Inference Control in Logic Databases as a Constraint Satisfaction Problem

Joachim Biskup, Dominique Marc Burgard, Torben Weibert*, and Lena Wiese

Fachbereich Informatik, Universität Dortmund, 44221 Dortmund, Germany
{biskup,burgard,weibert,wiese}@ls6.cs.uni-dortmund.de

Abstract. We investigate inference control in logic databases. The administrator defines a confidentiality policy, i. e., the pieces of information which may not be disclosed to a certain user. We present a static approach which constructs an alternative database instance in which the confidential information is replaced by harmless information. The construction is performed by the means of constraint programming: The task of finding an appropriate database instance is delegated to a hierarchical constraint solver. We compare this static approach to a dynamic inference control mechanism – Controlled Query Evaluation – investigated in earlier work, and we also point out possible extensions which make use of the various opportunities offered by hierarchical constraint solvers.

Keywords: Inference control, confidentiality, logic databases, constraint satisfaction problems, constraint hierarchies.

1 Introduction

A key feature of a secure information system is preservation of *confidentiality*: Each user must only learn the information he is allowed to. Traditional approaches rely on static access rights assigned to the *data*, and suffer from the *inference problem* [1]: The user may combine several pieces of accessible information in order to infer confidential information. For example, the two pieces of data "Alice is a manager" and "a manager's salary is $50,000" can be easily combined to the information that Alice's salary must be $50,000.

This problem can be overcome by a proper *inference control mechanism*: The administrator defines a *confidentiality policy* which specifies which pieces of *information* may not be disclosed. The inference control mechanism will then make sure that this confidential information cannot be inferred from the data returned to the user. Basically, there are *dynamic* and *static* approaches to the inference problem. A dynamic inference control mechanism monitors the queries and answers during runtime, and possibly distorts or filters part of the answers. On the other hand, a static approach modifies the original data such that the confidential information is removed or replaced, and queries can be processed in the ordinary manner without the need for any additional processing at runtime.

* This author is funded by the German Research Foundation (DFG) under Grant No. BI-311/12-1.

P. McDaniel and S.K. Gupta (Eds.): ICISS 2007, LNCS 4812, pp. 128–142, 2007.

Controlled Query Evaluation (CQE) [2] has been designed as a dynamic approach to the inference problem in logical databases. After each query, the system checks whether the answer to that query – combined with the previous answers and possible a priori assumptions – would enable the user to infer any secret information. If so, the answer is either refused or modified. Finally, the answer is stored in a log file in order to be accounted for later. CQE has been studied under various parameters [2,3,4], and there is also a static, SAT-solver based approach to CQE [5].

In this paper, we pick up the framework of CQE and present a static approach in which an alternative database instance is constructed from the original instance which does not contain any confidential information anymore. As opposed to [5], finding such a database instance is achieved by modelling the requirements as a *constraint satisfaction problem (CSP)* [6,7]. A *constraint solver* is a piece of software which tries to find an assignment over a set of variables such that a set of user-defined *constraints* is satisfied. In particular, *boolean constraint solvers* operate on the domain $\{true, false\}$ (meaning that each variable is assigned a value of either *true* or *false*), and allows us to specify the constraints as a set of boolean formulas. A problem arises in case the constraints are inconsistent, for example, if one constraint demands that $a = true$, and another constraint demands that $a = false$. In this situation, a *hierarchical constraint solver* can be used to identify an assignment which satisfies only part of the constraints, according to some previously established hierarchy. The concept of constraints can be found in various research fields of security, for example in the context of role-based access control [8,9] or secure workflow models [10].

The paper is outlined as follows: Section 2 presents the logical framework and the declarative requirements for a confidentiality-preserving inference control mechanism. In Section 3, we recall Controlled Query Evaluation as a dynamic enforcement method. The foundations of hierarchical constraint networks are presented in Section 4. In Section 5, we show how to use hierarchical constraint networks in order to construct a suitable alternative database instance. A comparison of this static approach to the existing dynamic mechanisms can be found in Section 6. In Section 7, we propose some extensions which further exploit the abilities of hierarchical constraint solvers. We finally conclude in Section 8.

2 Declarative Framework

We consider complete logic databases, founded on some logic $\mathcal{L}$, for example propositional or first-order logic. Let S `model_of` Φ denote that the structure S is a model of the sentence Φ wrt. to the semantics of the logic under consideration. Let the logical implication operator $\models$ be defined as usual: A set of sentences Σ implies a single sentence Φ ($\Sigma \models \Phi$) iff each structure which is a model of Σ is also a model of Φ.

