

Maximizing Sharing of Protected Information¹

Steven Dawson¹, Sabrina De Capitani di Vimercati², Patrick Lincoln¹, Pierangela Samarati³

(1) *SRI International, Menlo Park, CA 94025, USA*

(2) *Dipartimento di Elettronica e Automazione, Università di Brescia, Italy*

(3) *Dipartimento di Tecnologie dell'Informazione, Università di Milano, Italy*

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. Without support for coping against these attacks, sensitive information can be put at risk because of release of other (less sensitive) related information. The ability to protect information disclosure against such improper leakage would be of great benefit to governmental, public, and private institutions, which are, today more than ever, required to make portions of their data available for external release.

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 overclassified. We present an approach to the solution of this problem and give an algorithm implementing it which is linear in simple cases, and quadratic in the general case. We also analyze a variant of the problem that is NP-complete.

Key Words: security, privacy, data classification, data inference, constraint solving, lattice

1. INTRODUCTION

Information has become the most important and demanded resource. We live in an internetworked society that relies on the dissemination and sharing of information in the private as well as in the public and governmental sectors. This situation is witnessed by a large body of research, and extensive development and use of shared infrastructures based on federated or mediated systems [35], in which organizations come together to selectively share their data. In addition, governmental, public, and private institutions are increasingly required to make their data electronically available. This often involves large amounts of

¹This 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. A preliminary version of this paper appeared under the title "Minimal Data Upgrading to Prevent Inference and Association Attacks," in *Proc. of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, May 31-June 3, 1999 [5].

legacy or historical data, once considered classified or accessible only internally, that must be made partially available to outside interests.

This information sharing and dissemination process is clearly selective. Indeed, if on the one hand there is a need to disseminate some data, there is on the other hand an equally strong need to protect those data that, for various reasons, should not be disclosed. Consider, for example, the case of a private organization making available various data regarding its business (products, sales, etc.), but at the same time wanting to protect more sensitive information, such as the identity of its customers or plans for future products. As another example, government agencies, when releasing historical data, may require a sanitization process to “blank out” information considered sensitive, either directly or because of the sensitive information it would allow the recipient to infer. Effective information sharing and dissemination can take place only if the data holder has some assurance that, while releasing information, disclosure of sensitive information is not a risk. Given the possibly enormous amount of data to be considered, and the possible inter-relationships between data, it is important that the security specification and enforcement mechanisms provide automatic support for complex security requirements, such as those due to inference channels and data association [15].

Multilevel mandatory policies, providing a simple (in terms of specification and management) form of access control appear suitable for the problem under consideration, where, in general, classes of data need to be released to classes of users. Multilevel 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 levels that dominate theirs — mandatory policies provide a simple and effective way to enforce information protection [3, 19]. 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.

Unfortunately, capabilities of existing classification-based (multilevel) systems remain limited, and little, if any, support for the features mentioned above is provided. First, proposed multilevel database models [3] work under the assumption that data are classified upon insertion (by assigning them the security level of the inserting subject) and therefore provide no support for the classification of existing, possibly unclassified, databases, where a different classification lattice and different classification criteria may need to be applied. Second, despite the large body of literature on the topic and the proposal of several models for multilevel database systems [14, 16, 23, 26, 36], the lack of support for expressing and combating inference and data association channels that improperly leak protected information remains a major limitation [13, 15, 20]. 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 infer-

ence channels that leak sensitive information to lower levels. We provide a uniform formal framework to express constraints on the classification to be assigned to objects. Like others [6, 29, 30], we consider constraints that 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). In addition, we allow for constraints that express upper bounds on the classifications to be assigned to objects to take into consideration visibility requirements and subjects' existing (or prior) knowledge.

One of the major challenges in the determination of a data classification for a set of constraints is maximizing information visibility. Previous proposals in this direction are based on optimality cost measures, such as upgrading (i.e., bringing to a higher classification, assuming all data is at the lowest possible level, otherwise) the minimum number of attributes or executing the minimum number of upgrading steps [29, 30], or on explicit constraints allowing the specification of different preference criteria [6]. Determining such optimal classifications is often an NP-hard problem, and existing approaches typically perform exhaustive examination of all possible solutions [6, 30]. Moreover, these proposals are limited to the consideration of totally ordered sets of classifications [6, 29, 30] and intra-relation constraints (i.e., constraints involving attributes in a single relational table) due to functional and multivalued dependencies [30]. 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.

We introduce a notion of minimality that captures the property of a classification satisfying the protection requirements without overclassifying data. We propose an efficient approach for computing a minimal classification and present an algorithm implementing our approach that executes in low-order polynomial time. We also identify an important class of constraints, termed *acyclic* constraints, for which the algorithm executes in time linear in the size of the constraints. Finally, we show that the problem of computing classifications becomes intractable if the set of classification levels is not a lattice, but may be an arbitrary poset.

The technique we describe can form the basis of a practical tool for efficiently analyzing and enforcing classification constraints. This technique can be used for the classification of existing data repository to be classified (or sanitized) for external release in the design of multilevel database schemas, as well as in the enhancement of already classified data whose classification may need to be upgraded to account for inference attacks.

2. BASIC DEFINITIONS AND PROBLEM STATEMENT

Multilevel mandatory policies are based on the assignment of access classes to objects and subjects. Access classes in a set 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 (*no-read-up* principle [2]). The expression $a \succeq b$ is read as “ a dominates b ”, and $a \succ b$ (i.e., $a \succeq b$ and $a \neq b$) as “ a strictly dominates b ”. 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* 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

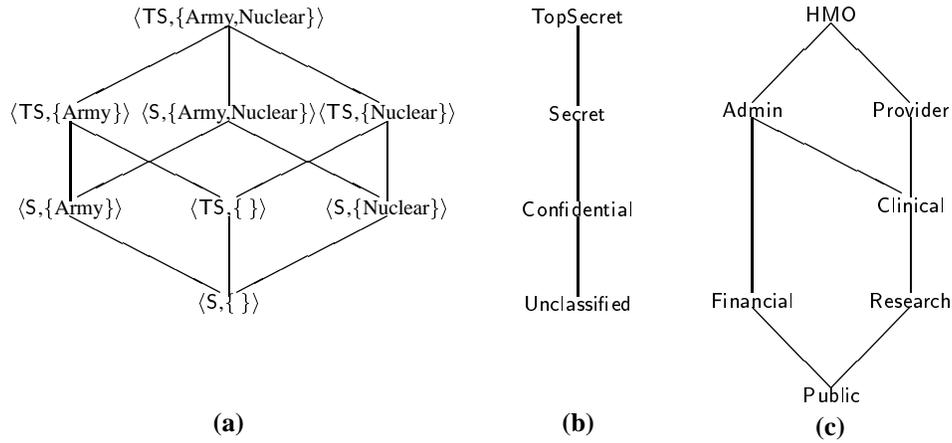


FIG. 1. Examples of security lattices.

a superset of that of the latter. For instance, Figure 1(a) illustrates a classification lattice with two levels ($TS \succ S$) and two categories (Army, Nuclear). 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. We refer to the maximum and minimum elements of a lattice as \top (top) and \perp (bottom), respectively, according to standard practice. Figure 1 depicts three classification lattices that are used to illustrate examples throughout the paper.

The security level to be assigned to an attribute may depend on several factors. The most basic consideration in determining the classification of an attribute is the sensitivity of the information it represents. For instance, if the names of a hospital's patients are not considered sensitive, the corresponding `patient` attribute might be labeled at a level such as `Public`. On the other hand, the illnesses of patients may be considered more sensitive, and the `illness` attribute might be labeled at a higher level, such as `Research`. Additional considerations that can affect the classification of an attribute include data *inference* and *association*. Data inference refers to the possibility of determining, exactly or inexactly, values of high-classified attributes from the values of one or more low-classified attributes. For instance, a patient's insurance and employer may not be considered sensitive, yet insurance and employer together may determine a specific insurance plan. Thus, knowledge of a patient's insurance and employer may permit inference of at least the type of insurance plan. If the insurance plan is considered more sensitive than either the insurance or the employer, it may be necessary to raise the classification of either the `insurance` or the `employer` attribute (or possibly both) to prevent access to information that would enable inference of insurance plan. Data association refers to the possibility that two or more attributes are considered more sensitive when their values are associated than when either appears separately. For instance, the fact that there is a patient named Alice may be `Public`, and the fact that there is a patient whose illness is HIV may be classified at `Research` level, but the fact that Alice's illness is HIV may be considered even more sensitive (e.g., `Clinical`). If the association of two or more attributes is considered more sensitive than any of the individual attributes, the classification of at least one of the attributes must be

set sufficiently high to prevent simultaneous access to all the attributes involved in the association.²

2.1. Classification Constraints

Classification constraints specify the requirements that the security levels assigned to attributes must satisfy. Specifically, they are constraints on a mapping $\lambda : \mathcal{A} \mapsto L$ that assigns to each attribute $A \in \mathcal{A}$ a security level $l \in L$, where (L, \succeq) is a classification lattice. We first identify four general classes of constraints according to the requirements they specify.

- *Basic constraints* specify minimum ground classifications for individual attributes, for example, $\lambda(\text{patient}) \succeq \text{Public}$ and $\lambda(\text{illness}) \succeq \text{Research}$. They reflect the sensitivity of the information represented by individual attributes and ensure that the attributes are assigned security levels high enough to protect the information.

- *Inference constraints* are used to prevent bypassing of basic constraints through data inference. They require the classification of an attribute, or the least upper bound of the classifications of a set of attributes, to dominate the classification of another attribute. For instance, the constraint $\text{lub}\{\lambda(\text{employer}), \lambda(\text{insurance})\} \succeq \lambda(\text{plan})$, corresponding to the inference example from the previous discussion, requires that the least upper bound (lub) of the levels assigned to attributes `employer` and `insurance` dominate the level assigned to attribute `plan`. Note that this constraint does indeed express the desired prevention of (low-to-high) inference from `employer` and `insurance` to `plan`, since, if the constraint is satisfied, a subject can access both `employer` and `insurance` only if the subject's clearance level dominates the classification of `plan`. It is also the weakest such constraint, allowing the greatest possible flexibility in the assignment of classifications to `employer` and `insurance`. In particular, it does not necessarily require the classification of either `employer` or `insurance` to dominate that of `plan`. For instance, referring to the lattice in Figure 1(c), if $\lambda(\text{plan}) = \text{Admin}$ the assignments $\lambda(\text{employer}) = \text{Research}$ and $\lambda(\text{insurance}) = \text{Financial}$ satisfy the constraint, since $\text{lub}\{\text{Research}, \text{Financial}\} = \text{Admin}$, although neither `Research` nor `Financial` dominates `Admin`.

- *Association constraints* are used to restrict the *combined* visibility of two or more attributes, requiring the least upper bound of their classifications to dominate a given ground level. For instance, the association constraint $\text{lub}\{\lambda(\text{patient}), \lambda(\text{bill})\} \succeq \text{Admin}$ requires that the least upper bound of the classifications of `patient` and `bill` dominate `Admin`. Note that the same effect could be achieved by a basic constraint requiring the classification of either `patient` or `bill` to dominate `Admin`, but this alternative is in general stronger than necessary. For instance, if `patient` is already classified at the `Provider` level because of a basic constraint, a classification of `Financial` would suffice for `bill`. Also, there is no need to raise the level of `Patient` above `Admin`. The explicit association constraint is the weakest constraint form that specifies the desired requirement.

- *Classification integrity constraints* are imposed by the security model itself and have the same form as inference and association constraints. They typically include *primary key* constraints and *referential integrity* constraints [36]. Primary key constraints require

²Note that, in principle, association constraints could be enforced without direct classification of the involved attributes (by run time monitoring and logging [18]). In this paper we do not consider this hypothesis (expensive and most often infeasible in practice) and require, like others, association constraints to be satisfied directly on the classification of the attributes in the association.

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 attributes representing a foreign key dominate the classification of the attributes for which it is foreign key. The purpose of classification integrity constraints is to ensure that the view of a database visible at any given level adheres to the data integrity constraints imposed by the data model. For instance, if primary key constraints are not satisfied, views at certain levels would contain null values for key attributes. Similarly, if referential integrity constraints are not satisfied, views at certain levels would have foreign keys with no corresponding primary key.

The constraints in all the four categories express restrictions on the visibility of information and can be expressed in a single general form termed *lower bound constraints*, defined formally as follows.

DEFINITION 2.1 (Lower Bound Constraint). *Let \mathcal{A} be a set of attributes and $\mathcal{L} = (L, \succeq)$ be a classification lattice. A lower bound constraint over \mathcal{A} and \mathcal{L} is an expression of the form $\text{lub}\{\lambda(A_1), \dots, \lambda(A_n)\} \succeq X$, where $n > 0$, $A_i \in \mathcal{A}$, $i = 1, \dots, n$, 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 $\lambda(A_1) \succeq X$.*

Although inference and association constraints differ in form (i.e., any inference constraint always has the level of an attribute on its right-hand side, while any association constraint always has a security level on its right-hand side), this distinction is not important in our classification approach, and Definition 2.1 does not distinguish between the different classes of lower bound constraints.

Note that all constraints allowed by Definition 2.1 have the form \succeq (as opposed to \preceq), with security levels on the right-hand side only. That is, each of them specifies a lower bound on the classification that can be assigned to the attributes (which can be upgraded as required by other constraints). For instance, a constraint requiring `illness` to be classified at level `Research` will be stated as $\lambda(\text{illness}) \succeq \text{Research}$, implying that `illness` must be classified *at least* `Research`. This interpretation is a property of the problem under consideration, where data classification may need to be upgraded to combat inference channels and to solve association constraints. Note that assigning to an attribute a classification lower than that required by lower bound constraints would, directly or indirectly, leak information to subjects not cleared for it. Thus, the main function of lower bound constraints is to capture the requirements on classification assignments that will prevent improper downward information flow.

Although lower bound constraints are sufficient for expressing the prevention of downward information flow, it can also be useful to establish *maximum* levels that should be assigned to attributes. Such maximum levels can be specified by *upper bound constraints*, defined as follows.

DEFINITION 2.2 (Upper Bound Constraint). *Let \mathcal{A} be a set of attributes and $\mathcal{L} = (L, \succeq)$ be a classification lattice. An upper bound constraint over \mathcal{A} and \mathcal{L} is an expression of the form $l \succeq \lambda(A)$, where $l \in L$ is a security level and $A \in \mathcal{A}$ is an attribute.*

Upper bound constraints have two main uses. One is the specification of visibility requirements, since their satisfaction ensures that the attribute will be visible to all subjects with level dominated by the specified upper bound. For instance, if we wish to guarantee

that names of patients in a hospital are always accessible to the administrative staff, we might impose the constraint $\text{Admin} \succeq \lambda(\text{patient})$ to prevent the classification of `patient` from being raised above `Admin`. In fact, this constraint together with the lower bound constraint $\lambda(\text{patient}) \succeq \text{Admin}$ effectively forces the classification of `patient` to be exactly `Admin`. The other main use of upper bound constraints is the modeling of subjects' existing or prior knowledge of certain information stored in the database. If such knowledge is not accounted for in the classification of the database, it is possible to produce classifications that provide a false sense of security. For instance, suppose that providers know the illnesses of patients in a hospital. This knowledge could be captured by the constraint $\text{Provider} \succeq \lambda(\text{illness})$. Without this constraint, it might happen that inference or association constraints produce a higher (or incomparable) classification, say `HMO`, for `illness`. Thus, although `illness` appears to be information classified at `HMO`, it really is not, since providers already know the illnesses of patients. Explicit upper bound constraints can prevent the assignment of such misleading classifications.

