Controlled Query Evaluation and Inference-free View Updates

Joachim Biskup, Jens Seiler, and Torben Weibert

Technische Universität Dortmund, Dortmund, Germany
{biskup|seiler|weibert}@ls6.cs.uni-dortmund.de

Abstract. We extend Controlled Query Evaluation (CQE), an inference control method to enforce confidentiality in static information systems under queries, to updatable databases. Within the framework of the lying approach to CQE, we study user update requests that have to be translated into a new database state. In order to avoid dangerous inferences, some such updates have to be denied even though the new database instance would be compatible with a set of integrity constraints. In contrast, some other updates leading to an incompatible instance should not be denied. We design a control method to resolve this seemingly paradoxical situation and then prove that the general security definitions of CQE and other properties linked to user updates hold.

1 Introduction

The enforcement of confidentiality in information systems, that is allowing access to information only to authorised users, is often done by static mechanisms like *discretionary access control* or *mandatory access control*. While these mechanisms can prevent unauthorised users from a direct access to secret information such users might still be able to gather enough data to infer secrets indirectly. This is known as the *inference problem* [FJ02]. Inference channels can be created by various means, in particular by observing the system's behaviour linked to integrity constraints and other a priori knowledge about databases.

Controlled Query Evaluation (CQE) is a dynamic control method that counteracts such inferences. A comprehensive overview of CQE can be found in [BB04a]; see also [BW08]. All previous work on CQE deals with confidentiality in static databases. In this paper, we want to enable users of an information system to alter the contents of the database by means of issuing update requests, which should be appropriately reflected in the database.

Enabling users to update information is known to result in unwanted inferences as can be seen in *multilevel secure (MLS) databases* and the phenomenom of *polyinstantiation* (see, for example, [DA87,LD90,JS91,SJ92,CG99,CG01]). In MLS databases there exists data on high levels that is invisible to users on a lower level. If a low level user tries to insert data that does already exist on a higher level such an update request cannot be denied or otherwise the user would be able to infer the existence of that data on a higher, secret level. Therefore the

update is accepted and the same data entity exists on two or more levels; the database becomes polyinstantiated. Controlled Query Evaluation shares certain similarities with polyinstantiated MLS databases in that CQE too has multiple levels. While the database administrator has a complete and exact knowledge of the database instance, the (unprivileged) database user constructs his own belief on the database instance using his a priori knowledge and the data that he has explicitly obtained from the system, namely the potentially modified answers.

We consider the a priori knowledge and the obtained answers to previous queries as the user's current *view* on the database instance. Thus we treat the data obtained similarly to the traditional database management, where a "view" is syntactically defined as an identified query, and semantically interpreted either dynamically or statically, by "materialising" the answer to the identified query. A classical problem is the question of how to translate an update request that refers to a view to the underlying database instance, in particular how to resolve ambiguities (see, for example [BS81,DB82,La90,He04,BP06]). In this paper, we will avoid the ambiguity problem by only allowing requests to change the truth value of atomic sentences.

However, we identify and solve the problem of how to process such requests in an inference-free way. More specifically, in Sect. 2, we summarize query processing under the lying form of CQE. Then, in Sect. 3, we increase the interaction of the user level with the administrator level by view updates. This feature adds new sources of possibly dangerous inferences that, however, we can hope to control by suitably combining a simple means with the concept of lying. In Sect. 4, we design a full update algorithm in detail, which complements inference-free query answering with inference-free view updating, and afterwards, in Sect. 5, we exemplify a run of this algorithm. Then, in Sect. 6, the important property of confidentiality is proven to hold. Finally, in Sect. 7, we show that any view update issued by the user can be undone by him.

2 Controlled Query Evaluation with lying

We exploit a well-known logic-oriented approach to information systems, which provides formal semantics for both query answering and updating including enforcement of constraints. In particular, we assume the following prerequisites:

- A database *instance* db is a complete interpretation of a propositional logic (leaving generalisations to first-order logic [BB07] or to incomplete information systems [BW08] open for a future elaboration), such that a sentence of a propositional logic is either *true* or *false* with respect to a database instance but never undefined; an instance can be specified by listing for each atomic proposition χ whether that proposition or its negation should be included.
- A *query* ϕ is a (closed) sentence in the logic (suggesting again a generalisation to open queries [BB07] for future work).
- Accordingly, a query ϕ is *evaluated* relative to an instance db by the function $eval(\phi)(db)$ returning the pertinent truth value or, equivalently, either ϕ or $\neg\phi$, which is denoted by $eval^*(\phi)(db)$.

– We employ the usual notion of logical *implication*, denoted by $\models$.

Furthermore, we deal with inference control in a specific form of Controlled Query Evaluation that modifies a potentially harmful answer to a query by *lying* [BK95,BB01,BW08], that is returning the negation of the true answer (leaving the alternative forms of *refusal* [SJ83,BB01,BW08], that is returning *mum* instead of the true answer, and a *combined* form [BB04b] open for further research). The lying approach is based on the following additional prerequisites:

– The *confidentiality policy*, supposed to be *known* to the user, is declared as a set of potential secrets, that is a finite set *pot_sec* of sentences to be protected so that the user can never infer that any sentence $\psi \in pot\_sec$ actually holds.
– In order to preserve this kind of confidentiality, we have to protect not only the individual potential secrets but, in fact, the disjunction of all potential secrets $pot\_sec\_disj := \bigvee_{\psi \in pot\_sec} \psi$. This is necessary to avoid "hopeless situations" which lead to an inconsistency (see [BB01,BB04a] for further explanation).
– Basically, the control mechanism maintains a *user log log* for keeping the (postulated) a priori knowledge of the user and the answers returned to previous queries, and invokes a *censor* $censor^L$ for inspecting whether in a given situation the true answer $eval^*(\phi)(db)$ to a query ϕ will be harmful; more formally the function $censor^L$ returns a Boolean value as follows:

$$censor^L(db, pot\_sec, log, \phi) := log \cup \{eval^*(\phi)(db)\} \models pot\_sec\_disj. \quad (1)$$

