Preprocessing for Controlled Query Evaluation with Availability Policy*

Joachim Biskup and Lena Wiese†
Universität Dortmund, 44221 Dortmund, Germany
Tel.: +49-231-755-4813, Fax: +49-231-755-2405
{biskup,wiese}@ls6.cs.uni-dortmund.de
http://ls6-www.cs.uni-dortmund.de/issi/

Abstract

Controlled Query Evaluation (CQE) defines a logical framework to protect confidential information in a database. By modeling a user's a priori knowledge appropriately, a CQE system not only controls access to certain database entries but also accounts for information inferred by the user. In this article, we present a static (preprocessing) CQE-approach for propositional databases with an availability policy. The resulting inference-proof and availability-preserving database ensures confidentiality of secret information while guaranteeing availability of certain database entries to a highest degree possible. We illustrate the semantics of the system by a comprehensive example and state the essential requirements for an inference-proof and availability-preserving database. We present an algorithm that accomplishes the preprocessing by combining SAT solving and "Branch and Bound".

Keywords: Controlled Query Evaluation, inference control, lying, availability policy, confidentiality policy, complete database systems, propositional logic, SAT solving, Branch and Bound

1 Introduction and Related Work

Controlled query evaluation (cf. [1–7]) aims at preserving the confidentiality of some secret information in a sequence of queries to a database. Not just plain access to certain database entries is denied (as traditionally based on an "access control policy") but instead a "confidentiality policy" is specified and information that could be gained by logical reasoning is taken into account. This is what

*This article is an extended version of [7].

†Corresponding author; partially funded by a Research Training Group of the German Research Council (DFG).

is usually called *inference control*. There are several different approaches addressing inference control for example for statistical databases [11], distributed databases [8], relational databases with fuzzy values [16] and for XML documents [22]. In [13] the authors give a comprehensive overview of existing inference control techniques and state some of the fundamental problems in this area. Wang et al. [21] name two typical distortion mechanisms (in their case for online analytical processing (OLAP) systems): *restriction* (deleting some values in the query result) and *perturbation* (changing some values in the query result). In general, any method for avoiding inferences has an effect on the accuracy of the returned answers: there is a trade-off between *confidentiality of secret information* and *availability of correct information*; in order to protect the secret information, some (even non-secret) information must possibly be distorted.
The above mentioned approaches are typically based on specialized data structures (relational data model, XML documents); Controlled Query Evaluation (CQE) however offers a flexible framework to execute inference control based on an arbitrary logic satisfying some natural properties. In this paper, we restrict ourselves to CQE with propositional logic; we construct an inference-proof database considering the original (insecure) database, a user's a priori knowledge, a set of secrets, and additionally a set of propositional sentences that should at best *not* be distorted in the resulting inference-proof database (while still retaining confidentiality of the secrets). In Section 2 we introduce the CQE framework and state the prerequisites assumed in this paper. In Section 3 we formalize the notion of an inference-proof and availability-preserving database. Section 4 shows a transformation of our problem to SAT solving and "Branch-and-Bound" and presents an algorithm that computes an inference-proof and availability-preserving database. Section 5 concludes the article.

2 Controlled Query Evaluation

Basically, a system model for Controlled Query Evaluation consists of:

1. a database that contains some freely accessible information and some secret information

2. a single user (or a group of collaborating users, respectively) having a certain amount of information as a priori knowledge of the database and the world in general; the case that several different users independently query the database is not considered as the database cannot distinguish whether a group of users collaborates or not

The user sends queries to the database and the database returns corresponding answers to the user.

To prevent the user from inferring confidential information from the answers and his assumed a priori knowledge, appropriate restriction or perturbation is enforced by the CQE system on the database side. In CQE on the one hand *refusal* is used as a means of restriction: to a critical query the database refuses

to answer (i.e., just returns `mum`). On the other hand, *lying* is employed as a means of perturbation: the database returns a false value or declares the query answer as undefined although a value exists in the database. In this way, the CQE approach automates the enforcement of confidentiality: wanting to restrict access to some secret information, a database administrator just declares the secrets in the *confidentiality policy*; based on this, the CQE system computes the possibly distorted (hence inference-proof) answers. However, the database should be as cooperative as possible: availability of correct information should be maximized and thus only a minimum of answers should be distorted, while still ensuring confidentiality of secrets.

The notion of availability can even be taken further: In certain cases, availability of some database entries may be more important than availability of other entries. In such cases, the database administrator can additionally declare an *availability policy*. The CQE system then tries to return correct answers for the entries in the availability policy and favors distortion of entries not included in this policy – but still confidentiality takes precedence over availability.

CQE can be varied based on several different parameters (see [1–7]). In this paper we focus on a *complete* information system in *propositional logic* with a *known* confidentiality policy of *potential secrets* and an additional *availability policy*; we use *lying* as the only distortion mechanism. Thus, in this paper a CQE system is based on the following:

- a (possibly infinite) alphabet $\mathcal{P}$ of propositional variables; formulas can be built from the variables with the connectives $\neg$, $\vee$ and $\wedge$;[1] formulas contain "positive literals" (variables) and "negative literals" (negations of variables)

- a database instance as a finite set $db \subset \mathcal{P}$ that represents an interpretation I of the propositional variables: for each $A \in \mathcal{P}$, if $A \in db$, then I assigns A the value *true* (written as $I(A) = 1$), else I assigns A the value *false* (written as $I(A) = 0$); this means that we consider a complete database (to each query the database returns either *true* or *false*) and only instances with a finite positive part