Definition 1 (Logic databases and ordinary query evaluation). *A* database instance *db is a structure of the logic under consideration. The* database schema *DS captures the universe of discourse and is formally defined as the set of*

all instances. A (closed) query *is a sentence Φ. It is evaluated within a database instance db by the function*

$$eval(\Phi) : DS \to \{true, false\} \text{ with}$$
$$eval(\Phi)(db) := \begin{cases} true & \text{if } db \texttt{ model\_of } \Phi \\ false & \text{otherwise} \end{cases} \quad (1)$$

We also use an alternative evaluation function which returns the query or its negation, respectively:

$$eval^*(\Phi) : DS \to \{\Phi, \neg\Phi\} \text{ with}$$
$$eval^*(\Phi)(db) := \begin{cases} \Phi & \text{if } db \texttt{ model\_of } \Phi \\ \neg\Phi & \text{otherwise} \end{cases} \quad (2)$$

Definition 2 (Confidentiality policy). *The* confidentiality policy *is a set*

$$policy := \{\Psi_1, \ldots, \Psi_m\}$$

of potential secrets*, each of which is a sentence of the logic under consideration, with the following semantics: In case Ψ_i is true in the actual database instance db, the user may not learn this fact. Otherwise, if Ψ_i is false in db, this fact may be disclosed to the user. Accordingly, the user may believe that Ψ_i is false even if it is actually true.*

Example 3. Given a database which holds the medical record of some person, the confidentiality policy given by

$$policy := \{aids, cancer\}$$

defines that the user may not learn that the person suffers from aids, and may neither learn that the person suffers from cancer. In case the person does not suffer from one of these diseases, that information may be disclosed to the user.

The aim of an inference control mechanism is to protect the potential secrets in the aforementioned manner. We abstractly formalize an inference control mechanism as a function

$$f(Q, db, prior, policy) := \langle ans_1, \ldots, ans_n \rangle$$

where

- $Q = \langle \Phi_1, \ldots, \Phi_n \rangle$ is a (finite) query sequence,
- *db* is the actual database instance,
- *prior* is the user's *a priori assumptions*, given as a set of sentences in the logic under consideration, and
- *policy* is the confidentiality policy.

The function returns a sequence of answers, where each $ans_i \in \{\Phi_i, \neg\Phi_i\}$.[1] The answers are to be generated iteratively, i. e., the i-th answer must be returned before the $i+1$-th query is received.

We assume that each enforcement method f goes along with a function

$$precond(db, prior, policy) \in \{true, false\}$$

which defines the admissible arguments for f. For example, an enforcement method could refuse to start a session if any of the potential secrets can already be inferred from the a priori assumptions in the first place. Based on this abstract definition, we can introduce our notion of confidentiality.

Definition 4 (Confidentiality of an enforcement method). *An enforcement method f preserves* confidentiality *if and only if*

for all finite query sequences Q,
for all instances db,
for all confidentiality policies $policy$,
for all potential secrets $\Psi \in policy$,
for all sets of a priori assumptions $prior$
so that $(db, prior, policy)$ satisfies the precondition,
there exists an instance db'
so that $(db', prior, policy)$ satisfies the precondition,
and the following two conditions hold:
(a) [$(db, policy)$ and $(db', policy)$ produce the same answers]
$f(Q, db, prior, policy) = f(Q, db', prior, policy)$
(b) [Ψ is false in db']
$eval(\Psi)(db') = false$

Condition (a) guarantees that db and db' are indistinguishable to the user; he cannot tell whether db or db' is the actual database instance. Condition (b) makes sure that Ψ is *false* in db'; as the user considers db' as a possible actual database instance, he cannot rule out that Ψ is actually *false*.

3 A Dynamic Approach – Controlled Query Evaluation

We briefly recall Controlled Query Evaluation, in particular the uniform lying method for known potential secrets in complete databases, as found in [11,2]. This enforcement method keeps a *log file* of the past answers, and uses logical implication in order to detect threats to the confidentiality policy.

The log file log_i is a set of sentences of the logic under consideration, initialized with the a priori assumptions:

$$log_0 := prior$$

[1] Previous work [2] additionally uses the special symbol `mum` to indicate a refused answer; however, the present paper does not consider refusal.

