_{prop} \cup I_{clause}$ has a minimal solution if and only if the original 3-SAT problem has one. This may be observed by noting that every wc_i must be assigned some $C_iP_{i1}P_{i2}P_{i3}$, since wc_i must be greater than C_i and some propositions. Also, the only $C_iP_{i1}P_{i2}P_{i3}$ which exist in A correspond to assignments of propositions which satisfy the clause. Further, wu_j must be assigned P_j, and wp_j must be assigned either P_j^+ or P_j^-. We claim there is a correspondence between a proposition P_j being assigned true (or false, resp.) in the 3-SAT problem, and w_j being assigned P_j^+ (P_j^-, resp.) in the POL-SAT problem. Thus one may see that a solution to the 3-SAT may be derived from any solution to the constructed POL-SAT problem and vice-versa.