- the evaluation function $eval^*(\Phi)(db)$ that returns the formula Φ (if it is evaluated to *true* according to *db*) or its negation (if it is evaluated to *false*):

$$eval^*(\Phi)(db) = \begin{cases} \Phi & \text{if } I \models \Phi \quad \text{(with } \models \text{ being the model operator)} \\ \neg\Phi & \text{else} \end{cases}$$

- a confidentiality policy *pot_sec* as a finite set of formulas over $\mathcal{P}$ of "potential secrets"; the semantics is that for each formula $\Psi \in pot\_sec$, if Ψ evaluates to *true* according to *db* then the user may not know this, but the user may believe that the formula evaluates to *false* (that is, for a complete *db* the negation of Ψ evaluates to *true* according to *db*); furthermore,

[1] Two consecutive negations cancel each other out: $\neg\neg A \equiv A$

the user "knows" the confidentiality policy: he knows the specification of the secrets (but does not know a priori the truth values of the secrets according to the current database instance *db*)

- an availability policy *avail* as a finite set of formulas over $\mathcal{P}$ specifying important facts whose truth value (according to *db*) should preferably not be distorted; that is, if values have to be distorted to protect a secret, for a formula $\Theta \in avail$ the distortion should at best not change the value $eval^*(\Theta)(db)$ in the answer to the user
- the user's a priori knowledge as a finite set of formulas over $\mathcal{P}$ called *prior*; *prior* may contain general knowledge (like implications over $\mathcal{P}$) or knowledge of *db* (like semantic constraints)

There are some restrictions on the user's knowledge. In this paper, we presume:

(a) [**consistent knowledge**] *prior* is consistent and the user cannot be made believe inconsistent information at any time

(b) [**monotone knowledge**] the user cannot be "brainwashed" and forced to forget some part of his knowledge

(c) [**single source of information**] the database *db* is the user's only source of information (besides *prior*)

(d) [**unknown secrets**] the a priori knowledge does not imply a secret; that is, for all $\Psi \in pot\_sec$: $prior \not\models \Psi$ (with $\models$ being the implication operator)

(e) [**implicit closure under disjunction**] the user may not know (a priori) that the disjunction of the potential secrets is true:

$$prior \not\models pot\_sec\_disj \quad (\text{where } pot\_sec\_disj := \bigvee_{\Psi \in\, pot\_sec} \Psi) \qquad (1)$$

The first three requirements (a) – (c) originate from our system settings: On the one hand, we use "classical" logic (where contradictions imply any proposition – including the secrets – and thus have to be avoided). On the other hand, when trying to model a real-world user, we have to admit that we can merely produce an approximation of human knowledge, and influencing a user's knowledge by technical means is impossible anyway; that is why we assume a closed system where the user's knowledge cannot be changed from outside the system and just be incremented from inside the system.
We require (d) because if the user already knows a potential secret, we obviously cannot protect the secret anymore. Requirement (e) is an even stricter condition and is owed to the fact that lying is our only distortion mechanism: without it, the system could run into a situation where even a lie reveals a secret. To illustrate this, assume $pot\_sec = \{\alpha, \beta\}$ (for formulas α and β that are both *true* according to *db*) and $prior = \{\alpha \vee \beta\}$; to the query $\Phi = \alpha$ the CQE

system would return the lie $\neg\alpha$, but this would enable the user to conclude that β was true (and he is not allowed to know this); thus, we require *prior* to fulfill Equation (1).
This line of reasoning also demands that the CQE system lie to every query entailing the disjunction of some potential secrets (see [1, 3] for more information). This obviously is a disadvantage of the lying approach that restrains its applicability: whenever an exhaustive enumeration of alternatives is known by the user although each individual alternative is specified secret there is no option left for lying. That is, in the lying approach, not all alternatives can be specified secret: there has to be one non-secret alternative that permits a lie. As possible remedies we propose to either use refusal as a second distortion mechanism (see [3]) or allow the database to be incomplete such that a (non-secret) undefined value could be returned as a lie (see [6]). Both options are outside the system settings assumed in this paper and merely stated here without dwelling on technical details.

2.1 An Example System

The following example shall clarify the system design. We have a database with Alice's medical records. The curious user Mallory wants to find out whether she is seriously ill. We use the alphabet:

$$\mathcal{P} = \{\texttt{cancer}, \texttt{aids}, \texttt{flu}, \texttt{medA}, \texttt{medB}\}$$

Poor Alice is badly ill and her medical records (that is, the current database instance *db*) look like this:

$$db = \{\texttt{cancer}, \texttt{aids}, \texttt{medA}, \texttt{medB}\}$$

Mallory has certain background knowledge about the medication. He knows that:

1. if a patient takes medicine A, (s)he suffers from aids or cancer
2. if a patient takes medicine B, (s)he suffers from cancer or flu

Expressing these implications as formulas, we have the a priori knowledge:

$$prior = \{\neg\texttt{medA} \vee \texttt{aids} \vee \texttt{cancer}, \neg\texttt{medB} \vee \texttt{cancer} \vee \texttt{flu}\}$$

Apart from maintaining the database, the database administrator specifies the potential secrets; in our example, Mallory should *not* be able to infer the diseases cancer and aids:

$$pot\_sec = \{\texttt{aids}, \texttt{cancer}\}$$

Obviously, queries concerning potential secrets (for example, the two queries "`cancer`" and "`aids`") should prompt the CQE system to return lies (in this case, "$\neg$`cancer`" and "$\neg$`aids`"). Moreover, if the answer to a query would enable the user to infer a secret, the CQE system should return a lie, too (consider for

example the query "$\texttt{medB} \wedge \neg \texttt{flu}$" whose correct answer would imply the secret "cancer"). As can be seen from these considerations, confidentiality of secret information is considered more important than a correct and reliable answer. Secret information has to be kept secret even at the risk of returning inaccurate information.
Some database entries may be of more importance than others. For example, some medicine might have serious side effects or mutual reactions with other substances; that is why information regarding medication should at best not be distorted. In our example, the database administrator declares the availability policy:

$$avail = \{\texttt{medA}, \texttt{medB}\}$$

This example will be continued in Sections 4.1 and 4.2.