– CQE (with uniform lying) is then a function $control\_eval(Q, log_0)(db, pot\_sec)$ where Q is any finite *sequence of (closed) queries* $Q = \langle \phi_1, \phi_2, \ldots, \phi_n \rangle$, log_0 is an initial *user log* representing the a priori *user knowledge*, *db* is a *database instance* and *pot_sec* is the *confidentiality policy* to be enforced. The function *control_eval* returns a corresponding sequence of answers and updated user logs (ans_i, log_i). The return values are calculated by censoring the true answer of a user's query ϕ_i and in case of unwanted inferences by applying a *modification* (by lying) to the answer:

$$\begin{aligned} ans_i := \ & \text{if } censor^L(db, pot\_sec, log_{i-1}, \phi_i) \text{ then } \neg eval^*(\phi_i)(db) \\ & \text{else } eval^*(\phi_i)(db) \end{aligned} \quad (2)$$

$$log_i := log_{i-1} \cup \{ans_i\} \quad (3)$$

It was shown in [BB01] that the confidentiality property expressed by the following definition holds for this variant of CQE.

Definition 1 (Confidentiality). *A function* $control\_eval(Q, log_0)(db, pot\_sec)$ preserves confidentiality *iff for all finite sequences of queries Q, all initial user knowledges* log_0*, all instances db satisfying* log_0*, all sets of potential secrets pot_sec and all potential secrets* $\psi \in pot\_sec$ *with* $log_0 \not\models \psi$ *there exists an instance* db^S *satisfying* log_0 *such that*

1. *db and* db^S *return the same sequence of answers:*
   $control\_eval(Q, log_0)(db, pot\_sec) = control\_eval(Q, log_0)(db^S, pot\_sec)$
2. db^S *does not contain the potential secret* ψ*:* $eval^*(\psi)(db^S) = \neg\psi$

3 Inferences through view updates

Before formally introducing a model for view updates under CQE, we take an informal look at user modifications which demonstrate the necessity for a non-trivial means of handling a user's update requests. Simply allowing all "accepted" updates or forbidding all "denied" updates leads to inferences as the following examples show.

Example 1 (Inferences through accepted update requests). Suppose, a user issues an update request on a database restricted by an integrity constraint, e.g., $a \vee b$. He wants to ensure that the interpretation of a is *true*. If the system responds with "Value of a changed to *true*", the user can infer that previously a was *false* and that then b must have been and still is *true*.

If b is a secret to be protected then clearly the update request would have had to be denied. Even if the truth of b is not to be kept secret, the system has in some way to keep track of the user's ability to infer the truth of b in order to prevent future inferences.

Example 2 (Inferences through denied update requests). Suppose, a database instance $\{\neg a, b\}$ underlies the integrity constraint $\neg a \vee \neg b$. Again a user issues a request to ensure the truth of a. This, however, would lead to an integrity violating instance $\{a, b\}$. Denying the update request enables the user to reason about the value of b: if b was *false*, then the value of a would not matter. But since the update request was denied, b must in fact be *true*.

Again the user is able to infer the truth value of a variable he doesn't directly change or query. Interestingly enough, such inferences can be easily calculated in advance and thus be encountered. The necessary tool for that is a simple negation of variables in the formulas of propositional logic, as described next.

We define the set $FORMULA$ of allowed propositional formulas in the usual inductive way: *true*, *false* and each $v \in VAR$ are elements of $FORMULA$; if t_1 and t_2 are elements of $FORMULA$, then also $\neg t_1$, $(t_1 \wedge t_2)$ and $(t_1 \vee t_2)$. We will omit brackets where not necessary, and omit double negation where convenient.

While a variable serves as a placeholder for a truth value we will use literals as a means to specify such a truth value for a given variable:

Definition 2 (Literal). *For every variable $v \in VAR$, v and $\neg v$ are literals of the set LIT. For a literal $\chi \in \{v, \neg v\}$ with $v \in VAR$, we define $\chi^+ := v$. That way the symbol χ^+ is syntactically identical to the variable whose truth value is being defined by the literal χ.*

We can now define the negation of variables on formulas. Although we negate variables we will specify the variable to be negated by a literal. This is done to simplify the usage of variable negation later on when a user specifies the updated truth value of a variable using a literal.

Definition 3 (Negation of variables on formulas). *The negation of variables on formulas* $neg(\cdot,\cdot) : FORMULA \times LIT \longrightarrow FORMULA$ *is defined by*

$$neg(\phi,\chi) := \begin{cases} \phi & \text{for } \phi \in \{false, true\} \\ \phi & \text{for } \phi = v \neq \chi^+, v \in VAR \\ \neg\phi & \text{for } \phi = \chi^+ \\ \neg\big(neg(\phi',\chi)\big) & \text{for } \phi = \neg\phi' \\ \big(neg(\phi',\chi) \wedge neg(\phi'',\chi)\big) & \text{for } \phi = \phi' \wedge \phi'' \\ \big(neg(\phi',\chi) \vee neg(\phi'',\chi)\big) & \text{for } \phi = \phi' \vee \phi'' \end{cases} \tag{4}$$

Informally, every appearance of a variable specified by a literal χ *in the formula* ϕ *is negated. We call* $neg(\phi,\chi)$ *the* χ*-negated formula* ϕ*. Likewise, the negation of variables on a set of formulas* $neg(\cdot,\cdot) : \mathcal{P}(FORMULA) \times LIT \longrightarrow \mathcal{P}(FORMULA)$ *is defined by*

$$neg(M,\chi) := \{\ neg(\phi,\chi)\ \mid\ \phi \in M\ \}. \tag{5}$$

Example 3 (Negation of variables on a formula).

$$\begin{aligned} neg\big(\neg(a \wedge b) \vee \neg a, \neg a\big) &= neg\big(\neg(a \wedge b), \neg a\big) \vee neg(\neg a, \neg a) && (6)\\ &= \neg\big(neg(a \wedge b, \neg a)\big) \vee \neg\big(neg(a, \neg a)\big) && (7)\\ &= \neg\big(neg(a, \neg a) \wedge neg(b, \neg a)\big) \vee \neg(\neg a) && (8)\\ &= \neg(\neg a \wedge b) \vee a && (9) \end{aligned}$$