Lower and upper bound constraints can be represented abstractly as pairs (lhs, rhs) , where lhs is the security level or the (possibly singleton) 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. Among lower bound constraints we refer to constraints whose left-hand side is singleton as *simple* constraints, and to constraints with multiple elements in the left-hand side as *complex* constraints. Although the definitions do not permit lub expressions on the right-hand sides of constraints, this does not limit their expressiveness, since a constraint of the form $X \succeq \text{lub}\{\lambda(A_1), \dots, \lambda(A_n)\}$ is equivalent to the set of constraints $\{X \succeq \lambda(A_1), \dots, X \succeq \lambda(A_n)\}$. In the remainder of the paper we refer to arbitrary sets of lower and upper bounds constraints simply as *classification constraints* and distinguish between lower and upper bound constraints when necessary.

Any set of classification constraints can be viewed as a directed graph, which we call the constraint graph, not necessarily connected, containing a node for each attribute $A \in \mathcal{A}$ and security level $l \in L$. Each simple constraint is represented in the graph by a directed edge from the node representing the left-hand side to the node representing the right-hand side. For complex constraints, the left-hand side is represented by the set of nodes corresponding to the attributes on the left-hand side of the constraint. We refer to such a set of nodes as a *hypernode*. The complex constraint is itself represented in the graph by a directed edge from the hypernode representing the left-hand side to the node representing the right-hand side.

Figure 2 illustrates an example of a classification constraint graph, where security levels are taken from the lattice in Figure 1(c). Circle nodes represent attributes, square nodes represent security levels, and dashed ellipses represent hypernodes. Note that upper bound constraints are edges from level nodes to attribute nodes. All other edges represent lower bound constraints. The constraints refer to information in a hospital database and reflect the following scenario. The two association constraints, c_{23} and c_{24} , require protection of each patient's `illness` and `bill` information. In particular, the association between `patients` and their `illnesses` can be known only to subjects at level `Clinical` or above, and the association between `patients` and their `bills` can be known only to subjects at level `Admin` or above. Other lower bound constraints reflect (precise or imprecise) inferences that can be drawn from the data and that must therefore be reflected in the classification. For instance, by knowing the `doctor` who is caring for a `patient`, a subject can deduce the patient's `illness` within the specific set of illnesses falling in the doctor's specialty.

CLASSIFICATION CONSTRAINTS

Basic constraints

- $c_1 : \lambda(\text{exam}) \succeq \text{Public}$
- $c_2 : \lambda(\text{visit}) \succeq \text{Public}$
- $c_3 : \lambda(\text{treatment}) \succeq \text{Public}$
- $c_4 : \lambda(\text{doctor}) \succeq \text{Public}$
- $c_5 : \lambda(\text{patient}) \succeq \text{Public}$
- $c_6 : \lambda(\text{division}) \succeq \text{Public}$
- $c_7 : \lambda(\text{employer}) \succeq \text{Public}$
- $c_8 : \lambda(\text{plan}) \succeq \text{Financial}$
- $c_9 : \lambda(\text{bill}) \succeq \text{Financial}$
- $c_{10} : \lambda(\text{insurance}) \succeq \text{Financial}$
- $c_{11} : \lambda(\text{illness}) \succeq \text{Research}$
- $c_{12} : \lambda(\text{prescription}) \succeq \text{Clinical}$

Inference constraints

- $c_{13} : \lambda(\text{treatment}) \succeq \lambda(\text{visit})$
- $c_{14} : \lambda(\text{visit}) \succeq \lambda(\text{exam})$
- $c_{15} : \lambda(\text{exam}) \succeq \lambda(\text{treatment})$
- $c_{16} : \lambda(\text{treatment}) \succeq \lambda(\text{illness})$
- $c_{17} : \lambda(\text{prescription}) \succeq \lambda(\text{treatment})$
- $c_{18} : \lambda(\text{doctor}) \succeq \lambda(\text{illness})$
- $c_{19} : \lambda(\text{patient}) \succeq \lambda(\text{employer})$
- $c_{20} : \lambda(\text{illness}) \succeq \lambda(\text{division})$
- $c_{21} : \text{lub}\{\lambda(\text{division}), \lambda(\text{plan})\} \succeq \lambda(\text{doctor})$
- $c_{22} : \text{lub}\{\lambda(\text{employer}), \lambda(\text{insurance})\} \succeq \lambda(\text{plan})$

Association constraints

- $c_{23} : \text{lub}\{\lambda(\text{illness}), \lambda(\text{patient})\} \succeq \text{Clinical}$
- $c_{24} : \text{lub}\{\lambda(\text{bill}), \lambda(\text{patient})\} \succeq \text{Admin}$

Upper bound constraints

- $c_{25} : \text{Admin} \succeq \lambda(\text{patient})$
- $c_{26} : \text{Admin} \succeq \lambda(\text{exam})$
- $c_{27} : \text{Provider} \succeq \lambda(\text{illness})$

CLASSIFICATION CONSTRAINT GRAPH

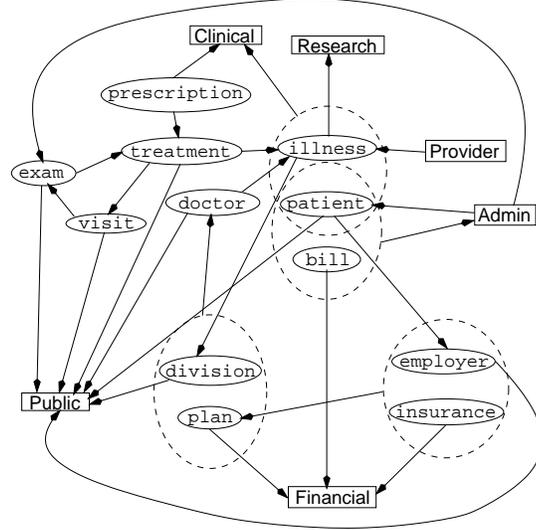


FIG. 2. An example of classification constraints and corresponding classification constraint graph

Hence, the classification of attribute `doctor` must dominate the classification of attribute `illness` (c_{18}). Analogously, the insurance plan together with the health care division allows inference on `doctor`. Hence, subjects should have visibility on both the `division` and `plan` only if they have visibility on `doctor`. In terms of the classifications, the least upper bound of the classification of `plan` and `division` must dominate the classification of `doctor` (c_{21}). The motivation behind the other inference constraints appearing in the figure is analogous. There are also several basic constraints that require the classification of certain attributes to dominate specific levels. For instance, `prescriptions` can be released only to subjects at level `Clinical` or above (c_{12}). In addition, there are three upper bound constraints reflecting the fact that specific information cannot be classified above certain levels because of required access or to account for information already known, as discussed. For instance, the classification of `patient` must be dominated by `Admin` (c_{25}) and the classification of `illness` must be dominated by `Provider` (c_{27}). It should be noted that the constraint graph is used only informally to help illustrate the approach. In the discussion 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 .

Among lower bound constraints we identify two subclasses of constraints. Intuitively, lower bound constraints whose graph representation is acyclic (i.e., is a DAG) are called

acyclic constraints, while those involved in a cycle, including cycles through hypernodes³, are called *cyclic* constraints. The notion of cyclic constraints is made more precise by the following definition.

DEFINITION 2.3 (Cyclic Constraints). *Let \mathcal{A} be a set of attributes, $\mathcal{L} = (L, \succeq)$ a classification lattice, and C a set of lower bound constraints over \mathcal{A} and \mathcal{L} . C is cyclic iff it can be ordered into a sequence $\langle (lhs_1, rhs_1), \dots, (lhs_n, rhs_n) \rangle$ such that $rhs_i \in lhs_{i+1}$, $1 \leq i \leq n-1$, and $rhs_n \in lhs_1$. Such a sequence is referred to as a constraint cycle. A set of constraints C is acyclic iff no subset of C is cyclic.*

A cycle involving only simple constraints is called a *simple cycle*. For instance, considering only the lower bound constraints in Figure 2, the constraints (illness, division), (division, plan), (doctor, illness), and (doctor, plan) are cyclic; the simple constraints (exam, treatment), (treatment, visit), and (visit, exam) constitute a simple cycle; and all other lower bound constraints are acyclic.

2.2. Minimal Classification

Given a set of classification constraints, the objective is to produce a classification $\lambda : \mathcal{A} \mapsto L$, which is an assignment of security levels in L to attributes in \mathcal{A} , that satisfies the constraints. 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 ordering. In general, there may exist many classifications that satisfy a set of constraints. However, not all classifications are equally good. For instance, the mapping $\lambda : \mathcal{A} \mapsto \{\top\}$ classifying all data at the highest possible level satisfies any set of lower bound 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 *overclassification* 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, classification 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 2.4 (Minimal Classification). *Given a set \mathcal{A} of attributes, classification 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 $\lambda' \models C$, $\lambda \succeq \lambda' \Rightarrow \lambda = \lambda'$.*

In other words, a minimal classification is one that both satisfies the constraints and is (pointwise) minimal in the lattice. Note that a minimal classification is not necessarily unique. For instance, referring to the lattice in Figure 1(c), the single constraint

³For the purpose of determining cycles only, 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.

$\text{lub}\{\lambda(A), \lambda(B)\} \succeq \text{Admin}$ has four minimal solutions, two of which classify one attribute Public and the other Admin, while the other two solutions classify one attribute Financial and the other Research.

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 classification 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 $\lambda(A) \succeq \perp$ for every attribute $A \in \mathcal{A}$. In addition, any set of lower bound constraints, which uses only the dominance relationship \succeq and security levels (constants) only on the right-hand side, is by itself consistent, since mapping every attribute to \top trivially satisfies all such constraints. Analogously, any set of upper bound constraints is by itself trivially consistent. However, a set of constraints that includes both upper and lower bound constraints is not necessarily consistent, the simplest example of inconsistent constraints being $\{\lambda(A) \succeq \top, \perp \succeq \lambda(A)\}$ (assuming that \top and \perp are distinct). Given an arbitrary set of constraints, our approach first enforces upper bound constraints to determine a firm maximum security level for each attribute. In the process, the consistency of the entire constraint set is checked. If the enforcement of upper bound constraints succeeds, a second phase evaluates the lower bound constraints to determine a minimal classification. We assume throughout that the left- and right-hand sides of each constraint are disjoint, since any constraint not satisfying this condition is trivially satisfied.

The remainder of this section describes the two solution phases at an intuitive level. An algorithm implementing the approach is presented formally in the following section.

3.1. Upper Bound Constraints

Upper bound constraints require the level of an attribute to be dominated by a specific security level. Because of the transitivity of the dominance relationship and the presence of lower bound constraints, upper bound constraints can indirectly affect other attributes besides those on which they are specified. For instance, the combination of upper bound constraint $\text{Admin} \succeq \lambda(\text{patient})$ and lower bound constraint $\lambda(\text{patient}) \succeq \lambda(\text{employer})$ forces Admin as a maximum level for attribute employer as well. Intuitively, an upper bound constraint affects all the attributes on those paths in the constraint graph that have the upper bound constraint as the initial edge. Each upper bound constraint can thus be enforced by traversing paths from security levels and propagating the constraint forward, lowering the levels of attributes encountered along the way accordingly (to the highest

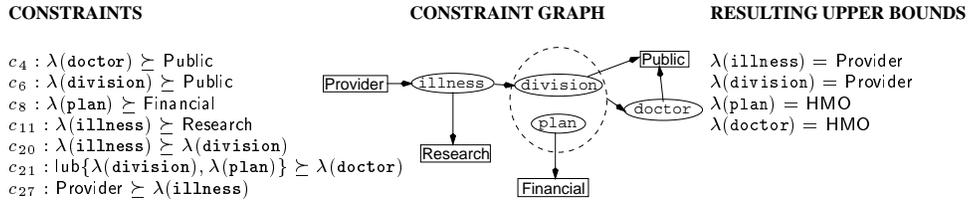


FIG. 3. Upper and lower bound constraints, graph, and resulting upper bounds

levels that satisfy the constraints). More precisely, let l be a security level on the left-hand side of some upper bound constraint. For each edge leaving from node l to some attribute node A , we propagate l forward as follows. If l dominates the current level of A , the upper bound constraint under consideration is satisfied, and the process terminates. If not, the level $\lambda(A)$ of attribute A is lowered to the greatest lower bound (\sqcap) of its current value and l . A unique such level l' is guaranteed to exist because we are working in a lattice. For each edge leaving from A , level l' is propagated to the node on which that edge is incident. Also, for each edge leaving from a hypernode that contains A , the least upper bound l'' of the levels of all the attributes in the hypernode is propagated to the node reached by that edge. Propagating a level l' to an attribute node A' means lowering $\lambda(A')$ to the greatest lower bound of its current level and l' , and proceeding recursively on all edges leaving from A' or from hypernodes containing it as just described. This process terminates for each path when a leaf node (security level) is reached. Then, if the level being propagated dominates the level of the leaf node, the process terminates successfully for the upper bound constraint being considered. Otherwise, an inconsistency in the constraints has been detected. In this case the process terminates with failure.⁴

The example in Figure 3 provides a simple illustration of the upper bound computation. Initially, the level of each attribute is set to HMO (\top). There is only one upper bound constraint, $\text{Provider} \succeq \lambda(\text{illness})$. Propagating level Provider forward causes $\lambda(\text{illness})$ to be lowered to $\text{HMO} \sqcap \text{Provider} = \text{Provider}$. Likewise, Provider is propagated to division as a result of the constraint $\lambda(\text{illness}) \succeq \lambda(\text{division})$, lowering $\lambda(\text{division})$ to Provider. Next, the constraint $\lambda(\text{division}) \succeq \text{Public}$ is checked and found to be satisfied, since $\text{Provider} \succeq \text{Public}$. Similarly, the constraint $\text{lub}\{\lambda(\text{division}), \lambda(\text{plan})\} \succeq \lambda(\text{doctor})$ is found to be satisfied, since the least upper bound of Provider and HMO is HMO, which dominates $\lambda(\text{doctor}) = \text{HMO}$. Finally, the remaining constraint on illness , $\lambda(\text{illness}) \succeq \text{Research}$ is checked and found to be satisfied, and the upper bound computation succeeds with the upper bounds as shown in the figure. Note that if we were to replace the upper bound constraint with $\text{Financial} \succeq \lambda(\text{illness})$, for example, the process would fail upon checking $\lambda(\text{illness}) \succeq \text{Research}$, since $\text{Financial} \not\succeq \text{Research}$ is false.

3.2. Lower Bound Constraints

Upon successful completion of the enforcement of upper bound constraints, the maximum allowed security level for each attribute is known, and the upper bound constraints require no further consideration. The second phase thus deals exclusively with lower

⁴In principle, the actions executed in this forward propagation process could be rescinded and the upper bound constraint ignored, pointing out the inability to satisfy it.

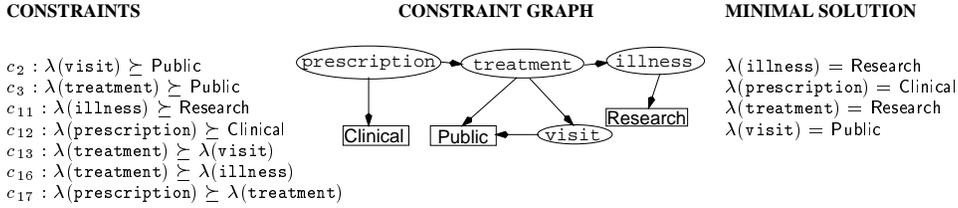


FIG. 4. Acyclic simple constraints, corresponding graph, and minimal solution

bound constraints to determine a minimal classification. Among lower bound constraints we consider separately the *acyclic* and *cyclic* constraints (see Definition 2.3). The reason for considering them separately is that acyclic constraints, which are expected to account for most constraints in practice, can be solved using a simpler and more efficient approach than that needed for cyclic constraints.