3 Constructing an Inference-Proof Database

Given a database db, a confidentiality policy pot_sec, an availability policy $avail$ and the user's a priori knowledge $prior$ as described in the previous section, we now want to construct a database db' (representing a new interpretation I') that is inference-proof and availability-preserving with respect to every possible sequence of queries the user may come up with. We demand the following properties for db' to be fulfilled:

i. [**complete**] db' is a complete database with a finite positive part

ii. [**consistent**] db' is consistent in itself and consistent with $prior$

iii. [**inference-proof**] I' does not satisfy any of the potential secrets; that is, for every $\Psi \in \ pot\_sec$: $I' \not\models \Psi$

iv. [**availability-preserving**] db' evaluates as many of the entries in $avail$ as possible as db does; only a minimum of entries changes its truth value:

$$avail\_dist := ||\{\Theta \mid \Theta \in avail,\ \text{eval}^*(\Theta)(db') \neq\ \text{eval}^*(\Theta)(db)\}|| \longrightarrow \min$$

v. [**distortion-minimal**] db' contains as few lies as possible (with respect to the original database db); that is, considering the set of propositional variables $\mathcal{P}$, the difference between db and db' is minimal:

$$db\_dist := ||\{A \mid A \in \mathcal{P},\ \text{eval}^*(A)(db') \neq\ \text{eval}^*(A)(db)\}|| \longrightarrow \min$$

As for the completeness property (i.), we want db' to represent an interpretation I' that assigns a value to every propositional variable in $\mathcal{P}$, but the value *true* only to a finite subset of $\mathcal{P}$.

The consistency property (ii.) means that we want to find an interpretation I' such that all formulas in $prior$ are satisfied because the user's a priori knowledge is fixed and we cannot make him believe inconsistent information; that is particularly, for every $\chi \in prior$: $\text{eval}^*(\chi)(db') = \chi$.

As for the inference-proofness property (iii.), in the special case treated in this paper – *known policy* and *lying* – the user knows that the system lies when queried after a potential secret: for every $\Psi \in$ *pot_sec*: eval*$(\Psi)(db') = \neg\Psi$. That is why we define the set of formulas:

$$Neg(pot\_sec) := \{\neg\Psi \mid \Psi \in pot\_sec\}$$

and try to find an interpretation I' that satisfies all formulas in $Neg(pot\_sec)$ in order for db' to be inference-proof.

With the availability preservation property (iv.), from all interpretations that ensure confidentiality of the secrets, we choose one that maximizes the availability of important database entries.
We give availability preservation (iv.) priority over distortion minimality (v.); however, if there is no unique solution with minimal availability distance, we consider distortion minimality as a basic background availability property: from all inference-proof interpretations that preserve availability equally well, we choose one that minimizes the amount of lies in db'.

3.1 Existence and Finiteness of Solution Database

All in all we conclude that I' has to be an interpretation (for the variables in $\mathcal{P}$) that first of all satisfies the set of formulas $prior \cup Neg(pot\_sec)$, second, retains the truth value of a maximum of formulas in *avail* and in the third place contains only a minimum of lies with respect to the original interpretation I.
Under the requirements (a) and (e) given in Section 2, such an inference-proof interpretation always exists. To prove this, first of all note that requirement (e) implies that *pot_sec_disj* is not a tautology. Combining requirements (a) and (e), we conclude that *prior* is consistent with the set $Neg(pot\_sec)$. Thus, there exists at least one interpretation I' satisfying $prior \cup Neg(pot\_sec)$.
The solution database db' contains all variables A having the truth value *true* (that is, $I'(A) = 1$). The finiteness of its positive part is ensured by a restriction to a finite set $\mathcal{P}_{decision}$ of "decision variables" and just computing a new interpretation $I'_{decision}$ for these variables. The decision variables are all variables occurring in *prior*, $Neg(pot\_sec)$ and *avail*.[2] For the set of variables $Vars(\cdot)$ occurring in a set of formulas, we have:

$$\mathcal{P}_{decision} := Vars(prior) \cup Vars(Neg(pot\_sec)) \cup Vars(avail)$$

All other (non-decision) variables get assigned the same truth value as before: $I'(A) := I(A)$ if $A \in \mathcal{P} \setminus \mathcal{P}_{decision}$. This restriction to a finite set of decision variables is indeed possible because changing truth values of non-decision variables has no effect on attaining consistency with the negations of the secrets. It is also the best we can achieve for distortion minimality as the distance restricted to the non-decision variables is $db\_dist_{|\mathcal{P}\setminus\mathcal{P}_{decision}} = 0$.
The minimization criteria are met with a "Branch and Bound" approach.

