Requirements and Protocols for Inference-Proof Interactions in Information Systems

Joachim Biskup, Christian Gogolin, Jens Seiler, and Torben Weibert

Fakultät für Informatik, Technische Universität Dortmund, D-44221 Dortmund, Germany
biskup@ls6.informatik.uni-dortmund.de

Abstract. Inference control aims at disabling a participant to gain a piece of information to be kept confidential. Considering a provider-client architecture for information systems, we present transaction-based protocols for provider-client interactions and prove that the incorporated inference control performed by the provider is effective indeed. The interactions include the provider answering a client's query and processing update requests of two forms. Such a request is either initiated by the provider and thus possibly to be forwarded to clients in order to refresh their views, or initiated by a client according to his view and thus to be translated to the repository maintained by the provider.

1 Introduction and Survey

A service *provider* maintaining an application of an *information system* supports his *clients* to share and communicate *information*. Basically, sharing information is accomplished by keeping available *(semi-)structured data* in a repository in a persistent and *integrity enforcing* way, and communicating information is the result of various interactions between the provider and his clients, including the provider *answering a client's query*, the provider *processing a client's update request*, and the provider *informing a client about an update* performed. Accordingly, the service provider acts as a mediator between the clients, and there are no direct interactions between the clients. In this work, we study a particular version of this general scenario including a particular security aspect, as outlined in the following.

Regarding *availability*, different clients might have different information needs and, complementarily, regarding *confidentiality*, the provider might not want to allow each individual client to share all the information. According to the mediation architecture, any restriction of the *information flow* between two clients has to be enforced by controlling the provider-client interactions.

In order to restrict information flows, at the site of the mediating provider some *control component* has to decide about whether and to which extent – or with which modifications – a requested interaction should be actually executed. Any such decision must be based on two complementary policies that are suitably declared in advance: For each client, a *confidentiality policy* states which information that client should never be able to gain, and an *availability policy* states which information should be supplied to that client on demand. Clearly, the two policies must be *conflict-free*, i.e., no piece of information is both prohibited and permitted, and the two policies should be *complete*, i.e.,

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 285–302, 2009.

for each called interaction and the pieces of information involved, a definitive decision can be obtained.

Unfortunately, regulating plain *access to data* is not sufficient to control the *gain of information*. Additionally, the control component has to take into consideration the potential *inferences* a client can derive from observing any aspect of the *system's behavior over the time* [19,23,10]. This behavior includes query responses, notifications of enforcing integrity constraints and control decisions. Moreover, the client's inferences could additionally exploit *a priori knowledge*, which might range from public knowledge, like the schema with the *integrity constraints* declared for the information system, to the client's specific experience. Accordingly, the control component must be based on an appropriate *assumption* about a client's a priori knowledge.

Within the context sketched above, we deal with the problem of policy-based *inference control* of interactions in an information system in three ways:

- We specify the requirements in detail, including a formal specification of the goal of *inference-proofness* in terms of *indistinguishability*.
- Exemplarily considering a specific instantiation of the given context, we propose *control protocols* for the basic interactions of querying and updating.
- We formally outline a *verification* of these control protocols w.r.t. the requirements.

We substantially extend previous work on controlled query evaluation [39,14,3,4,5,6,8], which assumes a static information system, never updated after its initialization. Moreover, our results identify inference control as an important feature of view updating and view refreshing [2,18,30,27,13], and they complement the rich literature on mandatory control of information systems with polyinstantiation [20,31,28,37,16,17,41] by investigating a discretionary, policy-based control mode. The main general insight supplied and the most important results presented can be summarized as follows:

- A provider can effectively control the basic interactions of *querying* and *updating* including *enforcing integrity constraints* in an inference-proof way, i.e., such that any forbidden information gain by his clients is provably impossible.
- Applying an inference-proof protocol for *view refreshing*, a provider can support a client who maintains a local view by recalling all query answers and needs to get informed about updates.
- Applying an inference-proof protocol for *view updating*, a provider can support a client who both issues queries and modifies data held by the provider.
- Both protocols are designed to handle *transactions*, i.e., atomically treated sequences of update requests, and thus inference-proof interactions are compatible with advanced enforcement of integrity constraints.

The remainder of this paper is structured as follows. In Section 2, we further describe the context already sketched and explain the inference problems involved in some more detail. In Section 3, we introduce a formal model for our investigations, present the requirements and recall a known result on controlled query evaluation. In Section 4, we propose a protocol for processing provider updates requests and view refreshing, and in Section 5 a protocol for processing view update requests. In the respective sections, both protocols are proved to satisfy the requirements. Finally, in Section 6, we discuss related work, comment on the achievements and suggest some lines of further research.