3.2.1. 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). Starting from the leaves, we traverse the graph backward (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 set of constraint edges $\{(A, X_1), \dots, (A, X_n)\}$ means assigning to A the least upper bound of all levels represented by X_1, \dots, X_n . Note that, because of the successful termination of the upper bound phase, this least upper bound is guaranteed to be dominated by the level assigned in the upper bound phase. As long as each X_i is guaranteed to remain fixed, propagating levels 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.

As an example, consider the simple constraints and the corresponding constraint graph in Figure 4. Applying the process just outlined, we first propagate level Public to `visit` and level Research to `illness`. With the final levels for `visit` and `illness` now known, we next propagate the least upper bound of Public, $\lambda(\text{visit})$, and $\lambda(\text{illness})$ to `treatment`, thus classifying it Research. Finally, we propagate the least upper bound of $\lambda(\text{treatment})$ and Clinical, to `prescription`, classifying it Clinical. The resulting minimal solution is shown in Figure 4.

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 modifying 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 a complex constraint can be solved minimally by choosing any single attribute on the left-hand side and assigning it a minimal level that satisfies the constraint, 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. Intuitively, this corresponds to enforcing the constraint on the attribute in the hypernode whose classification is computed last. As long as the

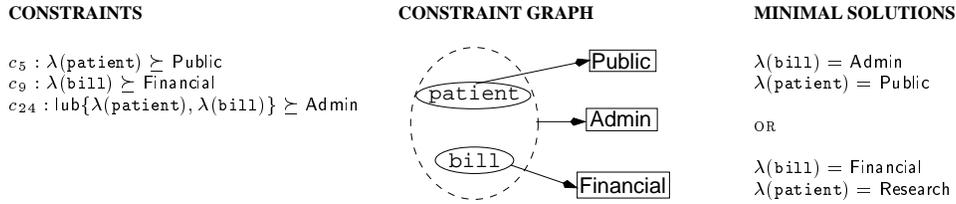


FIG. 5. Acyclic (simple and complex) constraints, graph, and minimal solutions

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 minimal assignment. For instance, consider the constraints in Figure 5. The complex constraint $\text{lub}\{\lambda(\text{bill}), \lambda(\text{patient})\} \succeq \text{Admin}$ (c_{24}) can be solved by assigning either Admin to `bill` or Research to `patient`. Note that either solution is minimal according to Definition 2.4. The particular minimal solution generated depends on the order of constraint evaluation. For the example in Figure 5, the first solution ($\{\lambda(\text{bill}) = \text{Admin}; \lambda(\text{patient}) = \text{Public}\}$) is computed if the simple constraint on `patient` (c_5) is evaluated first, whereas the second solution ($\{\lambda(\text{bill}) = \text{Financial}; \lambda(\text{patient}) = \text{Research}\}$) is computed if the simple constraint on `bill` (c_9) is evaluated first.

3.2.2. 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, that is, cycles involving only simple constraints, 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 instance, we might imagine replacing the simple cycle involving attributes `exam`, `treatment`, and `visit` in Figure 2 by a single node labeled “`exam, treatment, visit`” 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 on which to impose the constraint, 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 impose a level on 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 assignment to A may have been higher than necessary 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 on which to impose each complex constraint that guarantees a minimal classification using back-propagation. However, as the cycles become more complicated, the determination of such attributes becomes more complex. The problem becomes particularly acute for cyclic complex constraints whose left-hand sides are nondisjoint, since the choice of one attribute for one constraint may invalidate the choice made for another. For instance, consider the following two constraints from Figure 2: ($\{\text{patient}, \text{bill}\}, \text{Admin}$), ($\{\text{illness}, \text{patient}\}, \text{Clinical}$). Assigning level Admin to `patient` to satisfy the first constraint automatically satisfies the second constraint, im-

plying that the second constraint must also be imposed on `patient`. Moreover, it is not generally possible to choose a single attribute in the intersection of two or more left-hand sides on which to impose all the intersecting constraints. As an example, consider three constraints whose left-hand sides are $\{A, B\}$, $\{B, C\}$, and $\{A, C\}$, respectively. Two attributes appearing together in the left-hand side of one of these constraints will necessarily have a constraint imposed on them. 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 level assignment performed during back-propagation of levels through cycles involving complex constraints 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 assignments 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 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 the level of each such attribute incrementally 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 allowed by the upper bound constraints. 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 the 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 continue and try an even lower level for A . 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 definitively assigned (a ground level or an attribute whose level is already determined), and therefore cannot be lowered, we fail. 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 A 's final level. Repeating this procedure for each attribute, the result at the end of the entire process is a minimal classification for all attributes in the cycle.

For a simple illustration of this procedure, consider the constraint set in Figure 6, which contains a cyclic subset (c_{18} , c_{20} , c_{21}). Assume that all four attributes in the cycle (`division`, `doctor`, `illness`, and `plan`) are initially labeled at level HMO (\top). To enforce the cyclic constraints, we select an arbitrary attribute, say `illness`, from the cycle and attempt to lower its level. We may try either `Admin` or `Provider`, and we choose `Admin` arbitrarily. We can lower $\lambda(\text{illness})$ to `Admin` as long as all affected constraints remain satisfied. The first constraint on `illness` remains satisfied, since `Admin` \succeq `Research`. The second constraint on `illness` remains satisfied only if $\lambda(\text{division})$ can also be lowered to `Admin`. Attempting to lower $\lambda(\text{division})$ to `Admin`, we find the simple constraint on

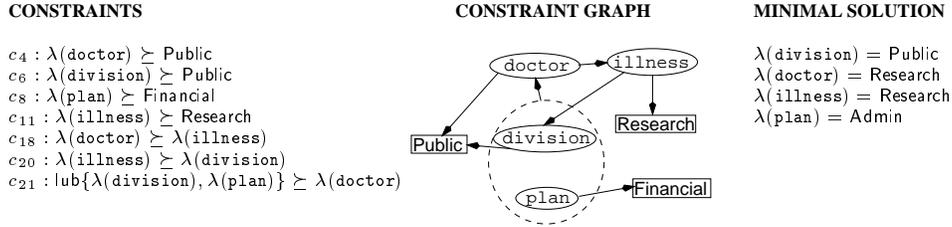


FIG. 6. Cyclic constraints, constraint graph, and a minimal solution

`division` (c_6) remains satisfied, and the complex constraint (c_{21}) as well, since $\lambda(\text{plan})$ is HMO. It is easy to see, then, that $\lambda(\text{illness})$ and, consequently, $\lambda(\text{division})$ can be lowered ultimately to `Research`. Suppose now that we attempt to lower $\lambda(\text{division})$. Since $\lambda(\text{division}) = \text{Research}$ we may try `Public` and find that both c_6 and c_{21} remain satisfied. Finally, we attempt to lower $\lambda(\text{doctor})$. We first try level `Admin` and find that the simple constraints on `doctor` (c_4 and c_{18}) remain satisfied, since `Admin` \succeq `Public` and `Admin` \succeq $\lambda(\text{illness}) = \text{Research}$. We now try to lower $\lambda(\text{doctor})$ to `Financial`. This attempt fails because constraint c_{18} remains satisfied only if $\lambda(\text{illness})$ can be lowered to `Public`. This is not possible because of the constraint $\lambda(\text{illness}) \succeq \text{Research}$ (c_{11}). We then try to lower $\lambda(\text{doctor})$ to `Clinical` and succeed. Subsequently, we try to lower $\lambda(\text{doctor})$ to `Research`. Again constraints c_4 and c_{18} remain satisfied, and so in the last step of the process we try to lower $\lambda(\text{doctor})$ to `Public`. As before, this attempt fails because it would require $\lambda(\text{illness})$ to be lowered to `Public` as well, which cannot be done. Once the cyclic constraints have been solved `plan` is assigned level `Admin`, which is the lowest level that it can assume without violating the dominance constraints imposed on it, namely $\text{lub}\{\lambda(\text{division}), \lambda(\text{plan})\} \succeq \lambda(\text{doctor})$ (c_{21}) and $\lambda(\text{plan}) \succeq \text{Financial}$ (c_8). The computed minimal solution appears in Figure 6.

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 four 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 both the upper and lower bound constraint solving phases. The second part enforces upper bound constraints and, in the process, checks the entire input constraint set for consistency. The third and fourth 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 presented formally in Figures 7 – 11. Here we describe them informally.

We assume that the input constraint set C is partitioned into two sets: C_{upper} , the upper bound constraints, and C_{lower} , the lower bound constraints. The upper bound constraints are considered only in the computation of upper bounds for the security levels of attributes (procedure **compute_upper_bounds**), while lower bound constraints are considered in all phases of the algorithm. To begin the algorithm, procedure **main** (Figure 7) initializes several variables that are used either during the DFS procedures or in the actual classification process, as follows. For each attribute A , $Constr[A]$ is the set of (lower bound) constraints whose left-hand side includes attribute A , $visit[A]$ is used in the graph traversal to denote if A has been visited, and $done[A]$ is set to `TRUE` when A becomes definitively labeled. For each security level $l \in L$, we set $done[l]$ to `TRUE`, since security levels are constants, and $visit[l]$ to 1, since security levels are leaves in any (lower bound) constraint graph and thus represent terminal points in any traversal of the graph. With each constraint $c \in C_{lower}$ we associate a count, $count[c]$, initialized to the number of attributes in the left-hand side of c , and used during the computation of a solution to keep track of the number of attributes in the left-hand side of c remaining to be considered (in various phases of the algorithm).

Next, **main** proceeds to its primary task, which is to determine an ordering of the attributes that both identifies cyclic relationships and captures the order in which attributes will be considered when evaluating the classification constraints on them. This ordering reflects possible dependencies between the security levels of the attributes, as specified by the lower bound constraints, and is a total order over sets of attributes. Two attributes whose levels may be mutually dependent are part of a constraint cycle and are considered equivalent in terms of the attribute ordering. Intuitively, the security level of one attribute depends on that of a second attribute if the second is reachable from the first in the constraint graph. For the purpose of determining reachability *only*, we interpret each edge from a hypernode to a node as a set of edges, one from each attribute in the hypernode to the node. For instance, in the constraint $\text{lub}\{\lambda(\text{division}), \lambda(\text{plan})\} \succeq \lambda(\text{doctor})$ in Figure 6, we would consider `doctor` to be reachable from either `division` or `plan`. This interpretation reflects the fact that the security levels of either `division` or `plan` may depend on the level of `doctor`. Using this interpretation of reachability, then, attributes involved in cyclic constraints correspond to those in strongly connected components (SCCs) of the constraint graph. More formally, we define SCCs as disjoint sets of attributes involved in constraint cycles (Definition 2.3), as follows.

DEFINITION 4.1 (Strongly Connected Component). *Let \mathcal{A} be a set of attributes, $\mathcal{L} = (L, \succeq)$ a classification lattice, and C a set of lower bound constraints over \mathcal{A} and \mathcal{L} . A strongly connected component (SCC) of C is a nonempty subset of \mathcal{A} having the following properties:*

1. Every attribute $A \in \mathcal{A}$ is a member of exactly one SCC.
2. Any two distinct attributes $A_1, A_2 \in \mathcal{A}$ belong to the same SCC if and only if there exist constraints $(lhs_1, A_1) \in C$ and $(lhs_2, A_2) \in C$ and a constraint cycle $C' \subseteq C$ such that $(lhs_1, A_1) \in C'$ and $(lhs_2, A_2) \in C'$.

In other words, the SCCs of a constraint set partition the attributes, and each SCC is the collection of attributes on the right-hand sides of a maximal set of cyclic constraints (which may include constraints in intersecting or overlapping constraint cycles).

Constraint cycles can therefore be identified by applying known methods for identification of SCCs. If we think of each such SCC as a kind of node (or node group) itself, the

ALGORITHM 4.1 (Minimal Classification Generation).

Input: A set of attributes \mathcal{A} , a classification lattice \mathcal{L} , and a set of constraints $C = C_{lower} \cup C_{upper}$ over \mathcal{A} and \mathcal{L} , where C_{lower} is a set of lower bound constraints and C_{upper} is a set of upper bound constraints.

Output: A minimal classification of \mathcal{A} .

MAIN

/ Variable initialization and setting */*

For $A \in \mathcal{A}$ **do**

$Constr[A] := \emptyset$; */* Keeps track of sets of constraints with A on the left-hand side */*

$visit[A] := 0$; */* Keeps track of whether A has been visited in service routines */*

$done[A] := \text{FALSE}$ */* Keeps track of whether the level of A is final or can still change */*

For $l \in L$ **do** $done[l] := \text{TRUE}$; $visit[l] := 1$ */* Security level constants are flagged as done and visited */*

For $c=(lhs,rhs) \in C_{lower}$ **do**

$count[c] := 0$; */* General-purpose counter of attributes in lhs of c that have been, or need to be, processed */*

For $A \in lhs$ **do** */* Add c to set of constraints to be imposed on A and increment counter of c by one */*

$Constr[A] := Constr[A] \cup \{c\}$; $count[c] := count[c] + 1$

/ Compute SCCs and topological order between them. Each SCC is an ordered list of attributes */*

$Stack := \emptyset$; */* Start DFS with empty stack. */*

For $A \in \mathcal{A}$ **do** **if** $visit[A] = 0$ **then** $dfs_visit(A)$ */* Execute DFS on constraint graph */*

$max_scc := 0$ */* Initialize number of SCCs */*

For $i = 1, \dots, |\mathcal{A}|$ **do** $scc[i] := \langle \rangle$ */* Initialize all SCCs to the empty list */*

For $A \in \mathcal{A}$ **do** $visit[A] := 0$ */* Reinitialize all attributes to unvisited for reverse DFS */*

While $NOTEMPTY(Stack)$ **do** */* Execute reverse DFS, considering attributes in order dictated by $Stack$ */*

$A := POP(Stack)$

if $visit[A] = 0$ **then** */* Create new SCC with A and all attributes reached from it in reverse DFS */*

$max_scc := max_scc + 1$; $scc[max_scc] := \langle A \rangle$; $dfs_back_visit(A)$

/ Enforce constraints to compute minimal solution */*

For $A \in \mathcal{A}$ **do** $\lambda(A) := \top$; $visit[A] := 0$ */* Initialize levels of all attributes and set them as unvisited */*

compute_upper_bounds */* Enforce upper bound constraints (Figure 8) */*

compute_partial_lubs */* Initialize cache of lub computations for complex constraints (Figure 9) */*

compute_minimal_solution */* Enforce lower bound constraints (Figure 10) */*

DFS_VISIT(A) */* Execute DFS from A pushing attributes in $Stack$ as their visit is finished */*

$visit[A] := 1$

For $(lhs, rhs) \in Constr[A]$ **do** **if** $visit[rhs] = 0$ **then** $dfs_visit(rhs)$

$PUSH(A, Stack)$

DFS_BACK_VISIT(A) */* Execute reverse DFS from A , appending attributes to A 's SCC */*

$visit[A] := 1$

For $(lhs, A) \in C_{lower}$ **do**

For $A' \in lhs$ **do**

if $visit[A'] = 0$ **then**

$scc[max_scc] := \text{concat}(\langle A' \rangle, scc[max_scc])$

$dfs_back_visit(A')$

FIG. 7. Main algorithm for computing a minimal classification.

attribute order we seek is essentially the topological order of the attribute nodes (in the case of acyclic constraints) and SCCs in the constraint graph. Once computed, this order is used to guide the evaluation of both upper and lower bound constraints.