We can now identify a simple property of variable negation as defined above:

Lemma 1 (Negation equivalence). *For any instance* db*, all* $\phi \in FORMULA$*, any literal* χ*, using the following definition of* db^χ *via* db*,*

$$db^\chi := \begin{cases} \big(db \setminus \{\chi\}\big) \cup \{\neg\chi\} & \text{for } \chi \in db \\ \big(db \setminus \{\neg\chi\}\big) \cup \{\chi\} & \text{otherwise} \end{cases} \tag{10}$$

we have that

$$eval(\phi)(db) = eval\big(neg(\phi,\chi)\big)\big(db^\chi\big). \tag{11}$$

This means that we get the same results evaluating a formula on an instance and evaluating the χ*-negated formula on the instance created by negating the variable specified by* χ*.*

Lemma 1 can now be used to identify inferences as they appeared in Examples 1 and 2. We claim (and prove below) that

- the set $neg(log,\chi)$ contains valid formulas *after* changing the truth value of χ, if beforehand the formulas contained in *log* were true, and
- the formula $neg\big(\neg(\bigwedge_{\phi \in constraints} \phi),\chi\big)$ describes the knowledge gained by learning that the set of *constraints* doesn't hold under an interpretation where the truth value of the variable specified by χ has been negated, provided that the *constraints* were valid beforehand.

This can be utilised to create a secure, that is inference-free, view update mechanism for Controlled Query Evaluation under a uniform lying censor.

4 Inference-free view updates under CQE with lying

Definition 4 (Controlled Query Evaluation with view updates). *We define a sequence Q of queries* and update requests *by:*

$$Q := \langle\ \Theta_1, \Theta_2, \ldots, \Theta_i, \ldots \Theta_k\ \rangle \quad \textit{with} \tag{12}$$

$$\Theta_i := \begin{cases} \Phi_i & \textit{a query with } \Phi_i \in \textit{FORMULA or} \\ \textit{update}(\chi_i) & \textit{a (view) update operation with } \chi_i \in \textit{LIT} \end{cases} \tag{13}$$

Additionally we have the following:

- $\textit{constraints} \subseteq \mathcal{P}(\textit{FORMULA})$ *is a finite set of constraints, which have to be satisfied before and after each update,*
- $\textit{log}_0 \subseteq \mathcal{P}(\textit{FORMULA})$ *is an initial set of the assumed user knowledge with* $\textit{log}_0 \supseteq \textit{constraints}$*,*
- $\textit{db}_0$ *is an initial database instance and*
- $\textit{pot\_sec} \subseteq \mathcal{P}(\textit{FORMULA})$ *is a finite set of potential secrets.*

Then we define Controlled Query Evaluation with view updates by

$$\textit{control\_eval\_update}(Q, \textit{log}_0, \textit{constraints})(\textit{db}_0, \textit{pot\_sec}) \tag{14}$$

$$= \big\langle (\textit{ans}_1, \textit{log}_1, \textit{db}_1), \ldots, (\textit{ans}_i, \textit{log}_i, \textit{db}_i), \ldots (\textit{ans}_k, \textit{log}_k, \textit{db}_k) \big\rangle \tag{15}$$

For queries Φ_i, we define the triple $(\textit{ans}_i, \textit{log}_i, \textit{db}_i)$ as in normal CQE with $\textit{db}_i := \textit{db}_{i-1}$. Update requests $\textit{update}(\chi_i)$ are defined by the algorithm described in the following and formalised in Def. 5.

Using the properties of *neg* as identified in Sect. 3 we can describe an algorithm that provides for inference-free view updates. The algorithm consists of four steps which also represent four disjunct cases determining the response to the user. These cases can be outlined roughly as follows:

1. The requested update is already compatible to the user's view and thus the database instance is not to be modified.
2. Allowing the requested update would infer a secret or be incompatible with the set of constraints and this fact is known to the user a priori.
3. Allowing the requested update would be incompatible with the set of constraints and this is unknown to the user a priori.
4. The requested update is accepted and the user receives confirmation.

In the following, we shall describe each of the cases in some more detail.

1. The first case is comparable with the property of acceptability in Def. 3.1 in [BS81], where view updates without confidentiality requirements are studied. An update that is already compatible with the current view needs not to be performed. In our case, a user's request to update χ_i is compatible with his current view if we have that

$$\textit{control\_eval}\big(\langle \chi_i \rangle, \textit{log}_{i-1}\big)\big(\textit{db}_{i-1}, \textit{pot\_sec}\big) = \langle (\chi_i, \textit{log}_{i-1} \cup \{\chi_i\}) \rangle. \tag{16}$$

Under the lying censor as defined in Sect. 2 this is equivalent to:

$$eval^*(\chi_i)(db_{i-1}) = \chi_i \quad AND\ log_{i-1} \cup \{\chi_i\} \not\models pot\_sec\_disj \tag{17}$$
$$OR\ eval^*(\chi_i)(db_{i-1}) = \neg\chi_i\ AND\ log_{i-1} \cup \{\neg\chi_i\} \models pot\_sec\_disj \tag{18}$$

So, if that condition holds we will tell the user that χ is already valid and update the log accordingly.

2. If the first case did not occur, then obviously $\neg\chi$ is valid from the user's point of view and a view update can take place if confidentiality or consistency isn't threatened. To verify this we check if the updated view would imply the disjunction of all potential secrets. We claim that this is the case if the following condition holds:

$$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \models pot\_sec\_disj \tag{19}$$

Interestingly we don't use the actual instance db_{i-1} and thus the aforementioned condition can be evaluated purely from information available to the user. Consequently the only additional information learned by the user is the passing of the first case and thus the fact that $\neg\chi_i$ is true within his view. This fact is added to the log.