2 Scenario and Problem Statement

We distinguish between (syntactically given) *data* and the (semantically interpreted) *information* denoted by such data. Given a meaning of information, we can also speak about *logical implications* between pieces of information. To keep a piece of information *confidential* to a client, it is necessary that this piece is not logically implied by the information available to that client. Accordingly, given a *confidentiality policy* as a set of sentences, a provider has to enforce an *invariant* expressing that the current information of a client does not logically imply that any of those sentences holds. However, we consider it harmless that a client obtains the information that such a sentence does *not* hold. Seeing the primary goal of an information system to support the sharing of information, we treat confidentiality requirements as an exception from the rule of guaranteeing availability as far as possible. Accordingly, whereas we specify the confidentiality policy extensionally by explicitly enumerating the respective sentences (as the "exceptions"), we express the complementary *availability policy* intensionally just by requiring that the holding of any other information should be correctly communicated unless a distortion is actually needed for preventing a violation of confidentiality.

At the beginning, the provider has to postulate the pertinent invariant as a *precondition* about the information available to that client. In general, the *a priori knowledge* of a client includes the *integrity constraints* of the *schema*. Before returning an answer to any *query* issued by that client, the provider has to censor the correct answer whether it would violate the invariant given the current information available to the client. Thus, maintaining a *log file* for each of the clients, the provider has to consider both the client's (postulated) a priori knowledge and all the information the client obtained from previous interactions since the beginning. If the provider detects that a violation of the invariant would arise, basically, he has two options to react: Either he notifies the client that he *refuses* to deal with the query or, without notification of course, he returns an answer where the correct truth value is switched, a *lie* for short. In this paper, we exemplarily deal with lies; thus, in order to avoid running into a "hopeless situation" in the future, the invariant must be strengthened such that the client's current information of a client does not logically imply that the disjunction of of all sentences to be kept confidential holds. The overall approach leads to a behavior of "last minute distortions" and, consequently, the dependence of the returned answers from the submission sequence.

The basic arguments regarding answers to queries also apply to any reaction that a provider shows to a client in whatever kind of interaction. In this paper, we will study two kinds of *update* processing, aiming to identify sufficient conditions to block any forbidden gain of information. The central issue of any update processing is maintaining the *integrity constraints* declared: Inductively assuming that the integrity constraints are valid for the current instance, after completely processing an update request, the integrity constraints should be valid again for the new instance. If the update request is compatible with the integrity constraints, we actually get a modified instance; otherwise, in case of incompatibility, the current instance is left unchanged. In both cases, the requester is notified accordingly. Similar to answers to queries, such a notification conveys information, and thus it has to be controlled regarding options for forbidden inferences.

Notifying an *accepted* update request needs care. For example, we let the client request to set the truth value of the sentence "Mr X suffers from aids" to *true*, while we consider the sentence "Mr X suffers from aids or Mr X suffers from cancer" as an integrity constraint. If the provider notifies the client that the truth value has been *changed* indeed, then the client receives the information that previously the truth value of the sentence "Mr X suffers from aids" was *false* and thus, according to the constraint, "Mr X suffers from cancer" must have been and still is *true*. Hence, this update request partially includes the query whether "Mr X suffers from cancer" as a side effect. Another example indicates that notifying a *rejected* update might be crucial, too. Again, we let the client request to set the truth value of the sentence "Mr X suffers from aids" to *true*, but we now consider the sentence "Mr X does not suffer from aids or Mr X does not suffer from cancer" as integrity constraint. If the provider notifies the client that the request failed due to a violation of the integrity constraint, then the client receives the information that "Mr X suffers from cancer" must have been and still is *true*. Hence, this update request again partially includes a query, and thus must be treated accordingly.

In a first kind of processing updates, the *requesting agent* is the *provider* himself. If the update succeeds and the new instance differs from the previous one, then, in principle, the provider should inform all his clients accordingly. For, in our context, the clients are supposed to recall all previously received information and to consistently combine the accumulated knowledge into a local view for their respective tasks. However, an unobserved update could make a local view useless and thus threatens availability. Hence, once the instance has actually been modified, the provider has to *refresh* all local views, which in our context means to reevaluate the sequence of queries previously submitted by a client and to forward the new answers to that client. Since each single answer depends on the set of answers previously returned, a reevaluation after a succeeded update might cause subtle inference problems. In particular, a client could try to gain hidden information from comparing the original answers with the refreshed ones.

In the second kind of processing updates we study in this paper, the *requesting agent* is a *client*. For this kind, the client is supposed to possess a local *view* on the actual (but hidden) instance (which is stored at the site of the provider), and his update request is seen as referring to his local view (which might contain lies returned in previous interactions). Accordingly, the provider handles the request similarly to a classical *view update*, namely by translating the requested update of the view into an actual update of the full instance, as far as possible. Moreover, the provider has to send notifications about the success or failure of enforcing integrity constraints to the requesting client. As far as this client is confined by inference control, again the provider has to ensure that the notifications are inference-proof.