[2] Actually, we could leave out variables from formulas in *avail* that are not affected by variables in *prior* or $Neg(pot\_sec)$; for sake of simplicity we do not consider this case here.

4 A "Branch and Bound"-SAT-solver

In order to find interpretation I', we combine SAT-solving (for the completeness and satisfiability requirements) with "Branch and Bound" (for the minimization requirements). The database db' representing I' will be inference-proof and availability-preserving by construction, as we describe in the following.

SAT solvers try to find a satisfying interpretation for a set of clauses (i.e. disjunctions of literals). The basis for nearly all non-probabilistic SAT solvers is the so-called DPLL-algorithm (see [9, 10]). It builds an interpretation step-by-step by assigning variables a truth value with the methods:[3]

1. "elimination of one-literal clauses" (also called "boolean constraint propagation", **BCP**): a unit clause (i.e., a clause consisting of just one literal) must be evaluated to *true*

2. **splitting** on variables: take one yet uninterpreted variable, set it to *false* (to get one subproblem) and to *true* (to get a second subproblem), and try to find a solution for at least one of the subproblems

Whenever a variable is assigned a value, the set of clauses can be **simplified** by unit **subsumption** (if a clause contains a literal that is evaluated to *true*, remove the whole clause) or unit **resolution** (if a clause contains a literal that is evaluated to *false*, remove this literal from the clause but keep the remaining clause). If there is only the empty set left (which is equivalent to *true*), the current interpretation is satisfying; however, if the clause set eventually contains the empty clause $\square$ (which is equivalent to *false*), the interpretation is not satisfying.

"Branch and Bound" (B&B, for short) is a method for finding solutions to an optimization problem. It offers the features "branching" (dividing the problem into adequate subproblems), "bounding" (efficiently computing local lower and upper bounds for subproblems), and "pruning" (discarding a subproblem due to a bad bound value). For a minimization problem a global upper bound is maintained stating the currently best value. A B&B-algorithm may have a super-polynomial running time; however, execution may be stopped with the assurance that the optimal solution's value is in between the global upper bound and the minimum of the local lower bounds.

4.1 The Algorithm

We now describe the algorithm that computes an inference-proof database db' from *db*, *prior*, *Neg*(*pot_sec*) and *avail* by using SAT-solving and B&B. Listings 1 – 5 show the necessary functions in pseudocode.

In this section, we assume that the input sets *prior*, *Neg*(*pot_sec*) and *avail* are sets of formulas in conjunctive normal form (CNF). An extension to arbitrary formulas is given in Section 4.3. Thus, for now each formula can be represented

[3] We leave out the "affirmative-negative rule" for "pure literals" as it could contradict the minimization requirements.

by a set of clauses (each conjunct can be written as a clause $c = [l_1, \ldots, l_n]$ for the literals l_i in the formula). The clause representation of *prior* is:

$$C^{prior} = \{\{c_1^\chi, \ldots, c_m^\chi\} |\ \chi \in prior, c_j^\chi \text{ is the } j\text{th conjunct of } \chi\}$$

Analogously, $C^{Neg(pot\_sec)}$ is the clause representation of $Neg(pot\_sec)$. In our example, we have:

$$C^{prior} = \{\{[\neg\texttt{medA}, \texttt{aids}, \texttt{cancer}]\}, \{[\neg\texttt{medB}, \texttt{cancer}, \texttt{flu}]\}\}$$

$$C^{Neg(pot\_sec)} = \{\{[\neg\texttt{aids}]\}, \{[\neg\texttt{cancer}]\}\}$$

As we build the new interpretation I' step-by-step, each formula (that is, each clause set) is eventually simplified by subsumption and resolution.

For the availability distance *avail_dist* (as defined in the previous section), we have to count the differences between the original and the new interpretation. To be able to do this while still simplifying the clause sets, we add an "expected value" flag to each clause set; it is written as $\emptyset$ if the clause set is evaluated to *true* according to *db*, else it is written as $\{\square\}$. Thus the clause representation of the availability policy looks like this:

$$C^{avail} = \{\{c_1^\Theta, \ldots, c_m^\Theta\}(\mathit{flag}^\Theta) |\ \Theta \in avail, c_j^\Theta \text{ is the } j\text{th conjunct of } \Theta\}$$

where

$$\mathit{flag}^\Theta = \begin{cases} \emptyset & \text{if } eval^*(\Theta)(db) = \Theta \\ \{\square\} & \text{else} \end{cases}$$

If we now treat this data structure as a multiset (i.e., allowing duplicates in it), the availability distance of a database db' can easily be calculated by counting the clause sets whose evaluation according to db' does not coincide with their expected values (so-called "contradictory entries"). That is, the flag values are evaluated according to *db* once at the beginning; with them, the availability distance (and upper and lower bounds for it) can efficiently be computed at runtime without the need to re-evaluate all *avail* formulas on *db*.
In our example, we have:

$$C^{avail} = \{\{[\texttt{medA}]\}(\emptyset), \{[\texttt{medB}]\}(\emptyset)\}$$

If eventually `medA` is assigned *false* and `medB` is assigned *true*, the clauses are resolved and subsumed such that C^{avail} becomes $\{\{\square\}(\emptyset), \emptyset(\emptyset)\}$ with distance $avail\_dist = 1$ due to one contradictory entry.