After each query Φ_i, the answer ans_i is added to the log file:

$$log_i := log_{i-1} \cup \{ans_i\}$$

The uniform lying method makes sure that the log file does never imply the information that at least one potential secret must be *true*, by keeping

$$log_i \not\models pot\_sec\_disj \quad \text{with } pot\_sec\_disj = \bigvee_{\Psi \in policy} \Psi$$

as an invariant throughout the query sequence. For the a priori assumptions, the invariant is enforced by the precondition

$$precond(db, prior, policy) := prior \not\models pot\_sec\_disj.$$

Having received the query Φ_i, an appropriate answer is chosen so that the invariant is preserved: If the actual value $eval^*(\Phi_i)(db)$ does not lead to a violation, it is returned to the user. Otherwise, the negation of the actual value is returned as the answer, i. e., a lie is issued.

$$ans_i := \begin{cases} eval^*(\Phi_i)(db) & \text{if } log_{i-1} \cup \{eval^*(\Phi_i)(db)\} \not\models pot\_sec\_disj \\ \neg eval^*(\Phi_i)(db) & \text{otherwise} \end{cases}$$

Theorem 5. *The uniform lying method for known potential secrets preserves confidentiality in the sense of Definition 4.*

The full proof can be found in [2]. We give a short sketch here. Consider that $log_n \not\models pot\_sec\_disj$. This means that there must be a structure db' which is a model of log_n but not a model of pot_sec_disj, and thus also no model of Ψ for each particular $\Psi \in policy$. This satisfies condition (b) of Definition 4. It can also be shown that the same answers are returned under db and db', which satisfies condition (a).

4 Constraint Satisfaction Problems

In this section, we present the fundamentals of constraint satisfaction problems. We first introduce ordinary constraint networks, and then present the concept of hierarchical constraint networks which are able to handle conflicting set of constraints.

4.1 Constraint Networks

Basically, a constraint network consists of a set of variables, each with a specific domain, and a set of constraints over these variables. The task of a constraint solver is to find a variable assignment which satisfies all constraints.

Definition 6 (Constraint network). *A* constraint network *is a tuple* (X, D, C) *where*

- $X = \{x_1, \ldots, x_n\}$ *is a set of variables,*
- $D = \{d_1, \ldots, d_n\}$ *specifies the domain of each variable, and*
- $C = \{c_1, \ldots, c_m\}$ *is a set of constraints.*

Definition 7 (Solution of a constraint network). *A* variable assignment θ *for a constraint network* (X, D, C) *is a set*

$$\{(x_1, v_1), \ldots, (x_n, v_n)\},$$

with $\{x_1, \ldots, x_n\}$ *are the variables from* X*, and each* $v_i \in d_i$ *is a value from the respective domain.*

We write $c(\theta) = true$ *to indicate that a variable assignment* θ *satisfies a constraint* $c \in C$*, and* $c(\theta) = false$ *otherwise. A* solution *of a constraint network* (X, D, C) *is a variable assignment which satisfies all constraints from* C*, i. e.,* $c(\theta) = true$ *for all* $c \in C$*.*

A constraint network may have a unique solution, multiple solutions, or even no solutions, in case the constraints are inconsistent and thus conflicting.

4.2 Constraint Hierarchies

Given a conflicting set of constraints, an ordinary constraint network does not have a solution, because there is no variable assignment which satisfies all constraints at the same time. One could however be interested to find an approximate solution, i. e., a variable assignment which satisfies only some of the constraints. One approach to this problem are *hierarchical constraint networks* [12], which we introduce in this section.

Definition 8 (Hierarchical constraint network). *A* hierarchical constraint network *is a tuple* (X, D, C, H)*, where* (X, D, C) *is a constraint network, and* $H = \{H_0, \ldots, H_l\}$ *is a* constraint hierarchy*. The latter defines a partition of the constraint set* C*, assigning a* strength i *with* $0 \leq i \leq l$ *to each constraint* $c \in C$*. In particular, we have*

$$H_i \cap H_j = \emptyset \text{ for all } i \neq j$$

and

$$\bigcup_{H_i \in H} H_i = C.$$

The constraints $c \in H_0$ *are called the* required constraints*.*

The aim is to find a variable assignment which satisfies all of the constraints from H_0, and satisfies the other constraints from $H_1, \ldots, H_l$ "as good as possible". There might be various notions of what a "better" solution is; for the moment, we assume that we have a predicate

$$better(\sigma, \theta, H)$$

saying that σ is a better variable assignment than θ wrt. the constraint hierarchy H. *better* must be irreflexive and transitive. Based on this predicate, we can formally define a solution of a hierarchical constraint network.