3. While the previous case takes care of updates leading to inconsistencies of which the user is aware himself, the introducing Example 2 shows that there are also cases in which a user doesn't know that his update would lead to an inconsistent instance and additionally where the user *isn't allowed to know* that this would be the case.
We introduce $con\_conj := \bigwedge_{\phi \in constraints} \phi$ as the conjunction of all constraints and can easily verify if the future instance would be inconsistent after the user's update:

$$eval\big(con\_conj\big)\big((db_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}\big) = false \tag{20}$$

Additionally we have to check if telling the user about such an inconsistency would enable him to infer the disjunction of the potential secrets. Again we can make use of the properties of *neg* and require the following condition to be true in order to tell the user about an inconsistency found by the previous condition (20):

$$log_{i-1} \cup \{\neg\chi_i\} \cup \big\{neg(\neg con\_conj, \chi_i)\big\} \not\models pot\_sec\_disj \tag{21}$$

If both (20) and (21) are true then the user is informed about the inconsistency and his update request is denied. Additionally the log has to be updated by adding $\{\neg\chi_i\} \cup \big\{neg(\neg con\_conj, \chi_i)\big\}$.

4. This last case automatically occurs if all conditions of the three cases before do not hold. From the user's point of view the update is to be accepted now and consequently this fourth case notifies the user about his successful update. The log is being updated with the premise of condition (19) from the second case and we have $log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \not\models pot\_sec\_disj$.

However, as we learned from the third case there are updates that would be inconsistent with the set of constraints of the database scheme but which are nevertheless to be allowed in order to protect secrets. Therefore, in case condition (20) was indeed true but the protecting condition (21) was false, we won't update the actual database instance making the fact that we tell the user about a successful update a lie.

As we see from the description of the four cases, the proposed algorithm employs lies at two places: Firstly, it can lie about the current view of the user before the update and tell him either that his requested update is already compatible with the database instance although it isn't (then (18) is true) or the database instance already contains the literal as desired by the requested update but telling so would implicate a secret or inconsistency (and thus condition (17) is *not* true). Secondly, we may lie to the user about the affirmation of a successful update despite not touching the actual instance in the fourth case.

Given the brief outline and the more detailed descriptions, we are now ready to declare the algorithm formally.

Definition 5 (A secure view update algorithm for CQE).
`if` (condition for case 1)

$$eval^*(\chi_i)(db_{i-1}) = \chi_i \quad AND\ log_{i-1} \cup \{\chi_i\} \not\models pot\_sec\_disj \tag{22}$$

$$OR\ eval^*(\chi_i)(db_{i-1}) = \neg\chi_i\ AND\ log_{i-1} \cup \{\neg\chi_i\} \models pot\_sec\_disj \tag{23}$$

`then`

- $db_i := db_{i-1}$, $log_i := log_{i-1} \cup \{\chi_i\}$
- $ans_i :=$ *"The requested update is already contained in the database"*

`else if` (condition for case 2)

$$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \models pot\_sec\_disj \tag{24}$$

`then`

- $db_i := db_{i-1}$, $log_i := log_{i-1} \cup \{\neg\chi_i\}$
- $ans_i :=$ *"Updating χ_i is inconsistent with secrets or integrity"*

`else if` (condition for case 3)

$$eval\big(con\_conj\big)\big((db_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}\big)\ = false \tag{25}$$

$$AND\ log_{i-1} \cup \{\neg\chi_i\} \cup \big\{neg(\neg con\_conj, \chi_i)\big\}\ \not\models pot\_sec\_disj \tag{26}$$

`then`

- $db_i := db_{i-1}$, $log_i := log_{i-1} \cup \{\neg\chi_i\} \cup \big\{neg(\neg con\_conj, \chi_i)\big\}$
- $ans_i :=$ *"Updating χ_i is incompatible with integrity"*

`else` (case 4)

- *`if` condition* (25) *`then`* $db_i := db_{i-1}$ *`else`* $db_i := (db_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}$
- $log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints$
- $ans_i :=$ *"Update of χ_i successful"*

5 An example

To illustrate the algorithm of Def. 5 we give an example that will trigger all of the algorithm's cases:

- $pot\_sec := \{s_1, s_2\}$
- $constraints := \{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c\}$
- $db_0 := \{a, \neg b, \neg c, s_1, s_2\}$
- $log_0 := \{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c\}$
- $Q := \langle update(\neg a), update(c), update(b), update(c), update(a), update(b)\rangle$

1. $update(\neg a)$ will trigger case 1 due to condition (23):

$$eval^*(\neg a)(\{a, \neg b, \neg c, s_1, s_2\}) = a \; AND \; \{a \Rightarrow \; s_1, c \Rightarrow b, s_2 \Rightarrow \neg c\} \cup \{a\} \models s_1 \vee s_2$$

The algorithm has to lie in order to protect the disjunction of secrets. Despite a being true under db_0 it has to tell the user that $\neg a$ is already true since otherwise the user could infer that a was valid which would have implied the truth of the secret s_1. We thus get:

- $db_1 := \{a, \neg b, \neg c, s_1, s_2\}$
- $log_1 := \{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c, \neg a\}$
- $ans_1 :=$ "The requested update is already contained in the database"

2. $update(c)$ will trigger case 3 since we have firstly, that an inconsistent database instance would be created (equation (25) holds):

$$eval\big(\{a \Rightarrow s_1 \wedge c \Rightarrow b \wedge s_2 \Rightarrow \neg c\}\big)\big((\{a, \neg b, \neg c, s_1, s_2\} \setminus \{\neg c\}) \cup \{c\}\big) = false$$

and secondly, that this can be told to the user without implying a secret (equation (26) holds):

$$\{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c, \neg a\} \cup \{\neg c\} \cup \big\{neg(\neg con\_conj, c)\big\} \not\models \; pot\_sec\_disj$$

This is because with $\{\neg a, \neg b, \neg c, \neg s_1, \neg s_2\}$ we have a "witness" instance that makes true the premise of the implication but falsifies the conclusion. The same witness instance can be used to verify that indeed cases 1 and 2 will not be triggered. We now have:

- $db_2 := \{a, \neg b, \neg c, s_1, s_2\}$
- $log_2 := \{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c, \neg a\} \cup \{\neg c\} \cup neg\big(\neg(a \Rightarrow s_1 \wedge c \Rightarrow b \wedge s_2 \Rightarrow \neg c), c\big)$
  $= \{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c, \neg a, \neg c, \neg(a \Rightarrow s_1 \wedge \neg c \Rightarrow b \wedge s_2 \Rightarrow c)\}$
- $ans_2 :=$ "Updating c is incompatible with integrity"