We apply boolean constraint propagation and splitting of the DPLL-algorithm to find the interpretation $I'_{decision}$ for the set $\mathcal{P}_{decision}$ of decision variables having the properties stated in Section 3. B&B on the set $\mathcal{P}_{decision}$ yields a binary tree; its maximal depth is the cardinality of $\mathcal{P}_{decision}$. That is, in the worst case, the search space has size exponential in the number of decision variables. However, the aim of the B&B algorithm is to prune branches in the search tree as soon as possible and thus reduce the size of the search space.

Each node v in the search tree represents:

- a (partial) interpretation I_v of all the decision variables that already have been assigned a truth value by either BCP or splitting so far; the rest of the variables is undefined
- three local sets of clause sets C_v^{prior}, $C_v^{Neg(pot\_sec)}$ and C_v^{avail} that are generated from C^{prior}, $C^{Neg(pot\_sec)}$ and C^{avail} by simplification wrt. I_v
- a lower bound for the availability distance *avail_dist* called $min\_unavail_v$, defined as the number of clause sets in C_v^{avail} that do not coincide with their expected value (the "contradictory entries")
- an upper bound for the availability distance *avail_dist* called $max\_unavail_v$, defined as $min\_unavail_v$ plus the number of clause sets in C_v^{avail} that still contain undefined variables ("contradictory and undefined entries")
- a lower bound for the distortion distance *db_dist* called min_lies_v, defined as the number of variables with different value in I_v: $||\{A|I(A) \neq I_v(A)\}||$
- an upper bound for the distortion distance *db_dist* called max_lies_v, defined as the number of variables with different or undefined value in I_v: $||\{A \mid I(A) \neq I_v(A)$ or A is undefined$\}||$

We also have a current global optimum I_{best} (with the bounds $max\_unavail_{best}$ etc.) that stores the best *complete* interpretation found so far; it is however initialized with the partial interpretation of the root node (see Listing 1).
Availability preservation takes priority over distortion minimality. That is, we have a lexicographic ordering of the two distance measures: for two *complete* interpretations I_v and $I_{v'}$, I_v is better than $I_{v'}$ if $max\_unavail_v < max\_unavail_{v'}$ or if $max\_unavail_v = max\_unavail_{v'}$ and $max\_lies_v < max\_lies_{v'}$. Note that for complete interpretations, lower and upper bounds coincide and form the distance measures defined previously.

Now in each node v in our search tree, we can easily check the following pruning conditions (see Listing 5):

1. [**unsatisfiability**] if the simplified sets C_v^{prior} or $C_v^{Neg(pot\_sec)}$ contain an empty clause, the (possibly partial) interpretation I_v can never be expanded to a complete interpretation satisfying $prior \cup Neg(pot\_sec)$
2. [**bad availability distance**] if the local lower bound is worse than the current global upper bound, that is, $min\_unavail_v > max\_unavail_{best}$, we already found a solution that preserves availability better than I_v
3. [**bad distortion distance**] in case that I_v preserves availability equally well as the currently best solution, i.e. $min\_unavail_v = max\_unavail_{best}$ we check the distortion distance: if $min\_lies_v \geq max\_lies_{best}$, we already found a solution that contains less lies than (or as many lies as) I_v

If one of the pruning conditions is met, we can skip exploration of the current branch and backtrack to the next alternative branch to continue the search.

The search is started in the root node r with an "empty" interpretation I_r where all propositional variables in $\mathcal{P}_{decision}$ are undefined (see Listing 1). This initial interpretation is then gradually expanded in each node v by:

- choosing a unit clause from a clause set in C_v^{prior} or $C_v^{Neg(pot\_sec)}$ and propagating its value in C_v^{prior}, $C_v^{Neg(pot\_sec)}$ and C_v^{avail} with BCP (see Listing 2); note that in general we cannot choose unit clauses from C_v^{avail} as we want to minimize the number of contradictory entries in this set and satisfying those unit clauses could yield a suboptimal solution
- choosing a variable from $\mathcal{P}_{decision}$ that is undefined in I_v (the "splitting variable") and constructing a left and a right child node of v: in one child node we assign the variable the same truth value as in the original interpretation I and in the other child node we assign the variable the opposite truth value (see Listing 3); one child node is processed before the other one (e.g., according to a "best-first-search" heuristic based on a look-ahead strategy or by preferring the value of the original interpretation)

After an assignment, the clause sets are simplified (see Listing 4). Due to subsumption, variables (so called "don't care variables") might "disappear" from the clauses: they are still undefined but there are no clauses left that contain them. However, these variables had no influence on the satisfiability of the clauses; due to distortion-minimality, they are assigned the same truth value as in the original interpretation.

Having found $I'_{decision}$ satisfying C^{prior} as well as $C_v^{Neg(pot\_sec)}$ and having minimal distances to $avail$ and db, we can construct our inference-proof database db' as follows:

$$\begin{aligned} \text{for all } A \in \mathcal{P}_{decision}: &\quad A \in db' \text{ iff } I'_{decision}(A) = 1 \\ \text{for all } A \in \mathcal{P} \setminus \mathcal{P}_{decision}: &\quad A \in db' \text{ iff } A \in db \text{ (that is } I(A) = 1) \end{aligned}$$

1. **Initialization** for the root node r
   1.1. $I_r(A) :=$ *undefined* for all $A \in \mathcal{P}_{decision}$;
   1.2. $min\_unavail_r := 0$; $max\_unavail_r := ||avail||$;
   1.3. $min\_lies_r := 0$; $max\_lies_r := ||\mathcal{P}_{decision}||$;
   1.4. $C_r^{prior} := C^{prior}$; $C_r^{Neg(pot\_sec)} := C^{Neg(pot\_sec)}$; $C_r^{avail} := C^{avail}$;
   1.5. $I_{best} := I_r$;
   1.6. `BCP(r)`;