Definition 9 (Solution of a hierarchical constraint network). *Let* (X, D, C, H) *be a hierarchical constraint network. Let*

$$S_0 \;:=\; \{\; \theta \mid c(\theta) = \mathit{true} \text{ for all } c \in H_0 \;\}$$

be the set of variable assignments which satisfy all required constraints. A solution to the hierarchical constraint network (X, D, C, H) *is a variable assignment* θ *such that*

$$\theta \in S_0 \text{ and } \neg \mathit{better}(\sigma, \theta, H) \text{ for all } \sigma \in S_0.$$

A solution satisfies all required constraints from H_0. Regarding the other levels $1, \ldots, l$, we investigate different approaches to define a *better* predicate.

Locally better. A variable assignment σ is *locally better* than a variable assignment θ iff both assignments satisfy exactly the same set of constraints up to some level $k-1$, and for level k, σ satisfies some constraint $c \in H_k$ which θ does not satisfy, and σ also satisfies any constraint from H_k which θ satisfies. In other words, we only consider the lowest level on which σ and θ differ; any other level $k+1, \ldots, l$ does not have an influence on the decision. Given an *error function* e with

$$e(c, \theta) := \begin{cases} 0 & \text{if } c(\theta) = \mathit{true} \\ 1 & \text{if } c(\theta) = \mathit{false}, \end{cases}$$

we can formally define the locally-better predicate as follows:

$$\begin{aligned} \mathit{better}_{\mathit{locally}}(\sigma, \theta, H) \;:=\; & \exists k > 0 \text{ such that} \\ & \forall i \in \{1, \ldots, k-1\}, \forall c \in H_i : e(c, \sigma) = e(c, \theta) \text{ and} \\ & \exists c \in H_k : e(c, \sigma) < e(c, \theta) \text{ and} \\ & \forall d \in H_k : e(d, \sigma) \leq e(d, \theta) \end{aligned}$$

Globally better. The globally-better predicate is parameterized by a function $g(\theta, H_i)$ which calculates how good a variable assignment θ satisfies the constraints on level i. A variable assignment σ is *globally better* than a variable assignment θ if both have the same quality (according to g) up to level $k-1$, and σ has a better quality than θ on level k:

$$\begin{aligned} \mathit{better}_{\mathit{globally}}(\sigma, \theta, H) \;:=\; & \exists k > 0 \text{ such that} \\ & \forall i \in \{1, \ldots, k-1\} : g(\sigma, H_i) = g(\theta, H_i) \text{ and} \\ & g(\sigma, H_k) < g(\theta, H_k) \end{aligned} \tag{3}$$

A suitable g function could e. g. count the number of constraints satisfied on a given level. (This is different from $\mathit{better}_{\mathit{locally}}$, where the *exact set* of constraints needs to be satisfied in order to have the same quality on some level $i \leq k-1$.) One could also assign weights to the constraints and calculated the weighted sum of the satisfied constraints. Further options are pointed out in [12].

5 A Static Approach Using a Constraint Network

We present a static inference control method *csp* using a hierarchical constraint network. First, we show how to construct an alternative database instance db_{alt}, which does not contain any confidential information anymore. Based on this alternative database instance, we can easily construct an enforcement method which satisfies the requirements of Definition 4.

5.1 Construction of db_{alt}

Given a database instance db, a set of a priori assumptions $prior$, and a confidentiality policy $policy$, we construct a hierarchical constraint network $CN(db, prior, policy) = (X, D, C, H)$ as follows:

Variables X: The set $X = \{x_1, \ldots, x_n\}$ of variables corresponds to the set of atomic sentences in the corresponding database schema DS. For example, when using propositional logic, X corresponds to the set of propositions.

Domains D: Each variable x_i has the domain $d_i = \{true, false\}$.

Constraints C: The set C of constraints consists of three subsets C_{ps}, C_{prior} and C_{db}:

1. The potential secrets must not hold in the alternative database instance:

$$C_{ps} := \bigcup_{\Psi \in policy} \{\neg\Psi\} \tag{4}$$

2. The a priori assumptions must hold in the alternative database instance:

$$C_{prior} := \bigcup_{\alpha \in prior} \{\alpha\}$$

3. All atoms should have the same value as in the original database instance:

$$C_{db} := \bigcup_{x \in X} \{eval^*(x)(db)\}$$

Note that these constraints may be conflicting – in particular, C_{ps} and C_{db} will be inconsistent in case at least one potential secret is *true* in the original instance.