3. $update(b)$ will trigger case 4 and modify the instance since neither is case 1 triggered (the same "witness" $\{\neg a, \neg b, \neg c, \neg s_1, \neg s_2\}$ can be used to verify this) nor is case 2 triggered which can be verified using the witness instance $\{\neg a, b, \neg c, \neg s_1, \neg s_2\}$. We also have that case 3 isn't triggered because $eval(con\_conj)(\{a, b, \neg c, s_1, s_2\}) = true$. Therefore a true update of the instance is done and we get:

- $db_3 := \{a, b, \neg c, s_1, s_2\}$
- $log_3 := neg\big(\{a \Rightarrow s_1, c \Rightarrow b, s_2 \Rightarrow \neg c, \neg a, \neg c, \neg(a \Rightarrow s_1 \wedge \neg c \Rightarrow b \wedge s_2 \Rightarrow c)\}, b\big) \cup \{b\} \cup \mathit{constraints}$
  $= \{a \Rightarrow s_1, c \Rightarrow \neg b, s_2 \Rightarrow \neg c, \neg a, \neg c, \neg(a \Rightarrow s_1 \wedge \neg c \Rightarrow \neg b \wedge s_2 \Rightarrow c), b, c \Rightarrow b\}$
- $ans_3 :=$ "Update of b successful"

For those readers not willing to solve the logic puzzle of the now quite stuffed log we hint that only four instances remain possible: $\{\neg a, b, \neg c, \neg s_1, \neg s_2\}$, $\{\neg a, b, \neg c, \neg s_1, s_2\}$, $\{\neg a, b, \neg c, s_1, \neg s_2\}$ and $\{\neg a, b, \neg c, s_1, s_2\}$. As such the user has gained a complete knowledge about the non-secrets a, b and c while it appears possible to him that all secrets are false.
4. $update(c)$ will trigger case 4 but not modify the instance. Again cases 1 and 2 are not triggered as the reader can easily verify. This time, however, we have that a real update would create an instance not compatible with the set of constraints. The instance $\{a, b, c, s_1, s_2\}$ violates the constraint $s_2 \Rightarrow \neg c$. However, telling this to the user would imply the truth of s_2 since from the users point of view the other two constraints cannot be violated. Luckily our condition (26) of the update algorithm protects us from relating a violation of constraints to the user by remaining "silent" and thus telling him the lie of a successful update:

- $db_4 := \{a, b, \neg c, s_1, s_2\}$
- $log_4 := neg\big(\{a \Rightarrow s_1, c \Rightarrow \neg b, s_2 \Rightarrow \neg c, \neg a, \neg c, \neg(a \Rightarrow s_1 \wedge \neg c \Rightarrow \neg b \wedge s_2 \Rightarrow c), b, c \Rightarrow b\}, c\big) \cup \{c\} \cup \mathit{constraints}$
  $= \{a \Rightarrow s_1, \neg c \Rightarrow \neg b, s_2 \Rightarrow c, \neg a, c, \neg(a \Rightarrow s_1 \wedge c \Rightarrow \neg b \wedge s_2 \Rightarrow \neg c), b, \neg c \Rightarrow b, c \Rightarrow b, s_2 \Rightarrow \neg c$
- $ans_4 :=$ "Update of c successful"

With this update the set of possible instances is reduced to two, namely $\{\neg a, b, c, \neg s_1, \neg s_2\}$ and $\{\neg a, b, c, s_1, \neg s_2\}$.
5. $update(a)$ will trigger case 2 since case 1 isn't triggered and an update of a would imply the secret s_1 which is captured by condition (24) of the algorithm. We now have:

- $db_5 := db_4$
- $log_5 := log_4$ (since $\neg a$ was already in log_4)
- $ans_5 :=$ "Updating a is inconsistent with secrets or integrity"

6. $update(b)$ triggers case 1 via condition (22). We get:

- $db_6 := db_5$
- $log_6 := log_5$ (since b was already in log_5)
- $ans_6 :=$ "The requested update is already contained in the database"

The example shows that every case of the algorithm is reachable and that cases 1 and 4 can both be lying or telling the truth. Additionally it visualises the fact that from the user's point of view there does always exist at least one instance with all secrets false that is consistent with the previous answers given by the update algorithm. In our case these are $\{\neg a, \neg b, \neg c, \neg s_1, \neg s_2\}$ for the instance before the first three updates, $\{\neg a, b, \neg c, \neg s_1, \neg s_2\}$ for the instance after the update of b and $\{\neg a, b, c, \neg s_1, \neg s_2\}$ for the updates after that.

6 Properties of secure view updates

Given the algorithm from Def. 5, the following lemma states that the fundamental invariant of Controlled Query Evaluation under lying (see, for example, [BK95,BB01,BW08]) applies to secure view updates, too.

Lemma 2 (Invariant). *For any instance of Controlled Query Evaluation with view updates control_eval_update with a sequence* $Q = \langle \Theta_1, \Theta_2, \ldots, \Theta_i, \ldots \Theta_k \rangle$ *the following invariant for the user log* log_i *holds:* $log_i \not\models pot\_sec\_disj$

Proof. For queries $\Theta_i = \Phi_i$, the claim follows directly from the properties of normal CQE as shown in [BB01]. For updates $\Theta_i = update(\chi_i)$, we have to argue about the four different possible cases and their log updates:

1. If the update algorithm responds with an answer "The requested update is already contained in the database" then we have that either condition (17) or (18) is true. In the first case we directly get $log_i = log_{i-1} \cup \{\chi_i\} \not\models pot\_sec\_disj$. From the second case we get $log_{i-1} \cup \{\neg\chi_i\} \models pot\_sec\_disj$ and by induction we also have that $log_{i-1} \not\models pot\_sec\_disj$. From both follows $log_i = log_{i-1} \cup \{\chi_i\} \not\models pot\_sec\_disj$.
2. If the update algorithm responds with "Updating χ_i is inconsistent with secrets or integrity" we obviously have neither (17) nor (18) to be true:

$$\begin{aligned} NOT \Big(\quad & (eval^*(\chi_i)(db_{i-1}) = \chi_i \quad AND\ log_{i-1} \cup \{\chi_i\} \quad \not\models pot\_sec\_disj) \\ OR\ & (eval^*(\chi_i)(db_{i-1}) = \neg\chi_i\ AND\ log_{i-1} \cup \{\neg\chi_i\} \models pot\_sec\_disj) \Big) \end{aligned}$$