The computation of the attribute ordering is accomplished through an adaptation of known approaches to SCC computation involving two passes of the graph with a depth first search (DFS) traversal [4, 32] of the lower bound constraints. The first pass (**dfs_visit**) executes a DFS on the constraints, recording each attribute in a stack (*Stack*) as its visit is concluded. The second pass (**dfs_back_visit**) considers attributes in the order in which they appear in *Stack*, assigning each to the SCC list $scc[max_scc]$ (where max_scc is incremented as each attributed is visited) and marking the attribute as visited. The SCCs are maintained as lists rather than sets so that the attributes within an SCC can be processed in a predictable order in other parts of the algorithm. For each new attribute A popped from *Stack*, the process walks the graph backward with a (reverse) DFS and adds to the SCC list containing A all attributes it finds still unvisited, since such attributes are necessarily part of the SCC containing A . Each SCC satisfies the following properties: (1) each attribute is a member of exactly one SCC, (2) any two attributes belong to the same SCC if and only if they appear together in a cycle (i.e. are mutually reachable), and (3) the index of the SCC to which any attribute belongs is no greater than that of any attribute reachable from it (i.e., on which it depends). As an example, consider the constraints in Figure 2. The execution of **main** produces the following SCCs:

```

scc[1] = ⟨prescription⟩
scc[2] = ⟨exam, treatment, visit⟩
scc[3] = ⟨insurance⟩
scc[4] = ⟨bill⟩
scc[5] = ⟨patient⟩
scc[6] = ⟨employer⟩
scc[7] = ⟨plan⟩
scc[8] = ⟨doctor, division, illness⟩

```

After initialization and SCC computation, **main** initializes each attribute's classification to \top and concludes by invoking the constraint solving procedures **compute_upper_bounds** (Figure 8), **compute_partial_lubs** (Figure 9), and **compute_minimal_solution** (Figure 10).

Procedure **compute_upper_bounds** constitutes the process for enforcing upper bound constraints outlined in Section 3.1. The first step directly evaluates each upper bound constraint by assigning to the constrained attribute the greatest lower bound (\sqcap) of its current level and the level specified by the constraint. The remainder of the procedure then propagates the enforced upper bounds throughout the (lower bound) constraint graph. This propagation considers each attribute in increasing SCC index order, since the upper bound of some attribute can affect the upper bounds only of attributes of equal or higher SCC index, and thus, the number of traversals is minimized. As each attribute is considered, its upper bound is propagated to other attributes, via procedure **upper_bound**. As **upper_bound** processes each constraint c on an attribute, it decrements $count[c]$. The level of the left-hand side is then propagated to the right-hand side only if the count has reached 0, or the attribute on the right-hand side is in the same SCC. Such delayed propagation optimizes the processing of acyclic constraints, since the SCC index of the right-hand side attribute of any acyclic constraint is higher than that of any attribute on the left-hand side. Only

after the last attribute in the left-hand side has been processed is it necessary to propagate the level forward.

By considering all attributes in increasing SCC index order, we ensure that all upper bounds are eventually propagated through the graph (or found to violate the consistency requirement). Within a cycle (SCC), each upper bound is propagated (procedure **upper_bound**) only as far as necessary — the process terminates along any path in which the upper bound is already satisfied. To ensure that all upper bounds are eventually propagated throughout the cycle, procedure **compute_upper_bounds** calls **upper_bound** on all unvisited attributes in the cycle. This process guarantees that, even if constraints propagate upper bounds (from attributes of lower SCC index) into a cycle at several points, every upper bound will be propagated as far as necessary. Note that the level assigned to any attribute can always be lowered as much as required by any upper bound propagated into the attribute. Propagation failure can occur only when security levels (leaf nodes) are reached and the incoming upper bound does not dominate the level of the leaf node. Such failure indicates that the upper bound constraint that originated the failed propagation is inconsistent with the lower bound constraints. If **compute_upper_bounds** completes successfully, we know that the constraints are consistent and that the computation of a minimal solution will be successful (Theorem 5.1). The upper bound constraints need no further consideration.

The purpose of **compute_partial_lubs** is to precompute, or cache, the least upper bounds (lubs) of the levels of certain subsets of attributes. These caches of “partial lubs” are used in procedure **minlevel** (called by **compute_minimal_solution**) to compute quickly the lub of the current levels of all attributes, except the attribute currently being processed, in the left-hand side of an arbitrary constraint. The computation of the partial lubs is designed to take advantage of the fact that attributes in the left-hand side of an acyclic constraint are processed in a predictable and consistent order (the SCC index order determined by the DFS procedures). For each lower bound constraint c of the form (lhs, rhs) , $|lhs| + 2$ partial lubs are computed. At the conclusion of **compute_partial_lubs** (the initialization phase), the first and last partial lub entries for each constraint are each set to \perp , and each remaining partial lub $Plub[c][i]$ for constraint c is the least upper bound of the levels of all attributes from SCC index 1 up to i . Later, in **compute_minimal_solution**, the attributes will be processed in *reverse* SCC index order. As the final assignment for each attribute is determined, its corresponding partial lub entry for each constraint will be recomputed so that its value is the least upper bound of its own level and the levels of all other attributes in the left-hand side already processed (whose partial lub entries correspond to *higher* index values). The net effect of the precomputation of partial lubs (by **compute_partial_lubs**) and their later recomputation in **compute_minimal_solution** is to maintain the following property: for any constraint $c = (lhs, rhs)$, when the i^{th} attribute of lhs is processed in **compute_minimal_solution**, the least upper bound of the current levels of *all other* attributes in lhs is equal to the least upper bound of $Plub[c][i - 1]$ and $Plub[c][i + 1]$.

Procedure **compute_minimal_solution** integrates the two approaches (back-propagation as outlined in Section 3.2.1 and forward lowering as outlined in Section 3.2.2) for determining a minimal solution for lower bound constraints. Unlike **compute_upper_bounds** it considers attributes in decreasing order of SCC index. That is, **compute_minimal_solution** traverses the constraint graph from the leaves back, rather than from the roots forward. For each attribute A at the SCC index being considered, all constraints in $Constr[A]$ are processed as follows. For each constraint c whose right-hand side is definitively labeled ($done[rhs]=TRUE$), the procedure determines how to enforce the constraint on A . If c is

COMPUTE_UPPER_BOUNDS

```

For  $(l, A) \in C_{upper}$  do  $\lambda(A) \sqcap l$  /* Enforce direct upper bound constraints */
For  $i := 1, \dots, max\_scc$  do /* Consider SCCs in topological order */
  For  $A \in scc[i]$  do
    If  $visit[A] = 0$  then upper_bound $(A, i)$  /* Propagate upper bounds forward */

UPPER_BOUND $(A, i)$ 
 $visit[A] := 1$ 
For  $c = (lhs, rhs) \in Constr[A]$  do
  /* Here count is used to delay forward propagation until all lhs attributes have been visited. */
  If  $count[c] > 0$  then  $count[c] := count[c] - 1$ 
  If  $count[c] = 0$  or  $rhs \in scc[i]$  then /* A is the last lhs attribute to be visited or c belongs to a cycle */
    /* Compute levlhs as the least upper bound of attributes on the left-hand side of c */
     $levlhs := \perp$ ; For  $A' \in lhs$  do  $levlhs := levlhs \sqcup \lambda(A')$ 
    If  $\neg(levlhs \succeq \lambda(rhs))$  then /* c is not satisfied */
      If  $rhs \in L$  then Fail /* Right hand side is a security level; constraints are inconsistent */
      else /* Right-hand side is an attribute; impose constraint on it and propagate forward */
         $\lambda(rhs) := \lambda(rhs) \sqcap levlhs$ 
        If  $rhs \in scc[i]$  then upper_bound $(rhs, i)$ 

```

FIG. 8. Procedures for enforcing upper bound constraints.**COMPUTE_PARTIAL_LUBS**

```

/* For each constraint c, compute an array  $Plub[c]$  with one entry for each attribute in the lhs, plus a */
/* sentinel ( $\perp$ ) at either end. The entries are ordered opposite to the order in which attributes are processed */
/* later in compute_minimal_solution. At the end of this computation, each entry  $Plub[c][j]$  */
/* contains the lub of the levels of attributes corresponding to entries 1 through j. */
For  $c \in C_{lower}$  do  $count[c] := 0$ ;  $Plub[c][0] := \perp$ ;
For  $i := 1, \dots, max\_scc$  do /* Consider SCCs in topological order. */
  For  $A \in reverse(scc[i])$  do /* Within each SCC, consider attributes in reversed order */.
    For  $c = (lhs, rhs) \in Constr[A]$  do
       $count[c] := count[c] + 1$ ;  $j := count[c]$ 
       $Plub[c][j] := Plub[c][j - 1] \sqcup \lambda(A)$ 
For  $c \in C_{lower}$  do  $j := count[c] + 1$ ;  $Plub[c][j] := \perp$ 

```

FIG. 9. Computation of upper bound of complex constraints

simple ($|lhs| = 1$), the level of the right-hand side is accumulated via the least upper bound (\sqcup) operation into variable l (initialized to \perp). Otherwise, c is complex, and **minlevel** is called to compute a minimal level that A must dominate (accounting for the current levels of the other attributes on the left-hand side of the constraint) and still satisfy the constraint. Procedure **minlevel** first computes the least upper bound of the levels of all other attributes (*lubothers*) by using the precomputed partial lub. If A is the j^{th} attribute on the left-hand side to be processed, the lub of the other levels is simply the lub of the partial lubs $Plub[c][j - 1]$ and $Plub[c][j + 1]$. Next, **minlevel** computes a minimal level for A that maintains satisfaction of c by descending the lattice along a path from A 's current level, one level at a time, stopping at the lowest level found whose direct descendants would

COMPUTE_MINIMAL_SOLUTION

```

For  $i := \text{max\_scc}, \dots, 1$  do /* Consider SCCs in reverse topological order */
  For  $A \in \text{scc}[i]$  do /* Consider attributes in list order */
     $\text{done}[A] := \text{TRUE}; l := \perp$  /* Set  $A$  as tentatively done and initialize level  $l$  to be assigned to  $A$  to  $\perp$  */
    For  $c = (\text{lhs}, \text{rhs}) \in \text{Constr}[A]$  do /* Consider all constraints where  $A$  appears on the left-hand side */
      If  $\text{done}[\text{rhs}]$  then /*  $\text{rhs}$  is definitively labeled */
        case  $|\text{lhs}|$  of /* Depending on whether  $c$  is a simple ( $|\text{lhs}| = 1$ ) or complex ( $|\text{lhs}| > 1$ ) constraint */
          1:  $l := l \sqcup \lambda(\text{rhs})$  /* Raise  $l$  according to the level being back-propagated */
          >1:  $l := l \sqcup \text{minlevel}(A, c)$  /*  $\text{minlevel}(A, c)$  is a minimal level for  $A$  that does not violate  $c$  */
        else  $\text{done}[A] := \text{FALSE}$  /*  $A$  is involved in a cycle; computed level  $l$  is not final */
      If  $\text{done}[A]$  then  $\lambda(A) := l$  /*  $l$  is final; assign it to  $A$  */
    else /*  $A$  is in cycle; execute forward lowering propagation */
       $DSet := \{l' \mid l' \text{ is a maximal level, } \lambda(A) \succ l' \succeq l\}$  /* The set of levels directly dominated by  $l$  */
      While  $DSet \neq \emptyset$ 
        Choose  $l''$  in  $DSet$ ;  $DSet := DSet - l''$ 
         $Lower := \text{try\_to\_lower}(A, l'')$  /* Downgrades required if  $A$  is lowered to  $l''$  (Figure 11) */
        If  $Lower \neq \emptyset$  then /*  $A$  can be downgraded to  $l''$  */
          For  $(A', l') \in Lower$  do  $\lambda(A') := l'$  /* Enforce all required downgrades */
           $DSet := \{l' \mid l' \text{ maximal level, } \lambda(A) \succ l' \succeq l\}$  /* Try to lower again from the current level */
         $\text{done}[A] := \text{TRUE}$  /* Level of  $A$  is final */
      /* For each constraint  $c$  in which  $A$  appears on the lhs, update the entry  $j$  in  $Plub[c]$  corresponding to  $A$  */
      /* to be the lub of the levels of all attributes corresponding to entries  $j$  through  $|\text{lhs}|$ . Note that, because */
      /* of the order in which attributes are processed, entry  $j + 1$  has already been set to the lub of the final */
      /* levels of all attributes corresponding to entries  $j + 1$  and higher. */
      For  $c \in \text{Constr}[A]$  do
         $j := \text{count}[c]$ ;
         $Plub[c][j] := \lambda(A) \sqcup Plub[c][j + 1]$ 
         $\text{count}[c] := \text{count}[c] - 1$ 

MINLEVEL}(A,c)
/* Return a minimal level  $last$  for  $A$  that keeps  $c = (\text{lhs}, \text{rhs})$  satisfied */
 $j := \text{count}[c]$ ; /* Number of attributes in lhs of  $c$  whose level is not yet final */
 $(\text{lhs}, \text{rhs}) := c$ ; /* Get the lhs and rhs of  $c$  */
 $last := \lambda(A)$  /* Initialize  $last$  to  $A$ 's current level */
 $\text{lubothers} := Plub[c][j - 1] \sqcup Plub[c][j + 1]$ ; /* The lub of the levels of all attributes in  $\text{lhs}$  except  $A$  */
If  $\text{lubothers} \succeq \lambda(\text{rhs})$  then  $last := \perp$ ; /* If the lub of the other attributes satisfies  $c$ ,  $last$  can go to  $\perp$  */
else /* Lub of other attributes does not satisfy the constraint */
  /* Try successively lower levels for  $A$ , starting from  $last$  */
   $Try := \{l \mid l \text{ is a maximal level s. t. } last \succ l\}$  /* The set of levels directly dominated by  $last$  */
  While  $Try \neq \emptyset$  do
    Choose  $l$  in  $Try$ ;  $Try := Try - l$ 
    if  $(l \sqcup \text{lubothers}) \succeq \lambda(\text{rhs})$  then
       $last := l$ ;
       $Try := \{l \mid l \text{ is a maximal level s. t. } last \succ l\}$ 
return  $last$ 

```

FIG. 10. Lower bound constraint enforcement

all violate the constraint if assigned to A .⁵ The returned level is then accumulated via a lub operation into l . If all the constraints in $Constr[A]$ have the right-hand side done (which is always the case for acyclic constraints), A is simply assigned the level l so computed. Intuitively, this corresponds to enforcing back-propagation of security levels.

If, on the other hand, there is at least one constraint on A whose right-hand side is not definitively labeled ($done[rhs]=FALSE$), then attribute A must be involved in a constraint cycle. In this case, **compute_minimal_solution** proceeds by performing the forward lowering computation starting from A . At the start of this computation, level l represents a lower bound on A 's final level. Thus, A must eventually be assigned a level somewhere between its current level (which must be at least as high as l if the constraints are consistent) and l . We know that the constraints are satisfied with A at its current level, so the incremental forward lowering process begins by computing the set of levels ($DSet$) immediately below $\lambda(A)$ in the lattice. A member l'' of this set is chosen arbitrarily, and **try_to_lower** checks whether A can be lowered to level l'' . Procedure **try_to_lower** takes an attribute A and a level l and returns a set of attribute/level pairs that represent a satisfactory (but possibly non-minimal) assignment of levels to attributes that allows A to be lowered to l while maintaining satisfaction of all constraints. If no such assignment exists, **try_to_lower** returns the empty set to indicate failure. In the event that **try_to_lower** succeeds, **compute_minimal_solution** proceeds to enforce all level assignments (in set $Lower$) returned by **try_to_lower**. It then continues to attempt lowering the level of A from the most recent point of success. In the event that **try_to_lower** fails, another level to try is chosen from $DSet$. If all levels in $DSet$ are tried and fail ($DSet = \emptyset$), the current level assigned to A is a minimal level for A that maintains satisfaction of all constraints. Note that the condition $DSet = \emptyset$ must eventually become true, either because all attempts at lowering fail, or because \perp is reached. Note also that when a lowering attempt succeeds for some level in $l'' \in DSet$, it is not necessary to consider any other level in $DSet$. That is, a minimal level for A will always be found by considering only levels lower than the level that last succeeded. This point is discussed in more detail in the correctness proof for the algorithm.