Hierarchy H: We establish the following constraint hierarchy $H = \{H_0, H_1\}$: The constraints from C_{ps} and C_{prior} are the required constraints:

$$H_0 := C_{ps} \cup C_{prior}$$

The constraints from C_{db} are assigned to level 1:

$$H_1 := C_{db}$$

A valid solution to this constraint network is a variable assignment to all atoms of DS, and thus a structure which can be regarded as an alternative database instance db_{alt}. The constraints ensure that none of the potential secrets is *true* in db_{alt}, and that any sentence from *prior* holds in db_{alt}. Finally, db_{alt} should have a minimum distance to the original database instance db, i. e., a minimum number of atoms should have a different truth value in db and db_{alt}. This is achieved by employing the $better_{globally}$ predicate (3) with the underlying function

$$g(\sigma, H_i) := |\{c \in H_i \mid c(\sigma) = false\}|$$

which counts the number of constraints from H_i that are not satisfied (i. e., the number of atoms with a different truth value).

Remark 10. Given a relatively large database schema DS, the number of atoms (and thus the number of variables in X) can become very large. As an optimization, we can restrict X to the *relevant* atoms, i. e., those atoms which appear in at least one sentence of either *policy* or *prior*. The truth value of these relevant atoms will be calculated by the constraint network; all other, non-relevant atoms will have the same truth value in db_{alt} as in the original instance db.

5.2 Enforcement Method Based on db_{alt}

Based on the alternative database instance, we construct an enforcement method *csp*. The algorithm involves a preprocessing step which is initiated prior to the first query. In that step, the alternative database instance is generated with the means of the constraint network described in the previous section. The precondition $precond_{csp}$ is satisfied if a valid alternative database instance was identified. Finally, the evaluation of the query sequence is performed within the alternative database instance db_{alt}, using the $eval^*$ function (ordinary query evaluation).

Definition 11 (Enforcement method *csp*). *Let Q be a query sequence, db a database instance, prior a set of a priori assumptions, and policy a confidentiality policy. We define an enforcement method csp along with its precondition function $precond_{csp}$ as follows.*

1. Preprocessing step

If $db \models \Psi$ for some $\Psi \in policy$, or $db \not\models \alpha$ for some $\alpha \in prior$, construct an alternative database instance db_{alt} as specified in Section 5.1.

Otherwise, choose $db_{alt} := db$.

The precondition $precond_{csp}(db, prior, policy)$ is satisfied iff a valid alternative database instance db_{alt} could be identified. (Note that prior and policy might be inconsistent, so that the constraint network will not have a solution.)

2. Answer generation

All queries are evaluated in the alternative database instance db_{alt} using the ordinary query evaluation function:

$$ans_i := eval^*(\Phi_i)(db_{alt}) \quad for\ 1 \le i \le n \tag{5}$$

Theorem 12. *csp preserves confidentiality in the sense of Definition 4.*

Proof. Let Q be a query sequence, db a database instance, *prior* the a priori assumptions, and *policy* a confidentiality policy, such that $precond_{csp}(db, prior, policy)$ is satisfied. Let $\Psi \in policy$ be a potential secret.

In the preprocessing step, db_{alt} is either generated by the constraint network (in case at least one potential secret is *true* in the original db, or the a priori assumptions do not hold in db), or is identical to db. Query evaluation is performed within db_{alt} using the ordinary $eval^*$ function.

We show that db_{alt} can be regarded as a database instance db' as demanded by Definition 4, such that $precond_{csp}(db_{alt}, prior, policy)$ is *true*, and both conditions from that definition are satisfied.

Condition (b) [Ψ is *false* in db_{alt}]: If $db_{alt} = db$, then db does not imply any potential secret, in particular $db \not\models \Psi$. Otherwise, if db_{alt} was generated by the constraint network, the constraints in C_{ps} make sure that none of the potential secrets hold in db_{alt}.