Listing 1: Initialization for root node r

2. `BCP(`v`)`: **Boolean Constraint Propagation** in node v

   2.1. `while` (there is a unit clause $[l]$ in C_v^{prior} or $C_v^{Neg(pot\_sec)}$)

      2.1.1. `if` $(l = A)$ $\{I_v(A) := 1;\}$

      2.1.2. `if` $(l = \neg A)$ $\{I_v(A) := 0;\}$

      2.1.3. `SIMP(`v`,`A`);`

   2.2. `if` (there is an $A' \in \mathcal{P}_{decision}$ with $I_v(A') =$*undefined*) {`SPLIT(`v`);`}

Listing 2: Boolean Constraint Propagation

3. `SPLIT(`v`)`: **Splitting** on a decision variable in node v

   3.1. generate two child nodes v_{left} and v_{right} of node v and copy partial interpretation and bound values of v into them

   3.2. choose $A \in \mathcal{P}_{decision}$ with $I_v(A) =$*undefined*

   3.3. $I_{v_{left}}(A) := I(A)$;

   3.4. `SIMP(`v_{left}`,`A`);`

   3.5. `BCP(`v_{left}`);`

   3.6. $I_{v_{right}}(A) := \overline{I(A)}$;

   3.7. `SIMP(`v_{right}`,`A`);`

   3.8. `BCP(`v_{right}`);`

Listing 3: Splitting on a decision variable

4. `SIMP(`v`,`A`)`: **Simplification** of clause sets given $I_v(A)$

   4.1. `foreach` clause c of a clause set in C_v^{prior}, $C_v^{Neg(pot\_sec)}$ and C_v^{avail}

      4.1.1. `if` $(I_v(A) = 1$ and $A \in c$ or $I_v(A) = 0$ and $\neg A \in c)$

         4.1.1.1. **Subsumption:** remove c from the clause set

      4.1.2. `if` $(I_v(A) = 0$ and $A \in c$ or $I_v(A) = 1$ and $\neg A \in c)$

         4.1.2.1. **Resolution:** remove $(\neg)A$ from c

   4.2. `CHECKPRUNE(`v`)`;

   4.3. `if` $(I_v(A') \neq$ *undefined* for all $A' \in \mathcal{P}_{decision})$

      4.3.1. `if` $(max\_unavail_v < max\_unavail_{best}$ or $max\_unavail_v = max\_unavail_{best}$ and $max\_lies_v < max\_lies_{best})$

         4.3.1.1. $I_{best} := I_v$;

Listing 4: Simplification of clauses

5. `CHECKPRUNE(`v`)`: **Check pruning conditions** in node v

   5.1. $min\_unavail_v :=$ number of contradictory entries in C_v^{avail};

   5.2. $max\_unavail_v :=$ number of contradictory and undefined entries;

   5.3. `if` $(I_v(A) = I(A))$ $\{max\_lies_v$`--;`$\}$

   5.4. `if` $(I_v(A) \neq I(A))$ $\{min\_lies_v$`++;`$\}$

   5.5. `if` $(\{\square\} \in C_v^{prior}$ or $\{\square\} \in C_v^{Neg(pot\_sec)}$ or $min\_unavail_v > max\_unavail_{best}$ or $min\_unavail_v = max\_unavail_{best}$ and $min\_lies_v \geq max\_lies_{best})$

      5.5.1. `PRUNE`;

Listing 5: Check pruning conditions

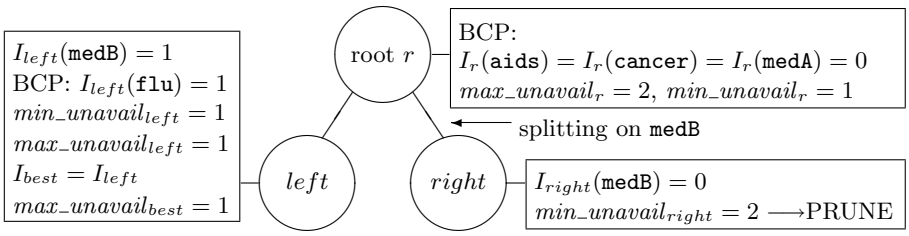

Figure 1: An example run

4.2 An Example Run

We now present the search tree for our example (see Figure 1). For the root node r we initially have:

$$C_r^{prior} = \{\{[\neg\texttt{medA}, \texttt{aids}, \texttt{cancer}]\}, \{[\neg\texttt{medB}, \texttt{cancer}, \texttt{flu}]\}\}$$
$$C_r^{Neg(pot\_sec)} = \{\{[\neg\texttt{aids}]\}, \{[\neg\texttt{cancer}]\}\}$$
$$C_r^{avail} = \{\{[\texttt{medA}]\}(\emptyset), \{[\texttt{medB}]\}(\emptyset)\}$$
$$min\_unavail_r = 0 \text{ and } max\_unavail_r = 2$$

There are three BCP steps in r: the first one on $[\neg\texttt{aids}]$, the second one on $[\neg\texttt{cancer}]$ and the third one on $[\neg\texttt{medA}]$. This yields the partial interpretation $I_r(\texttt{aids}) = I_r(\texttt{cancer}) = I_r(\texttt{medA}) = 0$; from this we compute the bound $min\_unavail_r = 1$.