Using the laws of de Morgan and the completeness of the database instance we obtain the following equivalent expression:

$$\begin{aligned} & (\underbrace{eval^*(\chi_i)(db_{i-1}) = \neg\chi_i}_{a}\ OR\ \underbrace{log_{i-1} \cup \{\chi_i\} \models pot\_sec\_disj}_{b}) \\ AND\ & (\underbrace{eval^*(\chi_i)(db_{i-1}) = \chi_i}_{c}\ OR\ \underbrace{log_{i-1} \cup \{\neg\chi_i\} \not\models pot\_sec\_disj}_{d}) \end{aligned}$$

Thus we have that $(a\ OR\ b)\ AND\ (c\ OR\ d)$ must be true and it follows from the completeness of the database instance that exactly one of a or c can be true. Simply enumerating all combinations of a, b, c and d show that only three such combinations are possible: for $\neg a, b, c, \neg d$ it would follow that

$$log_{i-1} \cup \{\chi_i\} \models pot\_sec\_disj\ AND\ NOT\ \big(log_{i-1} \cup \{\neg\chi_i\} \not\models pot\_sec\_disj\big)$$

and thus

$$log_{i-1} \cup \{\chi_i\} \models pot\_sec\_disj\ AND\ log_{i-1} \cup \{\neg\chi_i\} \models pot\_sec\_disj$$

holds. This however is a contradiction to the induction hypothesis and thus this case cannot occur.
From the two other combinations, that is $\neg a, b, c, d$ and $a, \neg b, \neg c, d$ it follows directly from d that $log_i = log_{i-1} \cup \{\neg\chi_i\} \not\models pot\_sec\_disj$ is true.

3. The third case of the algorithm, the response "Updating χ_i is incompatible with integrity", can only occur if condition (21) holds. Therefore we directly have that

$$log_i = log_{i-1} \cup \{\neg\chi_i\} \cup neg(\neg con\_conj, \chi_i) \not\models pot\_sec\_disj$$

4. Finally, if none of the previous cases were triggered, we have that the update is accepted. From the non-occurrence of the second case we have that condition (19) does not hold, from which we directly obtain that

$$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \not\models pot\_sec\_disj$$

is true, which shows that the log update of fourth case (the last case) is also consistent with the invariant we want to show.

From the invariant $log_i \not\models pot\_sec\_disj$ it follows that the log does not imply any potential secret. This, however, does not mean that the *answers* of the update algorithm will never enable a reasoning user to infer the truth of a secret. In order to show that our claims from the end of Sect. 3 are correct and their usage in the update algorithm provides security we will define Controlled Query Evaluation with view updates to be secure if a user can never infer that any particular potential secret is true in the actual database instance. This requirement is captured by demanding that there always exists an alternative database instance in which the respective potential secret is false, but under which the same answers are returned as under the actual database instance. We therefore adapt Def. 1 so it is compatible with the sequence of produced databases under view updates.

Definition 6 (Confidentiality for view updates). *We say that a function* $control\_eval\_update(Q, log_0, constraints)(db_0, pot\_sec)$ *preserves confidentiality iff for all sequences of queries and update requests* Q*, all initial user knowledges* log_0*, all sets of constraints* $constraints \subseteq log_0$*, all instances* db_0 *satisfying* log_0*, all sets of potential secrets* pot_sec*, all potential secrets* $\psi \in pot\_sec$ *with* $log_0 \not\models \psi$ *there exists an instance* db_0^S *satisfying* log_0*, such that*

1. db_0 *and* db_0^S *return the same sequence of answers:*

$$\upsilon\big(control\_eval\_update(Q, log_0, constraints)(db_0, pot\_sec)\big) = \upsilon\big(control\_eval\_update(Q, log_0, constraints)(db_0^S, pot\_sec))\big)$$

2. db_0^S *does not contain the secret* ψ*:* $\quad eval^*(\psi)(db_0^S) = \neg\psi$

Above, we define υ *to be the projection of a set of tuples of the form* (ans_i, log_i, db_i) *to the user-visible set of answers* ans_i*.*

Theorem 1. *CQE with secure view updates, i.e., the function* $control\_eval\_update$ $(Q, log_0, constraints)(db_0, pot\_sec)$ *as defined by Def. 4 together with Def. 5, preserves confidentiality in the sense of Def. 6.*

To prove this theorem, we need the following lemma, which can easily be proven via a backwards induction, which we omit here due to the lack of space.

Lemma 3. *For any sequence Q with length k there exists a sequence of instances db_i^S with the following two properties (where model_of means "makes true"):*

$$db_{i-1}^S := \begin{cases} (db_i^S \setminus \{\chi_i\}) \cup \{\neg\chi_i\} & \textit{for an update triggering case 4} \quad (27) \\ db_i^S & \textit{otherwise (queries or cases 1 to 3)} \quad (28) \end{cases}$$

$$db_i^S \ \textit{model\_of}\ log_i \cup \{\neg \textit{pot\_sec\_disj}\} \quad (29)$$

Proof (of Theorem 1). We show via induction that our system creates the same sequence of answers for a given sequence Q regardless if it started on db_0 or on db_0^S. This also means that the same sequence of logs is created.

For $i = 0$, we have $log_0 = log_0^S$ and no answers so far. For the step from $i-1$ to i, we differentiate between the nature of the operation $\Theta_i \in Q$. If Θ_i is a query Φ_i, then it follows from Lemma 3 and the proofs of confidentiality for CQE in terms of Def. 1 that the answer returned is the same under db and db^S.