The keys to the operation of procedure **try_to_lower** are the sets $Tocheck$ and $Tolower$. $Tocheck$ is the set of attribute/level pairs that remain to be checked to determine success or failure of the lowering attempt. $Tolower$ is the set of attribute/level assignment pairs that must ultimately be enforced if the lowering attempt succeeds. Now, for a given call **try_to_lower**(A, l), $Tocheck$ is initialized to (A, l) and $Tolower$ to \emptyset , since it is the attempt to lower the level of A to l that must be checked, while no assignments are yet implied by the attempted lowering. The procedure continues as long as there are assignments to check. The checking process amounts to propagating levels forward through the constraint graph, maintaining additional lowerings found to be necessary in set $Tocheck$, moving them then to set $Tolower$ for their later enforcement, if they do not result in any constraint violation. In the event of a constraint violation, **try_to_lower** fails immediately, returning the empty set. Otherwise, it returns the set $Tolower$ containing the assignments found to be necessary to enable the level of attribute A to be lowered to l .

⁵In the generally assumed case of classification lattices whose elements are pairs consisting of a classification level (taken from a totally ordered set) and a set of categories [25] (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. In this case, the entire **else** branch of the **minlevel** procedure can in fact be substituted with the simple computation, **If** ($lubothers_l < rhs_l$) **then** $last := \langle rhs_l, rhs_c \setminus lubothers_c \rangle$ **else** $last := \langle \perp, rhs_c \setminus lubothers_c \rangle$, where rhs_l ($lubothers_l$ resp.) is the classification level of rhs ($lubothers$ resp.) and rhs_c ($lubothers_c$ resp.) the corresponding set of categories.

TRY_TO_LOWER(A, l)

/ Determines whether A can be downgraded to l without violating any constraints. If so, returns the set */***

/ of required downgrades on all affected attributes; otherwise, returns \emptyset . **/**

$Tocheck := \{(A, l)\}$; */* Assignments of levels to attributes that must be checked for satisfaction **/**

$Tolower := \emptyset$; */* Assignments that have been checked and must be enforced if A is downgraded to l **/**

Repeat

/ Consider each assignment (A', l') in $Tocheck$ and determine whether it can be enforced **/**

Choose $(A', l') \in Tocheck$

$Tocheck := Tocheck - \{(A', l')\}$

$Tolower := Tolower \cup \{(A', l')\}$ */* Add (A', l') to the set of assignments to be enforced (tentative) **/**

For $c = (lhs, rhs) \in Constr[A']$ **do** */* Consider each constraint c with A' on the left hand side **/**

/ Compute level as lub of attributes in lhs **/**

$level := \perp$

For $A'' \in lhs$ **do**

If $\exists(A'', l'') \in Tolower$ **then** */* If A'' is in the set of attributes to be lowered **/**

$level := level \sqcup l''$ */* Use the level at which A'' is to be lowered **/**

else $level := level \sqcup \lambda(A'')$ */* Use its current level **/**

If $\neg(level \succeq \lambda(rhs))$ **then** */* c is not satisfied **/**

case $done[rhs]$ **of**

TRUE: **return** \emptyset ; */* Level of rhs cannot be changed, so c cannot be satisfied. Fail. **/**

FALSE: */* Try to lower level of rhs to keep c satisfied **/**

$newlevel := \lambda(rhs) \sqcap level$; */* Maximum level for rhs that keeps c satisfied **/**

If $\exists(rhs, l'') \in (Tolower \cup Tocheck)$ **then**

/ rhs already needed to be downgraded to some level l'' . If $newlevel$ is higher than l'' , **/**

/ nothing needs to be done. Otherwise, the level of rhs needs to be lowered to the glb **/**

/ of $newlevel$ and l'' , and this assignment needs to be checked. **/**

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 */* rhs was neither in $Tocheck$ nor in $Tolower$ **/**

$Tocheck := Tocheck \cup \{(rhs, newlevel)\}$

until $Tocheck = \emptyset$ */* Until there are no more required assignments to check **/**

/ Return set of downgrades to be enforced if A is downgraded to l . **/**

return $Tolower$

FIG. 11. Procedure `try_to_lower`

CLASSIFICATION CONSTRAINTS

Basic constraints

- $c_1 : \lambda(\text{exam}) \succeq \text{Public}$
- $c_2 : \lambda(\text{visit}) \succeq \text{Public}$
- $c_3 : \lambda(\text{treatment}) \succeq \text{Public}$
- $c_4 : \lambda(\text{doctor}) \succeq \text{Public}$
- $c_5 : \lambda(\text{patient}) \succeq \text{Public}$
- $c_6 : \lambda(\text{division}) \succeq \text{Public}$
- $c_7 : \lambda(\text{employer}) \succeq \text{Public}$
- $c_8 : \lambda(\text{plan}) \succeq \text{Financial}$
- $c_9 : \lambda(\text{bill}) \succeq \text{Financial}$
- $c_{10} : \lambda(\text{insurance}) \succeq \text{Financial}$
- $c_{11} : \lambda(\text{illness}) \succeq \text{Research}$
- $c_{12} : \lambda(\text{prescription}) \succeq \text{Clinical}$

Inference constraints

- $c_{13} : \lambda(\text{treatment}) \succeq \lambda(\text{visit})$
- $c_{14} : \lambda(\text{visit}) \succeq \lambda(\text{exam})$
- $c_{15} : \lambda(\text{exam}) \succeq \lambda(\text{treatment})$
- $c_{16} : \lambda(\text{treatment}) \succeq \lambda(\text{illness})$
- $c_{17} : \lambda(\text{prescription}) \succeq \lambda(\text{treatment})$
- $c_{18} : \lambda(\text{doctor}) \succeq \lambda(\text{illness})$
- $c_{19} : \lambda(\text{patient}) \succeq \lambda(\text{employer})$
- $c_{20} : \lambda(\text{illness}) \succeq \lambda(\text{division})$
- $c_{21} : \text{lub}\{\lambda(\text{division}), \lambda(\text{plan})\} \succeq \lambda(\text{doctor})$
- $c_{22} : \text{lub}\{\lambda(\text{employer}), \lambda(\text{insurance})\} \succeq \lambda(\text{plan})$

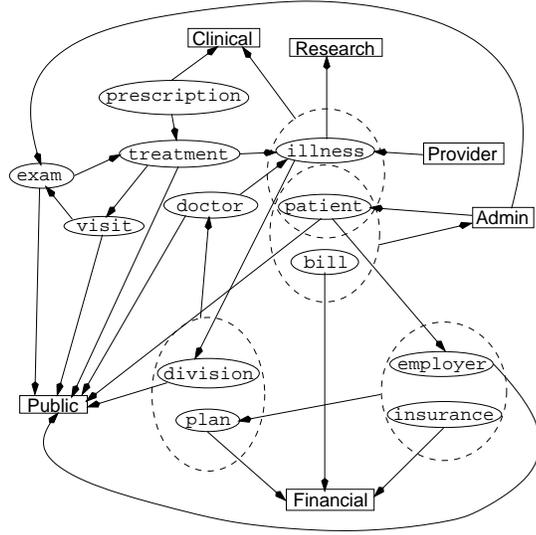
Association constraints

- $c_{23} : \text{lub}\{\lambda(\text{illness}), \lambda(\text{patient})\} \succeq \text{Clinical}$
- $c_{24} : \text{lub}\{\lambda(\text{bill}), \lambda(\text{patient})\} \succeq \text{Admin}$

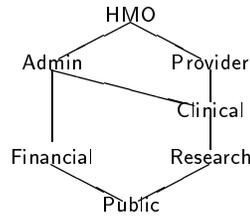
Upper bound constraints

- $c_{25} : \text{Admin} \succeq \lambda(\text{patient})$
- $c_{26} : \text{Admin} \succeq \lambda(\text{exam})$
- $c_{27} : \text{Provider} \succeq \lambda(\text{illness})$

CLASSIFICATION CONSTRAINT GRAPH



CLASSIFICATION LATTICE



EFFECTS OF THE CLASSIFICATION PROCESS

Attribute	Constr[]	SCC	Initial level	Result of compute_upper_bound	Result of compute_minimal_solution
prescription	c_{12}, c_{17}	1	HMO	HMO	Clinical
exam	c_1, c_{15}	2	HMO	Admin	Research
treatment	c_3, c_{13}, c_{16}	2	HMO	Admin	Research
visit	c_2, c_{14}	2	HMO	Admin	Research
insurance	c_{10}, c_{22}	3	HMO	HMO	Admin
bill	c_9, c_{24}	4	HMO	HMO	Financial
patient	$c_5, c_{19}, c_{23}, c_{24}$	5	HMO	Admin	Clinical
employer	c_7, c_{22}	6	HMO	Admin	Public
plan	c_8, c_{21}	7	HMO	HMO	Admin
doctor	c_4, c_{18}	8	HMO	HMO	Research
division	c_6, c_{21}	8	HMO	Clinical	Public
illness	c_{11}, c_{20}, c_{23}	8	HMO	Clinical	Research

FIG. 12. Classification constraints, constraint graph, and classification lattice of our working example and values produced by the classification process

Note that in the forward-lowering process, the level propagated 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 record this required assignment to be checked (in *Tocheck*) and propagate the level forward.

Once a minimal level has been computed for any attribute A , **compute_minimal_solution** updates the partial lubs in which $\lambda(A)$ is involved to keep the partial lubs correct with respect to λ , as described earlier.

EXAMPLE 4.1. Figure 12 displays the set of constraints of Figure 2 and their evaluation by the classification process. For the reader's convenience, the corresponding classification graph and the classification lattice are also shown. The table at the bottom of the figure reports for each attribute A , the set of lower bound constraints imposed on it ($Constr[A]$), the SCC to which it belongs as computed by **main**, and its security level at initialization, after the enforcement of the upper bound constraints (procedure **compute_upper_bounds**), and after the enforcement of the lower bound constraints (procedure **compute_minimal_solution**). The execution of the two procedures is illustrated in Figure 13. Table **compute_upper_bounds** illustrates the effects of the procedure for the different values of variable i (ranging over the SCC index in increasing order) and variable A (ranging over all the attributes within each SCC). For each such iteration, the table reports the calls to procedure **upper_bound** (where recursive calls are indented) together with the level assignment updates caused by them and the constraint whose evaluation caused the update (column c). The top portion of the table (with no entries for i) describes the initial direct enforcement of upper bound constraints on the constrained attribute.

The second table illustrates the execution of procedure **compute_minimal_solution** for the different values of variable i (ranging over the SCC index in decreasing order) and variable A (ranging over all the attributes within each SCC). For changes due to back-propagation (A is involved only in acyclic constraints), the table reports the change to A 's level and the constraints that caused that change, which are all the constraints with A on the left-hand side (i.e., $Constr[A]$). For changes due to forward propagation (A is involved in a cycle), the table reports the calls to procedure **try_to_lower** together with the level updates caused by them and the constraints (column c) that forced the updates. For unsuccessful calls to **try_to_lower**, value \emptyset is reported in the last column of the table, while column c reports the constraint that caused the failure (because it could not be satisfied).

5. CORRECTNESS AND COMPLEXITY ANALYSIS

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

THEOREM 5.1 (Correctness). *Algorithm 4.1 solves Problem 2.1 (MIN-LATTICE-ASSIGNMENT). That is, given a set C of classification constraints over a set A of attributes*

COMPUTE_UPPER_BOUNDS

i	A	calls to UPPER_BOUND	c	updates to levels
			c_{25} c_{26} c_{27}	$\lambda(\text{patient}) = \text{Admin}$ $\lambda(\text{exam}) = \text{Admin}$ $\lambda(\text{illness}) = \text{Provider}$
1	prescription	upper_bound (prescription,1)		
2	exam	upper_bound (exam,2)	c_{15}	$\lambda(\text{treatment}) = \text{Admin}$
		upper_bound (treatment,2)	c_{13}	$\lambda(\text{visit}) = \text{Admin}$
		upper_bound (visit,2)	c_{16}	$\lambda(\text{illness}) = \text{Clinical}$
	treatment			
	visit			
3	insurance	upper_bound (insurance,3)		
4	bill	upper_bound (bill,4)		
5	patient	upper_bound (patient,5)	c_{19}	$\lambda(\text{employer}) = \text{Admin}$
6	employer	upper_bound (employer,6)		
7	plan	upper_bound (plan,7)		
8	doctor	upper_bound (doctor,8)		
	division	upper_bound (division,8)		
	illness	upper_bound (illness,8) upper_bound (division,8)	c_{20}	$\lambda(\text{division}) = \text{Clinical}$

COMPUTE_MINIMAL_SOLUTION

i	A	calls to TRY_TO_LOWER	c	updates to levels
8	doctor	try_to_lower (doctor,Admin)		$\lambda(\text{doctor}) = \text{Admin}$
		try_to_lower (doctor,Financial)	c_{11}	\emptyset
		try_to_lower (doctor,Clinical)		$\lambda(\text{doctor}) = \text{Clinical}$
		try_to_lower (doctor,Research)	c_{18} c_{20}	$\lambda(\text{doctor}) = \text{Research}$ $\lambda(\text{illness}) = \text{Research}$ $\lambda(\text{division}) = \text{Research}$
	try_to_lower (doctor,Public)	c_{11}	\emptyset	
	division		c_6, c_{21}	$\lambda(\text{division}) = \text{Public}$
	illness		c_{11}, c_{20}, c_{23}	$\lambda(\text{illness}) = \text{Research}$
7	plan		c_8, c_{21}	$\lambda(\text{plan}) = \text{Admin}$
6	employer		c_7, c_{22}	$\lambda(\text{employer}) = \text{Public}$
5	patient		$c_5, c_{19}, c_{23}, c_{24}$	$\lambda(\text{patient}) = \text{Clinical}$
4	bill		c_9, c_{24}	$\lambda(\text{bill}) = \text{Financial}$
3	insurance		c_{10}, c_{22}	$\lambda(\text{insurance}) = \text{Admin}$
2	exam	try_to_lower (exam,Financial)	c_{16}	\emptyset
		try_to_lower (exam,Clinical)	c_{15} c_{13}	$\lambda(\text{exam}) = \text{Clinical}$ $\lambda(\text{treatment}) = \text{Clinical}$ $\lambda(\text{visit}) = \text{Clinical}$
		try_to_lower (exam,Research)	c_{15} c_{13}	$\lambda(\text{exam}) = \text{Research}$ $\lambda(\text{treatment}) = \text{Research}$ $\lambda(\text{visit}) = \text{Research}$
		try_to_lower (exam,Public)	c_{16}	\emptyset
	treatment			
	visit		c_2, c_{14}	$\lambda(\text{visit}) = \text{Research}$
1	prescription		c_{12}, c_{17}	$\lambda(\text{prescription}) = \text{Clinical}$

FIG. 13. Execution of the classification process on the constraints of Figure 12

and a classification lattice $\mathcal{L} = (L, \succeq)$, Algorithm 4.1 generates a minimal classification mapping $\lambda : \mathcal{A} \mapsto L$ satisfying 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 total 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 least upper bound or greatest lower bound of any two elements in \mathcal{L} . Define M to be maximum, for all paths from the top to the bottom of a lattice, of the sum of the branching factor of each element of the path. M is no greater than BH , and is also no greater than the size of \mathcal{L} (number of elements + size of the immediate successor relation).