Next we split on medB: we construct the child nodes *left* and *right* and begin with $I_{left}(\texttt{medB}) = 1$; the resulting bounds after simplification are:

$$min\_unavail_{left} = max\_unavail_{left} = 1$$

Simplification also produces the new unit clause $[\texttt{flu}]$; with BCP on this clause we have found a complete and satisfying interpretation in this branch and we set $I_{best} = I_{left}$ with $max\_unavail_{best}$=1.

In node *right* we set $I_{right}(\texttt{medB}) = 0$. We find that $min\_unavail_{left} = 2$, but $2 > max\_unavail_{best}$; this means, a solution better than the current I_{best} cannot be found in this branch. That is why we prune this branch.

We found the optimal solution in the first branch: $I'(\texttt{medB}) = I'(\texttt{flu}) = 1$ and $I'(\texttt{aids}) = I'(\texttt{cancer}) = I'(\texttt{medA}) = 0$; the transformed database is $db' = \{\texttt{medB}, \texttt{flu}\}$ with distance $avail\_dist = 1$. We have not considered db_dist here as there is only one solution with maximum availability.

4.3 Optimizations and Extensions

First of all, a vast number of optimization techniques and splitting heuristics have been proposed for the basic DPLL-algorithm; for example, subsumption removal [23], reduction of the number of clauses [12] or elimination of variables [19]. Such techniques can be employed to speed up the search process.

Furthermore, there exist several adaptations of the basic DPLL-algorithm to non-CNF formulas. For example, Giunchiglia et al. [15] apply renaming of subformulas by adding new variables (thus constructing an equisatisfiable formula in CNF for a non-CNF input formula) but propose to split only on the original ("independent") variables. Alternatively, Ganai et al. [14] introduce hybrid SAT for boolean circuits where only newly added ("learned") clauses are in CNF; propagation in circuits is supported by a lookup table and "watched literals" (see also [24]). Thiffault et al. [20] also represent the non-CNF input formula as a boolean circuit and base propagation very efficiently on "watch children" and "don't care watch parents". In the boolean circuit approaches, new propagation techniques are defined for the different boolean operators.

The boolean circuit approaches seem to be the most promising when extending our algorithm to the non-CNF case for the following reasons:

- as we want to find a satisfying interpretation for all (now arbitrary) formulas in *prior* as well as $Neg(pot\_sec)$, we can represent each formula by a boolean circuit and use splitting as well as non-CNF propagation techniques to simplify the circuits step-by-step
- as for availability preservation, we can represent each (now arbitrary) formula Θ in *avail* by a boolean circuit and maximize availability by minimizing the number of circuits that are evaluated differently analogously to our algorithm; we can also abandon the usage of flags and instead set the "top level node" (see [20]) of the circuit representing Θ to $eval^*(\Theta)(db)$ and count the number of inconsistent circuits for the availability distance

In order to give the database administrator greater flexibility when specifying the policies, instead of solving a SAT problem based on *prior* and $Neg(pot\_sec)$ and a minimization problem with *avail*, we can extend the algorithm to solve a hierarchical SAT problem: while only *prior* is in hierarchy level 0 (and thus has to be fully satisfied), there can be other hierarchy levels consisting of alternating sets of confidentiality and availability policies that have to be satisfied as good as possible while lower levels take precedence over higher levels. This way, the administrator can specify fine-grained confidentiality and availability requirements of differing importance. In this setting, we can also skip requirement (e) (see Section 2) as we now solve a MAX-SAT problem for the negations of the potential secrets and not all secrets have to be protected. This approach has been published in a more generalized form for a hierarchical constraint solver in [5].

Last but not least, if the number of changes in db' (that it the distortion distance) is very small, it may be more efficient to just maintain a small separate database db'', that contains all those entries of db' that were added (as positive entries) or deleted (as negative entries) in comparison to db. db'' returns the truth values for all changed variables; the original database db returns the truth values for the rest (that is, only if db'' did not return a result).

4.4 Implementational Issues and Applications

Our approach can profit from existing SAT solver implementations, particularly solvers for the weighted MAX-SAT problem (where weights are assigned to clauses and the summed weight of the satisfied clauses is maximized). To benefit from the expertise of the developers of these solvers, we implemented a tool that translates the original database and the input formulas into a weighted MAX-SAT instance. As input format we use the TPTP (http://www.tptp.org/) syntax and take advantage of the variety of TPTP tools for translations into solver-specific SAT formats (e.g., DIMACS). Our tool has been tested with the solvers MiniMaxSAT (see [17]), MAX-DPLL (Toolbar; see [18]) and SAT4J (http://www.sat4j.org/).

We made test runs with relational databases of the following form:

Ill	*Name*	*Diagnosis*
	Pete	Cancer
	Mary	Flu
	Mary	Cancer
	⋮	⋮

Treat	*Name*	*Medicine*
	Pete	MedA
	Mary	MedB
	⋮	⋮

Each attribute has a fixed finite domain such that the problem can be fully propositionalized: a new propositional variable is introduced for each ground fact (e.g., `pete_cancer` for the ground fact $Ill(\mathsf{Pete}, \mathsf{Cancer})$). We made test runs with domain size up to 2400 for the *Name* attribute, domain size 3 for the *Diagnosis* attribute and domain size 2 for the *Medicine* attribute. The propositionalized input database *db* consists of those propositional variables whose ground facts are included in the relational tables. Entries are permuted randomly in *db*. The input clauses analogous to the propositional example (as for *prior* e.g., $\forall x : \neg Treat(x, \mathsf{MedA}) \vee Ill(x, \mathsf{Aids}) \vee Ill(x, \mathsf{Cancer})$) are propositionalized accordingly for all possible combinations (e.g., $\neg$`pete_medA` $\vee$ `pete_aids` $\vee$ `pete_cancer`) – yielding 9600 clauses in total. With the given domains, per *Name* value there are 24 admissable combinations of *Diagnosis* and *Medicine* values that are consistent with *prior*. We uniformly distributed those combinations in the database, leading to up to 4400 tuples in *Ill* and up to 2200 tuples in *Treat*. With MiniMaxSAT, solutions were found in less than a second.