Precondition: The preprocessing step for $(db_{alt}, prior, policy)$ will notice that none of the potential secrets holds in db_{alt} (see proof for condition (b) above), and that the a priori assumptions are satisfied in db_{alt} (due to the constraints C_{prior} used for the construction of db_{alt}). It will thus choose db_{alt} as the "alternative" database instance, and will not initiate the generation of a different instance by the constraint solver. Thus, db_{alt} itself will serve as the "alternative" database instance, and the precondition is satisfied for $(db_{alt}, prior, policy)$.

Condition (a) [Same answers]: We have shown above that the "alternative" database instance under $(db_{alt}, prior, policy)$ will then be db_{alt} itself. Consequently, the same answers will be returned as under $(db, prior, policy)$.

6 Comparison

In this section, we compare the static, constraint-based approach from Section 5 to the existing dynamic CQE approach (cf. Section 3).

Generally, the dynamic approach involves a certain overhead at query time:

1. We need to keep a log file of all past answers which consumes space. In particular, when multiple users (with the same confidentiality requirements) query the database at the same time, we need to keep a distinct log file for each user.
2. At each query, an implication problem must be solved, as we need to make sure that the resulting log file does not imply the disjunction of all potential secrets *pot_sec_disj*. However, logical implication can be computationally expensive or even undecidable in certain logics.

On the other hand, the static approach involves the expensive preprocessing phase in which the alternative database instance is generated; there is however no overhead at query time. We can also re-use the alternative database instance for multiple users and/or sessions, given that the users are subject to the same confidentiality policy, and are assumed to have the same a priori assumptions.

Table 1. Answers for the query sequence $Q_1 = \langle a, b\rangle$

	Dynamic Approach		Static Approach
Query Φ_i	Answer ans_i	Log File log_i	Answer ans_i
a	a	$\{a\}$	a
b	$\neg b$	$\{a, \neg b\}$	$\neg b$

Given these considerations, the static approach is more favorable if we expect long query sequences or multiple sessions by users with the same confidentiality requirements.

Although neither approach can anticipate future queries, the dynamic CQE approach can take advantage of that fact that it can dynamically choose when to return a lie, and only issue a distorted answer as a "last resort". This can lead to a gain of availability in certain situations. We demonstrate this by a minimal example in propositional logic.

Example 13. Consider the database schema $DS = \{a, b\}$ and the database instance $db = \{a, b\}$ (both a and b are *true* in db). We assume that the user does not make any a priori assumptions ($prior = \emptyset$), and he is not allowed to know that a and b are both *true*: $policy = \{a \wedge b\}$.

As $eval(a \wedge b)(db) = true$, the preprocessing step of the static approach will need to construct an alternative database instance, using the constraint network (X, D, C, H) with

$$X := \{a, b\},$$
$$D := \{\{true, false\}, \{true, false\}\},$$
$$C := C_{ps} \cup C_{db} \text{ with } C_{ps} = \{\neg(a \wedge b)\} \text{ and } C_{db} = \{a, b\},$$
$$H := \{H_0, H_1\} \text{ with } H_0 = C_{ps} \text{ and } H_1 = C_{db}.$$

This constraint network has two possible solutions:

$$\theta_1 = \{a \to true,\ b \to false\}$$
$$\theta_2 = \{a \to false,\ b \to true\}$$

Both satisfy all constraints from $H_0 = C_{ps}$ and a maximum number of constraints from $H_1 = C_{db}$. We cannot predict which solution will be chosen by the constraint solver. Assume that it will chose θ_1, then we have

$$db_{alt} = \{a, \neg b\}.$$

Consider the query sequence $Q_1 = \langle a, b\rangle$. The respective answers are given in Table 1. The dynamic approach will first return the original answer a, as it does not imply the disjunction of all potential secrets $pot\_sec\_disj = a \wedge b$. However, it returns a lie for the second query b. The static approach with the alternative database instance $db_{alt} = \{a, \neg b\}$ returns exactly the same answers.

Table 2. Answers for the query sequence $Q_2 = \langle b, a \rangle$

	Dynamic Approach		Static Approach
Query Φ_i	Answer ans_i	Log File log_i	Answer ans_i
b	b	$\{b\}$	$\neg b$
a	$\neg a$	$\{b, \neg a\}$	a

When we reverse the query sequence ($Q_2 = \langle b, a \rangle$, cf. Table 2), the static approach returns the same answers: First the lie $\neg b$, then the honest answer a. The dynamic approach however gives the honest answer b first, and then returns the lie $\neg a$. You can see that the dynamic approach uses the lies only as a last-minute action. Nevertheless, both approaches issue the same number of honest and dishonest answers, respectively.