THEOREM 5.2 (Complexity). *Algorithm 4.1 solves any instance $(\mathcal{A}, \mathcal{L}, C)$ of the problem MIN-LATTICE-ASSIGNMENT in $O(N_{\mathcal{A}}SHMc)$ time, and, if the set of constraints C is acyclic, in $O(SMc)$ time. Therefore, MIN-LATTICE-ASSIGNMENT is solvable in polynomial time.*

Note, in particular, that the time taken by Algorithm 4.1 is linear in the size of the constraints for acyclic constraints, and no worse than quadratic for cyclic 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 bound above for the cyclic case is truly worst case — it assumes that the entire constraint set forms a single SCC, which rarely occurs in practice. For any instance of the problem, the acyclic complexity analysis applies to all acyclic portions of the constraint set. In Algorithm 4.1 the higher price is paid only for cyclic constraints, which 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. Previous work [31] 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 [1, 9] 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 classification 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 classification 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 [8] allows the encoding of any security level in a small number of machine words, effectively yielding constant-time lattice operations.

6. RETURNING A PREFERRED MINIMAL SOLUTION

As noted in Section 3.2.1, minimal solutions are generally not unique, since complex lower bound constraints can be solved minimally by assigning, if necessary, any one attribute on the left-hand side a sufficiently high level. The approach presented returns one minimal solution, where the particular solution returned depends both on the (fixed) topological order of attribute nodes and cycles that guides the back-propagation of security levels and on the (arbitrary) order in which constraints are evaluated within cycles. Not all minimal solutions to a set of constraints may be considered equal. Some solutions may be preferred over others, for instance because they grant greater visibility (i.e., accessibility to more subjects) on certain selected attributes.

Previous approaches addressing the problem of minimizing information loss while satisfying some upgrading constraints based the choice of the specific solution to be returned on the concept of “optimal” classification. Optimality is expressed as minimization of cost measures determined from the association of weights to attributes and costs to security levels, and where the cost of each solution is the weighted sum of the classifications assigned. Finding such an optimal solution is however an NP-hard problem, and existing approaches typically perform exhaustive examination of all possible solutions [30]. Beside suffering from a general computational intractability, these cost-based approaches are very difficult to use in practice, as it is generally far from obvious how to manipulate costs to achieve the desired classification behavior.

We describe here two ways of specifying preference criteria on the minimal solution to be returned which are intuitive and easy to use. We also illustrate how they can be included in our approach without increasing the computational cost of finding the solution.

Soft upper bound constraints. Soft upper bound constraints are, as their name suggests, upper bound constraints (Definition 2.2), whose satisfaction is not mandatory, rather they are desiderata on the solution. Intuitively, soft upper bound constraints express visibility requirements that should be satisfied in the solution, if possible. Since not all soft constraints may be simultaneously satisfiable, it is convenient to consider soft constraints ordered according to their importance. We assume a list of soft upper bound constraints is provided as input, where the order in which the constraints appear reflects their importance. Soft upper bound constraints are enforced just after the upper bound constraints provided as part of the problem specification (Section 2). The process for enforcing soft upper bound constraints is essentially the same as that for enforcing other upper bound constraints. The only difference is in the fact that constraints are considered in a specific order, and that constraints that cannot be satisfied (since they conflict with other upper or lower bound constraints or with soft upper bound constraints already enforced) can simply be ignored.

Attribute priority. Another, complementary, approach to specify and compute a preferred solution is the consideration of explicit priorities between attributes, which establish their importance in terms of visibility. The algorithm should then return the minimal solution that avoids penalizing those attributes whose visibility is more important.

To the purpose of considering priorities, we first assume that attributes are prioritized according to a total order \leq_O , where $A \leq_O B$ implies that the visibility of A is more important than the visibility of B . We then extend this order to classifications as follows.

DEFINITION 6.1 (Lexicographic Order). *Given a set A of attributes, a classification lattice $\mathcal{L} = (L, \succeq)$, and a total ordering \leq_O on the elements of A , a classification*

$\lambda : \mathcal{A} \mapsto L$ *lexicographically dominates* (with respect to \leq_O) another classification λ' , denoted $\lambda \succeq_O \lambda'$, iff $\forall A \in \mathcal{A} : (\forall A' \in \mathcal{A}, A' \neq A, A' \leq_O A : \lambda'(A') = \lambda(A')) \Rightarrow \lambda(A) \succeq \lambda'(A)$. In other words, $\lambda \succeq_O \lambda'$ iff, for the least attribute A (in the total order \leq_O) for which λ and λ' differ, $\lambda(A) \succeq \lambda'(A)$.

Based on the above definition, a classification is said to be priority-minimal if it classifies the attributes whose visibility is more important as low as possible. This concept is made precise by the following definition.

DEFINITION 6.2 (Priority-Minimal Classification). *Given a set \mathcal{A} of attributes, classification lattice $\mathcal{L} = (L, \succeq)$, a set C of classification constraints over \mathcal{A} and \mathcal{L} , and a total ordering \leq_O on the elements of \mathcal{A} , a classification $\lambda : \mathcal{A} \mapsto L$ is priority-minimal with respect to \leq_O and C iff (1) $\lambda \models C$; and (2) $\forall \lambda' : \mathcal{A} \mapsto L$ such that $\lambda' \models C$, $\lambda \succeq_O \lambda' \Rightarrow \lambda' = \lambda$.*

It is easy to see that the definition of priority-minimal is stronger than the definition of minimal (Definition 2.4) and that any classification that is priority-minimal is also minimal (the converse does not necessarily hold). The proof is trivial by contradiction. Suppose the implication does not hold and consider a classification λ that is priority-minimal (with respect to some total order \leq_O on the attributes) but is not minimal. Then, there exists a classification $\lambda' \neq \lambda$ such that $\lambda' \models C$ and $\lambda \succeq \lambda'$, i.e., $\lambda(A) \succeq \lambda'(A), \forall A \in \mathcal{A}$. Hence $\lambda \succeq_O \lambda'$ and $\lambda' \neq \lambda$, which contradicts the assumption that λ is priority-minimal.

While the additional control offered by the concept of attribute priority is useful, the assumption of totally ordered attributes is likely too strong as a practical requirement. We can imagine instead that attribute priorities will form a partial order, reflecting the fact that, while some attributes are more important than others in terms of visibility, there may be no relative importance between other attributes. We can also imagine the priority order to be only partially specified (on attributes whose visibility is most important), while all attributes not explicitly mentioned are assumed to have the same priority (at the top of the attribute ordering). For instance, referring to the example in Figure 2, a priority order specification might say simply that `patient` \leq_O `illness`, meaning that the solution should guarantee first the maximum visibility of `patient`, then the maximum visibility of `illness`, then the visibility of the other attributes (in no particular order).

To account for this general situation, we extend the definition of priority-minimal to allow the given priority ordering on the attributes \mathcal{A} to be partial. We say that a total ordering \leq_O respects a partial ordering \leq_P if for all $A, A' \in \mathcal{A}$: $A \leq_P A' \Rightarrow A \leq_O A'$.

DEFINITION 6.3 (Partial-Priority-Minimal Classification). *Given a set \mathcal{A} of attributes, classification lattice $\mathcal{L} = (L, \succeq)$, a set C of classification constraints over \mathcal{A} and \mathcal{L} , and a partial ordering \leq_P on the elements of \mathcal{A} , a classification $\lambda : \mathcal{A} \rightarrow L$ is partial-priority-minimal with respect to \leq_P and C iff (1) $\lambda \models C$; and (2) $\forall \lambda' : \mathcal{A} \rightarrow L$ such that $\lambda' \models C$, (\forall total orders \leq_O respecting \leq_P , $\lambda \succeq_O \lambda'$) $\Rightarrow \lambda' = \lambda$.*

Definition 6.3 simply extends Definition 6.2 to the case where the ordering on attributes is a partial order. Again, the condition of partial-priority-minimal is stronger than simple minimality and any solution satisfying Definition 6.3 is also a minimal solution. More

precisely, it is a minimal solution preferred according to the visibility constraints specified by the given partial order on attributes.

With minor modifications Algorithm 4.1 can be used to compute partial-priority-minimal solutions. Here we sketch how such a modified algorithm would work. The enforcement of upper bound constraints is carried out as in Algorithm 4.1, since their enforcement is deterministic. For lower bound constraints, the algorithm is modified to use the incremental lowering process (forward propagation) on attributes in nondecreasing attribute priority order, as determined by the partial order \leq_P . More specifically, for some total order \leq_O on the attributes that respects \leq_P , the incremental lowering procedure (**try_to_lower**) is applied successively to each attribute from least to greatest according to \leq_O . The level of each attribute is lowered as far as possible before proceeding to the next attribute. In this way, each attribute is assigned the lowest level that satisfies the constraints, subject to the additional constraint that the levels of attributes (lower in attribute priority order) already assigned cannot be modified. We state without proof that the solution so computed is partial-priority-minimal (with respect to \leq_P). The time complexity of this computation is the same as that of computing a minimal solution for a set of cyclic constraints (analyzed in the appendix), $O(N_{ASHMc})$.

When the priority ordering on attributes is only partially specified (i.e., some subset of the attributes is not prioritized), the performance of the algorithm for computing partial-priority-minimal solutions can be improved by first executing the incremental lowering process as described only on the prioritized attributes. Then, procedure **compute_minimal_solution** from Algorithm 4.1 can be run unmodified. Intuitively, running the forward propagation approach on the prioritized attributes will set their final levels as low as possible. Then, the algorithm will proceed by executing **compute_minimal_solution** to determine a classification as before.

EXAMPLE 6.1. Consider the constraints in Figure 12 and assume the partial priority order $\text{patient} \leq_P \text{plan}$, $\text{plan} \leq_P \text{doctor}$ (with no other attributes prioritized). Enforcement of upper constraints is as illustrated in Example 4.1. The process of enforcing lower bound constraints is illustrated in Figure 14. The first phase of the process takes care of priorities, considering the prioritized attributes (`patient`, `plan`, and `doctor`) in nondecreasing priority order and executing the lowering (forward propagation process) on them. For each attribute, the table illustrate the calls to **try_to_lower** and their possible effects on levels together with the constraints that caused that effect (column *c*). After this forward propagation, the level of the prioritized attributes is final. The level of the other attributes is then computed evaluating lower bound constraints as already discussed in Example 4.1, causing the effects illustrated in Figure 14. The table at the bottom of the figure reports the levels of the attributes before and after the enforcement of lower bound constraints.

7. ARBITRARY PARTIAL ORDERS

The results presented thus far are based on the assumption of classification levels forming a lattice. We consider here the problem of determining a classification if the security levels do not form a lattice but may instead be an arbitrary poset. It turns out that the problem becomes intractable under this new condition, as the following theorem states.

COMPUTE_MINIMAL_SOLUTION (with priorities {patient \leq_P plan, plan \leq_P doctor})

<i>i</i>	<i>A</i>	calls to TRY_TO_LOWER	<i>c</i>	updates to levels
	patient	try_to_lower(patient,Financial)	<i>c</i> ₁₉	$\lambda(\text{patient}) = \text{Financial}$
		try_to_lower(patient,Public)		$\lambda(\text{employer}) = \text{Financial}$
	plan	try_to_lower(plan,Admin)	<i>c</i> ₂₁	$\lambda(\text{patient}) = \text{Public}$
		try_to_lower(plan,Financial)		$\lambda(\text{employer}) = \text{Public}$
		try_to_lower(plan,Public)		$\lambda(\text{plan}) = \text{Admin}$
	doctor	try_to_lower(doctor,Financial)	<i>c</i> ₁₁	$\lambda(\text{doctor}) = \text{Admin}$
		try_to_lower(doctor,Clinical)		$\lambda(\text{plan}) = \text{Financial}$
		try_to_lower(doctor,Research)		\emptyset
	8	doctor		
	division		<i>c</i> ₆ , <i>c</i> ₂₁	$\lambda(\text{division}) = \text{Research}$
	illness		<i>c</i> ₁₁ , <i>c</i> ₂₀ , <i>c</i> ₂₃	$\lambda(\text{illness}) = \text{Clinical}$
7	plan			
6	employer		<i>c</i> ₇ , <i>c</i> ₂₂	$\lambda(\text{employer}) = \text{Public}$
5	patient			
4	bill		<i>c</i> ₉ , <i>c</i> ₂₄	$\lambda(\text{bill}) = \text{Admin}$
3	insurance		<i>c</i> ₁₀ , <i>c</i> ₂₂	$\lambda(\text{insurance}) = \text{Financial}$
2	exam	try_to_lower(exam,Financial)	<i>c</i> ₁₆	\emptyset
		try_to_lower(exam,Clinical)		$\lambda(\text{exam}) = \text{Clinical}$
		try_to_lower(exam,Research)		$\lambda(\text{treatment}) = \text{Clinical}$
	treatment	try_to_lower(treatment,Research)	<i>c</i> ₁₃	$\lambda(\text{visit}) = \text{Clinical}$
	visit		<i>c</i> ₁₆	\emptyset
1	prescription		<i>c</i> ₂ , <i>c</i> ₁₄	$\lambda(\text{visit}) = \text{Clinical}$
			<i>c</i> ₁₂ , <i>c</i> ₁₇	$\lambda(\text{prescription}) = \text{Clinical}$

EFFECTS OF THE CLASSIFICATION PROCESS

Attribute	Result of compute_upper_bound	Result of compute_minimal solution (with priorities)
prescription	HMO	Clinical
exam	Admin	Clinical
treatment	Admin	Clinical
visit	Admin	Clinical
insurance	HMO	Financial
bill	HMO	Admin
patient	Admin	Public
employer	Admin	Public
plan	HMO	Financial
doctor	HMO	Clinical
division	Clinical	Research
illness	Clinical	Clinical

FIG. 14. Lower bound computation with priorities on the constraints of Figure 12

We define the problem POSET-ASSIGNMENT similarly to MIN-LATTICE-ASSIGNMENT, except that the constraint set is restricted to simple constraints, the partial order is not restricted to be a lattice, and the problem is stated as a decision problem. Given a partial order (P, \geq) and a set of constraints C , each constraint taking one of two forms: $A \geq A'$, $A \geq l$, where the A s are attributes, and l is a constant drawn from P , is there an assignment from attributes to members of P that satisfies all the constraints C ?

THEOREM 7.1. POSET-ASSIGNMENT is NP-complete.

Proof. The proof is presented in the appendix. ■

8. RELATED WORK

Inference problems have been studied extensively in the context of multilevel database systems. Most inference research addresses detection of inference channels within a database or at query processing time. Initial proposals in the first category [12, 24, 28, 33] analyze the database schema to locate inference channels based on semantic relationships between attributes. For instance, DISSECT [24], analyses the database schema to determine inference paths due to sequences of foreign key relationships, and signals the database administrator a possible inference problem whenever two database relations are connected by multiple paths at different classifications. These approaches are mostly intended to support multilevel schema design by identifying possible inference channels for the database administrator rather than automatically solving them. More recent approaches [7, 11, 17, 20, 37] extend the inference analysis to the consideration of database content (finer-grained inference control) and possibly external information. In some sense, these approaches are complementary to ours, as the information produced by them could be used as input to our approach for the definition of classification constraints. The proposals in the second category [10, 18, 21, 27, 34] evaluate database transactions to determine whether they lead to illegal inferences and, if so, disallow the query. The solutions investigated are not applicable in our context, where constraint processing is executed offline for the purpose of producing a classified database that prevents improper information leakage without the need of (expensive and often impractical or infeasible) runtime control and logging.