If on the other hand Θ_i is an update $update(\chi_i)$, then we enumerate over the four possible cases of that operation under the instance db_i and show the identical reaction on the corresponding instance db_i^S:

1. From case 1 under db we have that $log_i := log_{i-1} \cup \{\chi_i\}$ and via Lemma 3 it follows that $db_i^S \ \textit{model\_of}\ log_i \subseteq \{\chi_i\}$. Assuming case 1 under db and construction of db_{i-1}^S via (28) we have that $eval^*(db_{i-1}^S)(\chi_i) = \chi_i$ which satisfies the first part of equation (22). We show that the second part, that is $log_{i-1}^S \cup \{\chi_i\} \not\models \textit{pot\_sec\_disj}$, holds too, for if it wouldn't, then via induction we also had $log_i := log_{i-1} \cup \{\chi_i\} \models \textit{pot\_sec\_disj}$, contradicting Lemma 2.
2. First we show that the first case is not triggered under db^S. From case 2 under db it follows that $log_i := log_{i-1} \cup \{\neg\chi_i\}$ and thus for db_i^S via Lemma 3 that $eval^*(\chi_i)(db_i^S) = \neg\chi_i$. For the same reasons we have $eval^*(\chi_i)(db_{i-1}^S) = \neg\chi_i$ which falsifies equation (22). Also, equation (23) is not true; otherwise we had $log_{i-1}^S \cup \{\neg\chi_i\} \models \textit{pot\_sec\_disj}$ contradicting Lemma 2 since we have that log_{i-1} under db is the same as log_{i-1}^S under db^S.
   Also our condition for case 2, namely equation (24) depends only on that log and it follows that case 2 is also triggered under db^S.
3. With the above argument it follows that case 2 is not triggered under db^S if it wasn't triggered under db. Therefore we now have to show that case 3 is triggered under db^S. From case 3 under db we have that $log_i = log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg \textit{con\_conj}, \chi_i)\}$. From Lemma 3 it follows that $db_i^S \ \textit{model\_of}\ log_i$ and via (28) we get $db_{i-1}^S = db_i^S$ resulting in $db_{i-1}^S \ \textit{model\_of}\ log_i$. From this follows

$$eval\big(neg(\neg \textit{con\_conj}, \chi_i)\big)\big(db_{i-1}^S\big) = \textit{true}. \quad (30)$$

   From $db_{i-1}^S \ \textit{model\_of}\ log_i$ and $\neg\chi_i \in log_i$ we have that $eval^*(\chi_i)(db_{i-1}^S) = \neg\chi_i$, enabling the usage of Lemma 1 on (30):

$$eval\big(\neg \textit{con\_conj}\big)\big((db_{i-1}^S \setminus \{\neg\chi_i\}) \cup \{\chi_i\}\big) = \textit{true} \quad (31)$$

Applying the definition of *eval* we get

$$eval\big(con\_conj\big)\big((db_{i-1}^S \setminus \{\neg\chi_i\}) \cup \{\chi_i\}\big) = false \tag{32}$$

This is equation (25), which the preceding arguments showed to hold under db^S. We also have equation (26) to be true since it only depends on the log of round $i-1$. It thus follows that case 3 is triggered under db^S, too.

4. Finally it remains to be shown that case 3 isn't triggered under db^S if db triggered case 4. From case 4 under db and Lemma 3 we have

$$db_i^S \ model\_of \ log_i = neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \tag{33}$$

This gives us $eval^*(con\_conj)(db_i^S) = true$ resulting in case 3 not being triggered due to the falseness of equation (25).

7 Reversibility

A user expects to be able to undo an update he issued. This is in fact one of the two requisites Bancilhon and Spyratos require for a set of view updates to be called complete (see [BS81]). We therefore demand and prove:

Theorem 2. *For any instance of Controlled Query Evaluation with view updates as defined by Def. 4 and Def. 5 and any $\chi \in LIT$ it is true that an operation $update(\chi)$ at time $i-1$ can be successfully undone at time i by the operation $update(\neg\chi)$.*

To prove this theorem, we need the following lemmas, which we state here without a justification, due to the lack of space.

Lemma 4. *For all sets of formulas $Q, P \subseteq \mathcal{P}(FORMULA)$ and all literals $\chi \in LIT$ we have* $P \models Q \Leftrightarrow neg(P, \chi) \models neg(Q, \chi)$.

Lemma 5. *If at the point in time $i-1$ the update $update(\chi)$ has been performed successfully, i.e., case 4 of the secure view update algorithm applies, then the following property holds:* $neg(log_{i-1}, \chi) \cup \{\neg\chi\} \cup constraints \not\models pot\_sec\_disj$

Proof (of Theorem 2). To outline the proof of the theorem, we assume that the update χ has been successfully completed at the point in time $i-1$. According to case 4 of the algorithm, we then have

$$log_{i-1} = neg(log_{i-2}, \chi) \cup \{\chi\} \cup constraints \tag{34}$$

We then have to verify that an $update(\chi_i)$ that has the form $update(\neg\chi)$ will be completed successfully, too, i.e., the cases 1 to 3 do not apply.

Case 1. We show that neither (22) nor (23) are satisfied. By (34), we have $\chi \in log_{i-1}$, so $log_{i-1} \cup \{\neg\chi\}$ is inconsistent and thus $log_{i-1} \cup \{\neg\chi\} \models pot\_sec\_disj$. This is a contradiction to (22). By Lemma 2, we have $log_{i-1} \not\models pot\_sec\_disj$, and as $\chi \in log_{i-1}$ also $log_{i-1} \cup \{\chi\} \not\models pot\_sec\_disj$, which is a contradiction to (23).

Case 2. This case cannot occur owing to Lemma 5.

Case 3. Consider $db' := (db_{i-1} \setminus \{\chi\}) \cup \{\neg\chi\}$, i.e., that instance that would become db_i, if db' satisfied the consistency constraints. First, assume $\neg\chi \in db_{i-2}$. Then $db' = db_{i-2}$, and since $eval(con\_conj)(db_{i-2}) = true$ we conclude that db', i.e., the potential db_i, will be consistent after the update $\neg\chi$.