Now imagine the user only issues a single query for b: $Q_3 = \langle b \rangle$. The dynamic approach returns the honest answer b, while the static approach returns the lie $\neg b$. This lie is not necessary to protect the potential secret $a \wedge b$. However, the static approach cannot know that the user will never ask for a, and cannot risk to omit the lie. In this situation, the dynamic approach offers a higher availability.

7 Extensions

The static inference control method presented in Section 5 resembles the dynamic uniform lying method of Controlled Query Evaluation, as summarized in Section 3, as well as the SAT-solver based approach from [5]. Hierarchical constraint networks however offer further possibilities, some of which we point out in this section, as a guideline for future work.

7.1 Explicit Availability Policy

A forthcoming extension of the SAT-solver based approach from [5] offers the ability to specify an explicit availability policy, namely a set *avail* of sentences for which the system must always return the correct truth value, and which must not be used as a lie in order to protect one of the potential secrets. In the context of our static method, we demand that any sentence from *avail* must have exactly the same truth value in db_{alt} as in the original instance db:

$$\textit{for each } \alpha \in \textit{avail} : \textit{eval}(\alpha)(db_{alt}) = \textit{eval}(\alpha)(db)$$

We can achieve this property by introducing another set of constraints

$$C_{avail} := \bigcup_{\alpha \in avail} \{\textit{eval}^*(\alpha)(db)\}$$

which is merged into the set of required constraints H_0:

$$H_0 := C_{ps} \cup C_{prior} \cup C_{avail}.$$

The constraint solver will then ensure the desired property. In particular, it will fail to find a solution if the actual truth values of the sentences from *avail* in *db* contradict to the negation of the potential secrets. For example, $avail = \{aids\}$ and $policy = \{aids\}$ will be inconsistent in case $eval(aids)(db) = true$.

7.2 Multiple Levels of Potential Secrets

The basic approach from Section 5, as well as Definition 4, assumes that each potential secret has the same quality wrt. secrecy: The user may not infer *any* of the potential secrets.

Depending on the application, one could imagine to soften this requirement and establish a *hierarchy* of potential secrets: secrets that the user *must not* learn, secrets that the user *should not* learn, secrets that the user *should rather not* learn, etc. The confidentiality policy is split up into multiple subsets of potential secrets,

$$policy = policy_0 \cup policy_1 \cup policy_2 \cup \ldots \cup policy_l,$$

where $policy_0$ are the *strict potential secrets*, and the potential secrets from $policy_1, \ldots, policy_l$ are called *loose potential secrets*. Similar to (4), we construct a set C_{ps_i} of constraints for each $0 \leq i \leq l$:

$$C_{ps_i} := \bigcup_{\Psi \in policy_i} \{\neg\Psi\}$$

These constraints, together with C_{prior} and C_{db}, are organized in the constraint hierarchy $H = \{H_0, \ldots, H_{l+1}\}$ as follows: The constraints C_{ps_0} for the strict potential secrets remain in the set of required constraints:

$$H_0 := C_{ps_0} \cup C_{prior}$$

Each set of constraints C_{ps_i} corresponding to a set of loose potential secrets $policy_i$ $(1 \leq i \leq l)$ is assigned a level of its own:

$$H_i := C_{ps_i}$$

Finally, the constraints C_{db}, demanding a minimum distance to the original database, build the highest level:

$$H_{l+1} := C_{db}$$

It is important to choose a suitable *better* predicate (cf. Section 4) which reflects the desired relationship between the various levels of potential secrets. The $better_{globally}$ predicate (3) may be a good choice; however, it might be favorable to "trade" a non-protected potential secret on some level i against multiple protected potential secrets on a level $k > i$. This would not be possible with $better_{globally}$, and the administrator would have to choose a different predicate.

Alternatively, or in addition to multiple levels of potential secrets, it is also possible to assign a weight to each potential secret. In this case, a suitable g function underlying the $better_{globally}$ predicate (3) must be used which considers these weights. Some possible functions are given in [12].

7.3 Refusal

The present approach constructs an alternative database instance db_{alt} in which the truth values of certain atoms have been *changed* such that none of the potential secrets hold in db_{alt}. This corresponds to the concept of lying in dynamic inference control.

Alternatively, one could *erase* the truth values of particular atoms in order to protect the secret information. This can be easily achieved by allowing an additional value – say, *undef* – for each variable x in the constraint network. The resulting alternative database instance is *incomplete*: The value of certain sentences cannot be determined due to the missing truth values.