The work closest to ours is that of Su and Ozsoyoglu [30] and that of Stickel [29]. Su and Ozsoyoglu [30] consider the problem of upgrading data to block inference channels due to functional and multivalued dependencies. Their approach takes as input a set of attributes together with a proposed classification for them and a set of functional dependencies assumed to cause inference. It returns an alternative inference channel-free classification for the attributes, obtained by upgrading the classifications provided as input. Intuitively, each functional dependency corresponds to a lower bound constraint requiring the least upper bound of the security levels of a given set of attributes to dominate the security levels of the attribute functionally dependent on them. Minimization of information loss due to possible upgrading is determined based on the following optimality criteria. Each attribute at each possible given level is associated with a weight, based on the usage and importance of the attribute to the application. The optimal solution to be produced is the one that satisfies all the constraints while minimizing the difference between the total weight of attributes before and after the security level adjustment. Determining such a solution is an NP-hard problem, and the algorithm proposed in [30] finds it at the price of executing an

exhaustive search among all possible solutions. For the same problem, Stickel [29] within the context of DISSECT [24], computes the optimal solution by applying the Davis-Putnam approach to theorem proving; where the Davis-Putnam procedure is used to produce all the minimal solutions to a set of constraints, from which the optimal can then be chosen. Apart from the same computational complexity concerns associated with the other approaches mentioned, the work in [29, 30] has several drawbacks. First, it is generally far from obvious how to manipulate costs to achieve the desired classification behavior. Second, the proposed approaches work only under the assumption of totally ordered security levels. For the simple case of classification composed of pairs (security level, set of categories), Stickel suggests their approach could be applied by exploding the problems into one problem for each possible category in the category set. However, details are not given there. Also, such a solution would add another dimension to the computational complexity of finding the optimal solution. The notion of minimal classification used in this paper was first proposed in [6], within the content-based classification of existing data repositories. There, the approach to the determination of a preferred minimal solution still computes all the possible classifications, thus bearing an exponential cost, and it, like others, is limited to the consideration of totally ordered security levels. As a final remark, none of the previous work considered upper bound constraints.

9. 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 the 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.

The work presented in this paper leaves space for further work. Work to be investigated include, the investigation of criteria and possible heuristics towards the determination of a possible preferred solution; the investigation of incremental solutions to the consideration of updates to the classification constraints; the investigation of the applicability of the approach to nonmandatory policies and its possible extensions to discretionary domains.

APPENDIX: PROOFS

A.1. CORRECTNESS OF ALGORITHM 4.1

We first establish several lemmas used in the proof of the main theorem. The proofs often refer to *immediate* constraints on an attribute, by which we mean either lower bound constraints in which the attribute appears on the left-hand side or upper bound constraints in which the attribute is on the right-hand side.

Lemma A.1 establishes the correctness of **compute_upper_bounds**, showing that it succeeds in generating an initial classification mapping that satisfies the input constraints if and only if the constraints are consistent.

LEMMA A.1. *Let $C = C_{lower} \cup C_{upper}$ be a set of classification constraints over a set of attributes \mathcal{A} and a classification lattice $\mathcal{L} = (L, \succeq)$.*

- i) *If **compute_upper_bounds** terminates with failure, then C is inconsistent.*
- ii) *If **compute_upper_bounds** terminates with success, then the computed classification mapping $\lambda : \mathcal{A} \mapsto L$ satisfies C ($\lambda \models C$), and hence, C is consistent.*
- iii) *Procedure **compute_upper_bounds** always terminates.*

Proof.

i). We first prove that the following property holds of the current classification mapping λ throughout the computation of **compute_upper_bounds**:

For all mappings λ' such that $\lambda \not\preceq \lambda'$, there exists a constraint $c \in C$ such that $\lambda' \not\models c$.
(A.1.1)

In other words, at all times in **compute_upper_bounds** it is not possible to change the levels of any attributes to higher or incomparable levels without violating at least one constraint. We prove this property by induction, showing that if it holds before the modification of any $\lambda(A)$, it also holds after the change. At the start of the procedure, $\lambda(A) = \top$ for all attributes A , and the property trivially holds, since there is no mapping that λ does not dominate. Now, there are two points in **compute_upper_bounds** at which attributes' levels may be modified. The first is at the start of **compute_upper_bounds** itself, where each upper bound constraint is enforced, and the second is in the subprocedure **upper_bound**. In both cases, the modification results from a processing a constraint c of the form (lhs, A) , where $A \in \mathcal{A}$, and the level assigned to A is the greatest lower bound (glb) of the level of lhs and $\lambda(A)$. Let l denote the level of lhs under λ , and let $l' = l \sqcap \lambda(A)$. Let $\lambda' = \lambda$ except that $\lambda'(A) = l'$. Note that λ' is the mapping that results from ensuring the satisfaction of c . We analyze two cases according to the possible relationships between l and $\lambda(A)$.

- Case 1: $l \succeq \lambda(A)$. In this case, $l' = \lambda(A)$ and $\lambda' = \lambda$. Hence, λ is not modified, and the property continues to hold.

- Case 2: $l \not\preceq \lambda(A)$. In this case, $\lambda(A) \succ l'$. Let λ'' be any mapping such that $\lambda' \not\preceq \lambda''$. Suppose $\lambda'(A) \not\preceq \lambda''(A)$. If $\lambda(A) \succeq \lambda''(A)$, then $l \not\preceq \lambda''(A)$, since $\lambda'(A) = l'$ is the glb of l and $\lambda(A)$. Hence, $\lambda'' \not\models c$. Otherwise, $\lambda(A) \not\preceq \lambda''(A)$. Hence, $\lambda \not\preceq \lambda''$, and by hypothesis, there exists $c' \in C$ such that $\lambda'' \not\models c'$. In either case, the property holds for the modified mapping λ' . Suppose instead that $\lambda'(A) \succeq \lambda''(A)$. Then, for some $A' \neq A$, $\lambda'(A') \not\preceq \lambda''(A')$. Since $\lambda' = \lambda$ except on A , we have $\lambda'(A') = \lambda(A')$, $\lambda(A') \not\preceq \lambda''(A')$, and hence, $\lambda \not\preceq \lambda''$. By hypothesis, then, there exists $c' \in C$ such that $\lambda'' \not\models c'$, and the property again holds for the modified mapping λ' . This establishes property A.1.1.

Suppose now that **upper_bound** (and hence, **compute_upper_bounds**) terminates with failure. This means that a constraint $c = (lhs, rhs)$, such that $rhs \in L$ is a security level, was found not to be satisfied; that is, $\lambda \not\models c$. Since rhs is fixed, the only way to satisfy the constraint would be to suitably modify λ for some attribute(s) in lhs . Let $\lambda' : \mathcal{A} \mapsto L$ be any mapping from attributes in \mathcal{A} to levels in L . If $\lambda \succeq \lambda'$, then $\lambda' \not\models c$, since $\lambda \not\models c$. Otherwise, $\lambda \not\preceq \lambda'$. By the property just proved (A.1.1), there exists a constraint $c' \in C$ such that $\lambda' \not\models c'$. Hence, for any mapping λ' , there exists a constraint that cannot be satisfied, and therefore C is inconsistent.

ii). Assume that **compute_upper_bounds** terminates successfully. We show that the computed mapping λ satisfies C by induction on SCC index. That is, we show that, if at the start of iteration i (of the loop over SCCs in **compute_upper_bounds**) λ satisfies all immediate constraints on all attributes in all SCCs of index less than i , then at the end of that iteration λ satisfies all immediate constraints on all attributes in all SCCs of index less than or equal to i .

We begin by noting that before the first iteration ($i = 1$), $\lambda \models C_{upper}$, since for each constraint $c \in C_{upper}$ of the form (l, A) , $\lambda(A)$ is assigned $\lambda(A) \sqcap l$, so that $l \succeq \lambda(A)$. Now consider an arbitrary iteration i . The following properties are readily established:

1. Every constraint on every attribute in $scc[i]$ is checked for satisfaction under λ . This follows from the fact that **compute_upper_bounds** calls **upper_bound** on all attributes A for which $visit[A]$ is 0, and $visit[A]$ is set to a nonzero value only by **upper_bound** itself.

2. For any constraint c of the form (lhs, rhs) (on the attribute A being processed) found to be violated by λ , $\lambda(rhs)$ is assigned the glb of its current level and that of lhs , satisfying c . Furthermore, any other constraint with the same rhs remains satisfied, if it was previously, since the new level of rhs is dominated by its previous level. Now, from the properties of the DFS procedures [32], we know that the SCC index of rhs must be greater than or equal to that of A . If it is equal, a recursive call to **upper_bound** on rhs ensures that all immediate constraints on it are (re)checked.

3. From the properties of the SCCs computed by the DFS procedures, we know that the levels of any attributes in an SCC of index less than i are unmodified after iteration i , since such attributes are not reachable by any constraints on attributes in $scc[i]$.

4. Since the levels assigned to attributes can only be lowered, the upper bound constraints remain satisfied.

From properties 1, 2, and 4, we can conclude that all immediate constraints on all attributes in $scc[i]$ are satisfied after iteration i . From properties 3 and 4, we can conclude that all constraints on all attributes in SCCs of index less than i remain satisfied. Hence, the induction step is proved.

iii). Procedure **compute_upper_bounds** is composed of three loops over finite sets. Termination of the procedure is straightforward to establish, except perhaps for the recursive subprocedure **upper_bound**, called in the third loop, for each SCC $scc[i]$. **upper_bound**(A, i) is called recursively only when an attribute A in the SCC being processed ($scc[i]$) is assigned a level strictly lower than the one it currently has. Each attribute can be lowered only a finite number of times (bounded by the height of the lattice), and the number of attributes in each SCC is finite. Hence, the number of times **upper_bound** can be called in an SCC is finite. ■

Lemma A.1 shows that, if the algorithm continues beyond the end of **compute_upper_bounds**, then the remainder of the algorithm starts from a point at which λ satisfies the constraints (and otherwise the constraints are inconsistent). The remaining lemmas are used in the proof of the main theorem to show that key parts of the final phase of the algorithm (**compute_minimal_solution**) preserve the property that, after every modification, λ remains a solution. First, we prove Lemma A.2, which shows that arguments about the satisfaction of generated classification assignments can be made locally. That is, it establishes that, if any changes to a solution mapping that are limited to a subset of

the attributes result in satisfaction of the immediate constraints on those attributes, then the modified mapping remains a solution for all constraints.

LEMMA A.2. *Given (1) a set C of constraints on a set \mathcal{A} of attributes and (2) a subset \mathcal{A}' of \mathcal{A} , let λ be an assignment of levels to attributes such that $\lambda \models C$ and λ' be an assignment such that $\lambda \succeq \lambda'$ and that differs from λ only on attributes in \mathcal{A}' . Let C' denote the set of immediate constraints on attributes in \mathcal{A}' , that is, $C' = \{(lhs, rhs) \in C \mid lhs \cap \mathcal{A}' \neq \emptyset\}$. Then, $\lambda' \models C$ if and only if $\lambda' \models C'$.*

Proof.

(If): Assume that λ' satisfies C' . Let $c = (lhs, rhs)$ be an arbitrary constraint in C . If $c \in C'$, then by assumption, $\lambda' \models c$. Otherwise, $c \notin C'$, so $lhs \cap \mathcal{A}' = \emptyset$, 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, $\lambda' \models c$.

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

■

The following lemma shows that any change to a solution λ resulting from the output of procedure **try_to_lower** in Algorithm 4.1 preserves λ as a solution.

LEMMA A.3. *Let AS be the set of pairs of the form (A', l') returned by **Try** (A, l) . If $AS \neq \emptyset$ the assignment obtained by replacing $\lambda(A')$ with $\lambda(A') = l'$ for all $(A', l') \in AS$ satisfies all immediate lower bound constraints on attributes in $\text{scc}[i]$, where $\text{scc}[i]$ is the SCC containing A .*

Proof. The following properties are readily established.

1. When a pair $(A', l') \in \text{Tocheck}$ is selected, all immediate lower bound constraints on A' are checked for satisfaction.
2. If any constraint c of the form (lhs, rhs) is found to be violated, either the right-hand side is done and cannot be satisfied (in which case **try_to_lower** returns \emptyset) or a pair of the form (rhs, l'') is added to *Tocheck*, where l'' is the greatest level that can be assigned to rhs and still satisfy all constraints checked up to that point.
3. From the established properties of the DFS procedures [32], we know that every attribute A' in the SCC containing A is reachable from A' .
4. For any attribute A' , at most one pair of the form (A', l') can exist in *Tocheck* or *Tolower* (but not both) at any time. This follows immediately from the fact that, whenever a pair involving A' is added to one set, any pair involving A' in the other set (if one exists) is first removed.
5. For any pair of the form $(A', l') \in \text{Tocheck}$ and any pair of the form (A', l'') subsequently added to *Tocheck*, $l' \succ l''$.
6. Every pair $(A', l') \in \text{Tocheck}$ is eventually selected.

Properties 1, 2, 3, and 6 together show that any constraint that could be violated by any assignment modification (represented in *Tolower*) is checked, and if possible, another modification is made to satisfy the constraint. Properties 1 through 4 together show that, at all times in **try_to_lower** the modifications to λ represented

in *Tolower* are such that λ satisfies all immediate constraints on all attributes in the SCC containing A , provided that all pairs in *Tocheck* also represent satisfying assignments. At the end of the procedure, *Tocheck* is empty, so the lemma holds. ■

Theorem 5.1 (Correctness). *Algorithm 4.1 solves MIN-LATTICE-ASSIGNMENT. That is, given a set C of classification constraints over a set A of attributes and a classification lattice $\mathcal{L} = (L, \succeq)$, Algorithm 4.1 generates a minimal classification mapping $\lambda : A \mapsto L$ that satisfies C , or terminates with failure if the set C is inconsistent.*

Proof.

We show that **compute_minimal_solution** produces an assignment λ that (i) satisfies C if one exists (ii) any attribute A for which $done[A] = \text{TRUE}$ has been assigned a minimal level that satisfies its constraints. We show this by induction on the outermost loop of **compute_minimal_solution** on SCCs. Initially λ assigns \top to every attribute, which trivially satisfies all lower bound constraints in C , and for every $l \in L$ we have the assignments $\lambda(l) = l$ and $done[l] = \text{TRUE}$, which trivially satisfies the minimality requirement.

By Lemma A.1, **compute_upper_bounds** always terminates and returns failure if the constraints are inconsistent, otherwise producing an assignment which satisfies C (but which is usually not minimal). Inductively, we assume that at the start of an iteration of the outermost loop λ is a solution, and that any attribute marked *done* has been assigned a minimal satisfying level, and we must show that λ is a solution at the end of that iteration, and any attribute marked *done* at the end of that iteration has been assigned a minimal satisfying level. By Lemma A.2 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 SCC, (2) the level assigned by λ to any attribute is never raised, and (3) all direct constraints on attributes of that SCC are satisfied at the end of any iteration.

Let i be the SCC index in the outermost loop of **compute_minimal_solution** and S be the list $scc[i]$. We argue by induction on the second-level loop (**For** $A \in scc[i]$), and show that λ satisfies C at the end of each iteration of this inner loop, and further that the minimality requirement is met for all attributes that are done. Let A be an arbitrary attribute in S . Consider $Constr[A]$. If every $(lhs, rhs) \in Constr[A]$ is such that $done[rhs] = \text{TRUE}$, we simply take the least upper bound of a set of predetermined levels, and since we are working in a lattice, a unique least upper bound l exists. Otherwise, there is at least one $(lhs, rhs) \in Constr[A]$ such that $done[rhs] = \text{FALSE}$. So, after processing each $c \in Constr[A]$, $done[A] = \text{FALSE}$, and we proceed from the initialization of $DSet$. At this point in the computation l holds a lower bound on the level that may be assigned to A , $\lambda(A) \succeq l$, and $DSet$ is initialized to the set of levels immediately below $\lambda(A)$ and that dominate l . We argue that λ satisfies C at the end of any iteration of the inner while-loop, and that the minimality requirement is met. If **try_to_lower** fails for every $l'' \in DSet$, no assignments in λ are modified, and thus, C remains satisfied, and since all lower levels failed, we have found a minimal assignment for A . Otherwise, by Lemma A.3, **try_to_lower** returns a set of pairs of the form (A', l') , where $A' \in scc[i]$, $\lambda(A') \succeq l'$, and such that replacing $\lambda(A')$ by $\lambda(A') = l'$ for all such A' satisfies all constraints on attributes in $scc[i]$. The inner while-loop concludes by making this replacement and resetting $DSet$ to levels immediately below $\lambda(A)$. Hence, λ satisfies C at the end of each iteration of the while-loop, and any attribute A for which $done[A] = \text{TRUE}$ has been assigned a minimal

level that satisfies its constraints. By induction λ satisfies C at the end of the enclosing for-loop, and thus at the end of the outermost loop.