5 Conclusion and Future Work

We presented an algorithm to preprocess an inference-proof and availability-preserving database based on a user's a priori knowledge and specifications of secret information on the one hand and (wrt. availability) important information on the other hand. While the worst-case runtime is exponential (in the number of propositional decision variables), there is a good chance to find an acceptable (or even optimal) solution in a smaller amount of time. We employed several state-of-the-art weighted MAX-SAT solvers that efficiently compute a solution database. Note that with the presented algorithm, even if we stopped

the algorithm prematurely and accepted a suboptimal solution, this solution would be definitely inference-proof; it might only be suboptimal with respect to availability: not all important information might be preserved and there may be more lies in the database than necessary.

Future work shall investigate how this approach can be adapted to other CQE parameters, namely, the unknown policy case, refusal as a restriction method and incomplete databases. Furthermore, this method shall be expanded to other logics (for example, first-order logic with an infinite underlying domain analogously to [4]). Moreover, a comparison of the CQE system to existing approaches for general purpose databases (see e.g. [13]) is in progress.

6 Acknowledgments

We are much obliged to the developers of the cited SAT solver programs and to Cornelia Tadros for her implementation of the translation tool. We also thank the anonymous referees whose comments helped improve the overall presentation of this article.

References

[1] Joachim Biskup and Piero A. Bonatti. Lying versus refusal for known potential secrets. *Data & Knowledge Engineering*, 38(2):199–222, 2001.

[2] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for enforcing confidentiality in complete information systems. *International Journal of Information Security*, 3(1):14–27, 2004.

[3] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Annals of Mathematics and Artificial Intelligence*, 40(1-2):37–62, 2004.

[4] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation with open queries for a decidable relational submodel. *Annals of Mathematics and Artificial Intelligence*, 50(1-2):39–77, 2007.

[5] Joachim Biskup, Dominique Marc Burgard, Torben Weibert, and Lena Wiese. Inference control in logic databases as a constraint satisfaction problem. In *Third International Conference on Information Systems Security, Proceedings*, volume 4812 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2007.

[6] Joachim Biskup and Torben Weibert. Keeping secrets in incomplete databases. *International Journal of Information Security*, 2007. `http://www.springerlink.com/content/g0565w1705t2155u/`.

[7] Joachim Biskup and Lena Wiese. On finding an inference-proof complete database for controlled query evaluation. In Ernesto Damiani and Peng Liu, editors, *20th Annual IFIP WG 11.3 Conference on Data and Applications Security, Proceedings*, volume 4127 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2006.

[8] LiWu Chang and Ira S. Moskowitz. A study of inference problems in distributed databases. In Ehud Gudes and Sujeet Shenoi, editors, *16th Annual IFIP WG*

11.3 Conference on Data and Applications Security, Proceedings, pages 191–204. Kluwer, 2002.

[9] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[11] Josep Domingo-Ferrer, editor. *Inference Control in Statistical Databases, From Theory to Practice*, volume 2316 of *Lecture Notes in Computer Science*. Springer, 2002.

[12] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *8th International Conference on Theory and Applications of Satisfiability Testing, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[13] Csilla Farkas and Sushil Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–11, 2002.

[14] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *39th Design Automation Conference, Proceedings*, pages 747–750. ACM, 2002.

[15] Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In Evelina Lamma and Paola Mello, editors, *6th Congress of the Italian Association for Artificial Intelligence, Proceedings*, volume 1792 of *Lecture Notes in Computer Science*, pages 84–94. Springer, 2000.

[16] John Hale and Sujeet Shenoi. Analyzing FD inference in relational databases. *Data & Knowledge Engineering*, 18(2):167–183, 1996.

[17] Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSAT: An efficient Weighted Max-SAT Solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[18] Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 172(2-3):204–233, 2008.

[19] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. Niver: Non increasing variable elimination resolution for preprocessing SAT instances. In *7th International Conference on Theory and Applications of Satisfiability Testing, Proceedings*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[20] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In *7th International Conference on Theory and Applications of Satisfiability Testing, Online Proceedings*, 2004.

[21] Lingyu Wang, Yingjiu Li, Duminda Wijesekera, and Sushil Jajodia. Precisely answering multi-dimensional range queries without privacy breaches. In Einar Snekkenes and Dieter Gollmann, editors, *8th European Symposium on Research in Computer Security, Proceedings*, volume 2808 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2003.

[22] Xiaochun Yang and Chen Li. Secure XML publishing without information leakage in the presence of data inference. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *30th International Conference on Very Large Data Bases, Proceedings*, pages 96–107. Morgan Kaufmann, 2004.

[23] Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In Fahiem Bacchus and Toby Walsh, editors, *8th International Conference on Theory and Applications of Satisfiability Testing, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer, 2005.

[24] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *18th International Conference on Automated Deduction, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.