Second, assume $\chi \in db_{i-2}$. If $eval(con\_conj)(db') = true$, the denial condition (25) does not hold. Otherwise, if $eval(con\_conj)(db') = false$, we will derive a contradiction. Under the assumptions made, we would have:

$$log_{i-1} \cup \{\chi\} \cup \{neg(\neg con\_conj, \chi)\} \not\models pot\_sec\_disj$$

Then, stepwise applying Lemma 4, the definition of *neg*, equation (34) and the idempotence of *neg*, we would finally get the result:

$$log_{i-2} \cup \{\neg\chi\} \cup neg(constraints, \chi) \cup \{\neg\chi\} \cup \{\neg con\_conj\} \not\models neg(pot\_sec\_disj, \chi)$$

This result cannot hold, since the premise is inconsistent, owing to $constraints \subseteq log_{i-2}$ on the one hand and $\neg con\_conj$ occurring on the other hand.

8 Conclusion

Data manipulation comprises queries and *updates* under preservation of *constraints*, and both kinds of operation might enable a user to infer information to be kept secret. In this paper, we extend previous insight about ensuring inference-free query answers to an original proposal for processing update requests in an *inference-free* way. The extension applies to both the formal *specification* of the confidentiality requirement and the *enforcement* mechanism.

Basically, the adapted requirement expresses that, from the user's point of view, the dynamically evolving actual instances of the information system are *indistinguishable* from alternative instances in which the protected information does not hold. Roughly summarised, for any single operation, whether a query or an update, the enforcement maintains a global *invariant*, which states that the current knowledge of the user does not imply the disjunction of the potential secrets. Under additional precautions, the local assurance suffices to guarantee the global goal of indistinguishability. Furthermore, our proposal complies to the basic rules of *acceptability* and *reversibility*, required for traditional view update mechanisms. The proposal is also in accordance with *polyinstantiation*, which is used in multilevel secure systems as an inevitable feature to resolve conflicts between preservation of constraints and hiding of confidential information.

In this paper, we have dealt with update requests issued by a user; in complementary work, we are studying updates triggered by an administrator, which leads to the problem of inference-free "view refreshments". Additionally, we are combining both kinds of updates. Interestingly, the complementary work strongly suggests to consider also transactions rather than only elementary updates.

There are many further challenging problems, including an extension to first-order logic, information systems permitting open queries, and an exploration of

explicit refusals on a requested updates while avoiding the known threats of meta-inferences, as well as suitable combinations of lying and refusal. An option to avoid distortions by lying could be worthwhile for applications where returning unreliable information is not acceptable (see [BB01,BB04a,BB04b]). Moreover, all investigations could be generalised to incomplete information systems.

References

[BS81] Bancilhon, F., Spyratos, N., *Update semantics of relational views*, ACM Trans. Database Syst. 6 (1981), no. 4, 557–575.

[BB01] Biskup, J., Bonatti, P.A., *Lying versus refusal for known potential secrets*, Data Knowl. Eng. 38 (2001), no. 2, 199–222.

[BB04a] Biskup, J., Bonatti, P.A., *Controlled query evaluation for enforcing confidentiality in complete information systems*, Int. J. Inf. Sec. 3 (2004), 14–27.

[BB04b] Biskup, J., Bonatti, P.A., *Controlled query evaluation for known policies by combining lying and refusal*, Ann. Math. Art. Intell. 40 (2004), 37–62.

[BB07] Biskup, J., Bonatti, P.A., *Controlled query evaluation with open queries for a decidable relational submodel*, Ann. Math. Art. Intell. 50 (2007), 39–77.

[BW08] Biskup, J., Weibert. T., *Keeping secrets in incomplete databases*, Int. J. Inf. Sec. 7 (2008), 199–217.

[BP06] Bohannon, A., Pierce, B.C., Vaughan, J.A., *Relational lenses: a language for updatable views*. In: PODS 06, ACM, 338–347.

[BK95] Bonatti, P.A., Kraus, S., Subrahmanian, V.S., *Foundations of secure deductive databases*, IEEE Trans. Knowledge and Data Engineering 7 (1995), no. 3, 406–422.

[CG99] Cuppens, F., Gabillon, A., *Logical foundation of multilevel databases*, Data Knowl. Eng. 29 (1999), 259–291.

[CG01] Cuppens, F., Gabillon, A., *Cover story management*, Data Knowl. Eng. 37 (2001), 177–201.

[DB82] Dayal, U., Bernstein, P.A., *On correct translation of update operations on relational views*, ACM Trans. Database Systems 8 (1982), 381–416.

[DA87] Denning, D.E., Akl, S., Heckman, M., Lunt, T., Morgenstern, M., Neumann, P., Schell, R., *Views for multilevel database security*, IEEE Trans. Software Eng. 13 (1987), no. 2, 129–140.

[FJ02] Farkas, C., Jajodia, S., *The inference problem: a survey*, SIGKDD Explor. Newsl. 4 (2002), no. 2, 6–11.

[He04] Hegner, S.J., *An order-based theory of updates for relational views*, Ann. Math. Art. Intell. 40 (2004), 63–125.

[JS91] Jajodia, S., Sandhu, R.S., *Towards a multilevel secure relational data model*. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, May 1991, 50–59.

[La90] Langerak, R., *View updates in relational databases with an independent scheme*, ACM Trans. Database Systems 15 (1990), 40–66.

[LD90] Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R., *The SeaView security model*, IEEE Trans. Software Eng. 16 (1990), no. 6, 593–607.

[SJ92] Sandhu, R.S., Jajodia, S., *Polyinstantiation for cover stories*. In: Proc. 2nd European Symp. Res. Computer Security, ESORICS 92, Lecture Notes in Computer Science 648, Springer, Berlin etc., 1992, 307–328.

[SJ83] Sicherman, G.L., de Jonge, W., van de Riet, R.P., *Answering queries without revealing secrets*, ACM Trans. Database Systems 8 (1983), no. 1, 41–59.

[WS94] Winslett, M, Smith, K., Qian, X., *Formal query languages for secure relational databases*, ACM Trans. Database Systems 19 (1994), no. 4, 626–662.