Termination. There are two aspects of termination that are not obvious once one takes into consideration the termination argument in Lemma A.1. First, the while-loop at the end of **compute_minimal_solution** 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_to_lower** 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_to_lower** 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. ■

A.2. ALGORITHM 4.1 IS LOW-ORDER POLYNOMIAL

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} . Define M to be maximum, for all paths from the top to the bottom of a lattice, of the sum of the branching factor of each element of the path. M is no greater than BH , and is also no greater than the size of \mathcal{L} (number of elements + size of the immediate successor relation).

Theorem 5.2 (Complexity). *Algorithm 4.1 solves any instance $(\mathcal{A}, \mathcal{L}, C)$ of the problem MIN-LATTICE-ASSIGNMENT in $O(N_{\mathcal{A}}SHMc)$ time, and, if the set of constraints C is acyclic, in $O(SMc)$ 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 in **main**, apart from **dfs_visit** and **dfs_back_visit**, require (in total) time proportional to $S + N_L$. Procedures **dfs_visit** and **dfs_back_visit** themselves are simply a minor adaptation of Tarjan’s linear-time SCC computing algorithm [32], and require time proportional to S . In the acyclic case, **compute_upper_bounds** processes all constraints in C once for each attribute on the left-hand side, and, for each constraint, may perform one lub and one glb operation. Thus, it requires time proportional to $S + N_Cc$, which is certainly $O(Sc)$. In the cyclic case, **compute_upper_bounds** may check all constraints in C multiple times per attribute. Whenever the level of the attribute A on the right side of some constraint is lowered and is in the same SCC as the attribute on the left side for which the constraint is being checked, all constraints on A must be rechecked. Since this rechecking is done only upon lowering the level of the attribute, the number of times an attribute’s constraints can be rechecked is

bounded by H , the height of the lattice. Thus, in the cyclic case the enforcement of upper bound constraints can be accomplished in $O(SHc)$ time. Overall, the time complexity for all parts of the algorithm before **compute_minimal_solution** is $O(Sc)$ in the acyclic case and $O(SHc)$ in the cyclic case.

It remains to determine the complexity of **compute_partial_lubs** and **compute_minimal_solution**. For each constraint (lhs, rhs) , **compute_partial_lubs** computes and stores a number of partial lubs requiring $O(|lhs|)$ space and $O(|lhs|c)$ time. Overall, the time complexity of **compute_partial_lubs** is $O(Sc)$.

For **compute_minimal_solution** 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_A times overall. In the acyclic case, note that every attribute is its own SCC. When considering any attribute A in **compute_minimal_solution**, 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**. The **minlevel** procedure first performs several constant-time initializations and one lub operation. The remainder of **minlevel** considers overall at most M security levels, each involving a lub operation. The time complexity of **minlevel**, then, is $O(Mc)$. Since the cost of **minlevel** is at least as high as that of a lub operation, the worst-case cost of the inner loop is when all S iterations involve **minlevel**, $O(SMc)$, which, for acyclic constraints, dominates the time complexity of all other parts of the algorithm.

For cyclic constraints, the cost due to the innermost for-loop of **compute_minimal_solution** 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_to_lower** computation. In the worst case, **try_to_lower** 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_to_lower**, but on average, **try_to_lower** may process as many as half the constraints involved in the SCC. Now, it can happen that, for some pair $(A, l) \in T_{lower}$ and level l' , (A, l) is removed from T_{lower} and (A, l') added to T_{check} , implying the reprocessing of constraints on A . For any attribute, this reintroduction into T_{check} can happen at most H times, 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 T_{lower} and T_{check} , the only remaining nonconstant cost comes from at most two glb operations. The time complexity of **try_to_lower**, then, is $O(HSc)$, and that of the while-loop in **compute_minimal_solution** is $O(HMSc)$. Over all attributes in the SCC, the time complexity of **compute_minimal_solution** due to the while-loop is $O(N_A HMSc)$,

which dominates the cost due to the innermost for-loop of **compute_minimal_solution**. ■

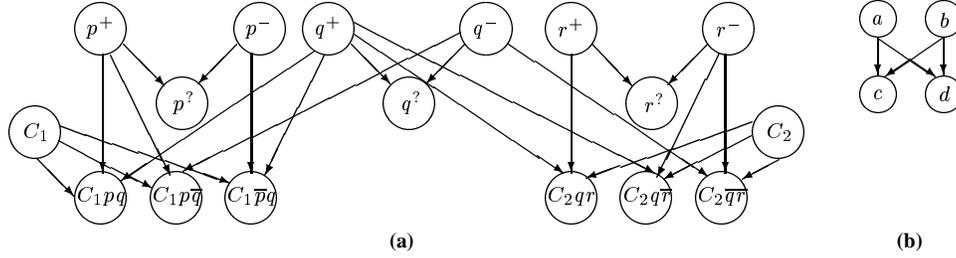


FIG. A.1. Poset for $(p \vee q) \wedge (q \vee \bar{r})$ (a), and four-element poset (b).

A.3. CLASSIFICATION ASSIGNMENT IN A POSET IS NP-COMPLETE

We define the problem POSET-ASSIGNMENT similarly to MIN-LATTICE-ASSIGNMENT, except that the constraint set is restricted to simple constraints, the partial order is not restricted to be a lattice, and the problem is stated as a decision problem. Given a partial order (P, \geq) and a set of constraints C , each constraint taking one of two forms: $A \geq A'$, $A \geq l$, where the A s are attributes, and l is a constant drawn from P , is there an assignment from attributes to members of P that satisfies all the constraints C ?

Theorem 7.1. [POSET-ASSIGNMENT is NP-complete.]

Informally, to see why POSET-ASSIGNMENT is a hard problem, consider a poset of security levels with four elements with two upper elements each dominating the two lower elements, as depicted in Figure A.1(b). If an attribute is known to dominate the two lower elements, in the final analysis that attribute must be assigned to one of the two upper elements, and thus a choice must be made. Multiple such choices may result in an exponential number of possibilities. Below we sketch a proof using this kind of choice to encode propositional truth or falsity in satisfiability problems.

We give a polynomial reduction from 3-SAT, demonstrating NP-hardness. We first define a partial order (the security levels), beginning with the empty set C , and for each clause $Clause_i$ of the form $P_{i_1} \vee P_{i_2} \vee P_{i_3}$, where each literal P_{i_j} is either the propositional variable p_{i_j} or its negation \bar{p}_{i_j} , we add the element named C_i to C , and further add seven more elements to C , one for each truth assignment that satisfies the clause. For convenience, we name these seven elements by simply concatenating the names of the clauses with the literals they contain, using overbars to denote negation: “ $C_i P_{i_1} P_{i_2} P_{i_3}$ ”, “ $C_i P_{i_1} P_{i_2} \bar{P}_{i_3}$ ”, “ $C_i P_{i_1} \bar{P}_{i_2} P_{i_3}$ ”, 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.

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 propositional variable p_j , $p_j^+ \geq p_j^?$ and $p_j^- \geq p_j^?$. We define the relation R_{clause} to include, for each element of the form $C_i P_{i_1} P_{i_2} P_{i_3}$ in C , $C_i \geq C_i P_{i_1} P_{i_2} P_{i_3}$ (seven for each clause). We also define the relation R_{true} to include, for each element of the form $C_i P_{i_1} P_{i_2} P_{i_3}$ in C , $P_{i_j}^+ \geq C_i P_{i_1} P_{i_2} P_{i_3}$ for each j , $1 \leq j \leq 3$, such that $P_{i_j} = p_{i_j}$ (i.e., p_{i_j} occurs as a positive literal in the corresponding clause). Similarly, we define the relation R_{false} to include, for each element of the form $C_i P_{i_1} P_{i_2} P_{i_3}$ in C , $\bar{P}_{i_j}^- \geq C_i P_{i_1} P_{i_2} P_{i_3}$ for each j , $1 \leq j \leq 3$, such that $P_{i_j} = \bar{p}_{i_j}$ (i.e., p_{i_j} occurs as a negative literal in the corresponding clause). The final partial order

of interest will be made up of elements of C , related by $R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false}$. There are eight ($= 2^3$) elements in C for each 3-SAT clause, plus three elements for each propositional variable. The partial order has height one. Figure A.1(a) displays the partial order produced for the SAT problem $(p \vee q) \wedge (q \vee \bar{r})$. Clauses of length two were used in the figure to improve readability.

To simplify our arguments about the reduction, we first define a set of constraints that does not conform to the syntactic restrictions of POSET-ASSIGNMENT, in that constraints of the form $l \geq A$ (upper bound constraints) are used. We later show how these constraints can be replaced with an equivalent set that does conform to the definition of POSET-ASSIGNMENT. To form the constraints, we use a set of attributes, one wp_j for each propositional variable p_j , and one wc_i for each clause $Clause_i$. We define a set of inequations C_{clause} to include, for each clause $Clause_i$, the constraint $C_i \geq wc_i$, and for each literal P_{i_j} in that clause, $wp_{i_j} \geq wc_i$. We also define a set of inequations C_{prop} to include, for each propositional variable p_j , the constraint $wp_i \geq p_i^?$. Thus there are four constraints in C_{clause} per 3-SAT clause, and one constraint in C_{prop} for each propositional variable. Continuing with our simple example, $(p \vee q) \wedge (q \vee \bar{r})$, the inequations $C_{clause} = \{C_1 \geq wc_1, wp_p \geq wc_1, wp_q \geq wc_1, C_2 \geq wc_2, wp_q \geq wc_2, wp_r \geq wc_2\}$, and $C_{prop} = \{wp_p \geq p^?, wp_q \geq q^?, wp_r \geq r^?\}$.

We claim that the POSET-ASSIGNMENT problem given by the partial order $(C, R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false})$, with the constraints $C_{prop} \cup C_{clause}$ has a solution if and only if the original 3-SAT problem has one. This may be observed by noting that every attribute wc_i must be assigned some $C_i P_{i_1} P_{i_2} P_{i_3}$, since wc_i must be lower than C_i and some propositions. Also, the only $C_i P_{i_1} P_{i_2} P_{i_3}$ that exist in C correspond to assignments of propositional variables that satisfy the clause. Further, wp_j must be assigned either p_j^+ or p_j^- . We claim there is a correspondence between a propositional variable p_j being assigned true (or false, resp.) in the 3-SAT problem, and wp_j being assigned P_j^+ (P_j^- , resp.) in the POSET-ASSIGNMENT problem. Thus one may see that a solution to the 3-SAT problem may be derived from any solution to the constructed POSET-ASSIGNMENT problem and vice versa.

Note that the constraints of the form $C_i \geq wc_i$ are upper bound constraints and do not conform to the restrictions on the constraints in the definition of POSET-ASSIGNMENT. Each such constraint can be replaced by a set of constraints of the required form as follows. For each $Clause_i$, create one new attribute wu_i and seven constraints of the form $wu_i \geq C_i P_{i_1} P_{i_2} P_{i_3}$, one for each of the seven clause elements for $Clause_i$ in the partial order. These lower bound constraints have the effect of forcing wu_i to have only one possible assignment in the partial order, namely C_i . Now, the constraints of the form $C_i \geq wc_i$ used in the reduction can be replaced by the equivalent constraints $wu_i \geq wc_i$, which are of the form required by the definition of POSET-ASSIGNMENT.

Finally, we note that POSET-ASSIGNMENT is in NP, since we can guess a solution and check it in polynomial time.

Using results of Pratt and Tiuryn [22], this result can be improved to apply to small fixed partial orders, including the four-element partial order of security levels with two upper elements each dominating the two lower elements (Figure A.1(b)).

REFERENCES

1. H. Ait-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.

2. D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, The Mitre Corporation, March 1976.
3. S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
5. S. Dawson, S. De Capitani di Vimercati, P. Lincoln, and P. Samarati. Minimal data upgrading to prevent inference and association attacks. In *Proc. of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, May 31-June 3 1999.
6. S. Dawson, S. De Capitani di Vimercati, and P. Samarati. Specification and enforcement of classification and inference constraints. In *Proc. of the 20th IEEE Symposium on Security and Privacy*, pages 181–195, Oakland, May 1999.
7. H.S. Delugach and T.H. Hinke. Wizard: A database inference analysis and detection system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):56–66, February 1996.
8. Dep. Defense, National Computer Security Center, Standard DOD 5200.28-STD, 1985. *Department of Defense Trusted Computer System Evaluation Criteria*.
9. D.D. Ganguly, C.K. Mohan, and S. Ranka. A space-and-time-efficient coding algorithm for lattice computations. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):819–829, October 1994.
10. J.T. Haigh, R.C. O'Brien, and D.J. Thomsen. The LDV secure relational DBMS model. In S. Jajodia and C.E. Landwehr, editors, *Database Security, IV: Status and Prospects*, pages 265–279, North-Holland, 1991. Elsevier Science Publishers.
11. J. Hale and S. Sheno. Catalytic inference analysis: Detecting inference threats due to knowledge discovery. In *Proc. of the 1997 IEEE Symposium on Security and Privacy*, pages 188–199, Oakland, CA, May 1997.
12. T. Hinke. Inference aggregation detection in database management systems. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 96–107, Oakland, April 1988.
13. 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.
14. 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.
15. 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.
16. 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.
17. D.G. Marks. Inference in MLS database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):46–55, February 1996.
18. C. Meadows. Extending the Brewer-Nash model to a multilevel context. In *Proc. of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 95–102, Oakland, CA, May 1990.
19. J. K. Millen. Models of multilevel computer security. Technical Report MTR-10537, The Mitre Corporation, January 1989. Also in *Advances in Computer Security*, Vol. 28, Academic Press.
20. 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.
21. A. Motro, D.G. Marks, and S. Jajodia. Enhancing the controlled disclosure of sensitive information. In *Proc. of the Fourth European Symposium on Research in Security and Privacy*, pages 290–303, September 1996.
22. V.R. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28(1–2):165–182, November 1996.
23. 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.
24. X. Qian, M.E. Stickel, P.D. Karp, T.F. Lunt, and T.D. Garvey. Detection and elimination of inference channels in multilevel relational databases. In *Proc. of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 196–205, Oakland, CA, May 1993.
25. P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2172. Springer-Verlag, 2001.
26. R. Sandhu and F. Chen. The multilevel relational (MLR) data model. *ACM Transactions on Information and System Security*, 1(1):93–132, November 1998.

27. G.L. Sicherman, W. de Jonge, and R.P. van de Riet. Answering queries without revealing secrets. *ACM Transactions on Database System*, 8(1):41–59, March 1983.
28. G.W. Smith. Modeling security-relevant data semantics. In *Proc. of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 384–391, Oakland, CA, 1990.
29. 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.
30. 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.
31. M. Talamo and P. Vocca. A data structure for lattice representation. *Theoretical Computer Science*, 175(2):373–392, April 1997.
32. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.
33. B. Thuraisingham. The use of conceptual structures for handling the inference problem. In C.E. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 333–362, North-Holland, 1992. Elsevier Science Publishers.
34. B. Thuraisingham and W. Ford. Security constraint processing in a multilevel secure distributed database management system. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):274–293, April 1995.
35. G. Wiederhold and M. Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, September-October 1997.
36. 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.
37. R.W. Yip and K.N. Levitt. Data level inference detection in database systems. In *Proc. of the 11th IEEE Computer Security Foundations Workshop*, pages 179–189, Rockport, Massachusetts, June 1998.