Of course, the user expects the answers to originate from a *complete* database. It is therefore not acceptable to disclose that certain information is missing in db_{alt}, and that a query Φ cannot be answered. A possible solution is to pick up the *refusal* approach from Controlled Query Evaluation [11]: In addition to Φ and $\neg\Phi$, we allow the special answer `mum` indicating a refused answer. Each time the alternative database instance db_{alt} cannot provide the answer to a query Φ, the system returns `mum` instead.

It is important to avoid harmful *meta inferences*: The user may not conclude from a refused answer that the secret information he might have asked for is actually true. For example, given $policy = \{a\}$ and $db = \{a, b\}$, the alternative database instance could be $db_{alt} = \{b\}$ (with the truth value of a removed). The query $\Phi = a$ will lead to a refusal. The user could then conclude that a must have been true in the original database instance.

To avoid such meta inferences, we must remove the truth value of the potential secret a even if it was *not* true in the original instance, for example in case $db' = \{\neg a, b\}$, which would then lead to the same alternative instance $db_{alt} = \{b\}$. Then, a refused answer does not provide any information about the original query value anymore.

8 Conclusion

We have presented a static approach for inference control in logic databases. The system is supported by a hierarchical constraint solver which generates an alternative database instance in which all confidential information has been replaced by harmless information. In general, this corresponds to the uniform lying approach of the (dynamic) Controlled Query Evaluation framework. In Section 6, we have shown that both approaches have advantages and drawbacks, and that maximum availability (measured by the number of distorted answers) can only be achieved by a dynamic approach, yet with a relatively high runtime complexity. This result justifies the employment of dynamic approaches when maximum availability is an issue.

The use of a constraint solver makes the construction of a suitable alternative database instance rather easy, as we can declaratively define the desired properties for that database instance (which generally correspond to the declarative properties demanded by Definition 4). While the basic static approach from

Section 5 only makes use of the fundamental abilities of hierarchical constraint networks, there are various options to exploit the remaining opportunities, some of which were presented in Section 7. These shall be further investigated in future work.

References

1. Farkas, C., Jajodia, S.: The inference problem: A survey. SIGKDD Explorations 4(2), 6–11 (2002)
2. Biskup, J., Bonatti, P.A.: Controlled query evaluation for enforcing confidentiality in complete information systems. International Journal of Information Security 3, 14–27 (2004)
3. Biskup, J., Bonatti, P.A.: Controlled query evaluation with open queries for a decidable relational submodel. In: Dix, J., Hegner, S.J. (eds.) FoIKS 2006. LNCS, vol. 3861, pp. 43–62. Springer, Heidelberg (2006)
4. Biskup, J., Weibert, T.: Keeping secrets in incomplete databases. Submitted, 2007. In: FCS 2005. Extended abstract presented at the LICS 2005 Affiliated Workshop on Foundations of Computer Security (2005), available from `http://www.cs.chalmers.se/~andrei/FCS05/fcs05.pdf`
5. Biskup, J., Wiese, L.: On finding an inference-proof complete database for controlled query evaluation. In: Damiani, E., Liu, P. (eds.) Data and Applications Security XX. LNCS, vol. 4127, pp. 30–43. Springer, Heidelberg (2006)
6. Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
7. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, Heidelberg (2003)
8. Ahn, G.J., Sandhu, R.: Role-based authorization constraints specification. ACM Trans. Inf. Syst. Secur. 3(4), 207–226 (2000)
9. Jaeger, T.: On the increasing importance of constraints. In: RBAC 1999. Proceedings of the fourth ACM workshop on Role-based access control, pp. 33–42. ACM Press, New York (1999)
10. Moodahi, I., Gudes, E., Lavee, O., Meisels, A.: A secureworkflow model based on distributed constrained role and task assignment for the internet. In: Lopez, J., Qing, S., Okamoto, E. (eds.) ICICS 2004. LNCS, vol. 3269, pp. 171–186. Springer, Heidelberg (2004)
11. Biskup, J., Bonatti, P.A.: Lying versus refusal for known potential secrets. Data & Knowledge Engineering 38, 199–222 (2001)
12. Borning, A., Freeman-Benson, B.N., Wilson, M.: Constraint hierarchies. Lisp and Symbolic Computation 5(3), 223–270 (1992)