Given sophisticated integrity constraints, we sometimes cannot modify a current instance stepwise by individually treating the information regarding single sentences; rather, we have to process a whole sequence of modifications in an atomic way as a *transaction*, where the *constraints* must be valid after considering the full sequence but may be violated in between. A similar observation applies to *notifications* and *refreshments*: Sometimes, such messages regarding individual sentences would result in a forbidden gain of information but the message about the full transaction will turn out to be harmless.

3 Formal Model and Confidentiality Requirements

We employ a logic-oriented approach to information systems [1]. We only consider *complete, propositional* information systems (leaving generalizations to incomplete information systems [7,8] or first-order logic [6,9,11] for a future elaboration). We assume a vocabulary of propositional *atoms*, from which we can construct propositional *sentences* using the connectives of *negation* and *disjunction* (and derived connectives). A *literal* is either an atom or a negated atom. The *schema* of an information system is given by the vocabulary and the *integrity constraints*, expressed as a finite set *con* of sentences. An *instance db* is a set of literals: For each atom α of the vocabulary, either the atom α itself or the negated atom $\neg\alpha$ is an element. Given the vocabulary, it suffices to explicitly specify only the atoms. An instance *db* defines a *truth-value assignment* to propositional atoms by making each atom $\alpha \in db$ *true* and all the remaining atoms *false*. Such an assignment is inductively extended to arbitrary sentences Φ; $eval(\Phi)(db)$ denotes the truth value assigned to Φ by *db*. We require that an instance *db* satisfies the integrity constraints *con*, i.e., $eval(con\_conj)(db) = true$ for $con\_conj := \bigwedge_{\phi \in con} \phi$. The notion of logical *implication* between (sets of) sentences is designated by $\models$.

A *query* request $que(\Phi)$, contains any sentence Φ of the underlying propositional logic (leaving a generalization to open queries [6] for future work). The *correct answer* to the query Φ under an instance *db* is the pertinent truth value $eval(\Phi)(db)$; we alternatively express the correct answer by $eval^*(\Phi)(db)$ that denotes either Φ or $\neg\Phi$ in a straightforward way. Regarding an *update* request, we focus on changing the truth-values of atoms, in order to avoid ambiguity problems [2] (leaving extensions to more sophisticated cases [2,18,30,27,13] for further research). A request contains one or more literals, assumed to refer to pairwise different atoms, that should be set to *true*, i.e., become an element of the updated instance. An update request *succeeds* for a given instance db_1, if adding the specified literal(s) and removing its (their) negation(s) transforms db_1 into db_2 that satisfies the constraints again; otherwise the request *fails*.

Definition 1 (interaction sequences). *An* interaction sequence $Q := \langle\, \Theta_1, \Theta_2, \ldots, \Theta_i, \ldots, \Theta_k \,\rangle$ *is composed of query requests and update requests submitted by the provider and the clients as follows:*

$$\Theta_i := \begin{cases} C_i : que(\Phi_i) & a \text{ query, } submitted\ by\ a\ client\ C_i,\ or \\ P : pup(\chi_i) & an \text{ elementary provider update } with \\ & a\ single\ literal,\ or \\ P : ptr(\langle \chi_{i,1}, \ldots, \chi_{i,l_i} \rangle) & a \text{ provider update transaction } with \\ & a\ set\ of\ literals\ from\ different\ atoms,\ or \\ C_i : vup(\chi_i) & an \text{ elementary view update } with \\ & a\ single\ literal,\ submitted\ by\ a\ client\ C_i,\ or \\ C_i : vtr(\langle \chi_{i,1}, \ldots, \chi_{i,l_i} \rangle) & a \text{ view update transaction } with\ a\ set\ of\ literals \\ & from\ different\ atoms,\ submitted\ by\ a\ client\ C_i. \end{cases} \quad (1)$$

Though not reflected by the notations used in the definition, an execution of an update request might produce messages for *all* clients for distributing refreshments.

To confine a client *C*, the provider declares a *client confidentiality policy* as a finite set $pot\_sec[C]$ of propositional sentences, called potential secrets, indicating that they are

not necessarily true in a current instance (leaving alternative, but not always applicable policies containing complementary sentences ("secrecies") [39,4] for further elaboration). The client involved is supposed to *know* this declaration (leaving the weaker assumption of non-awareness [39,4], which might cause less distortions, for future work). *SEC* denotes the collection of all client policies $pot\_sec[C]$. In order to prevent the client C from ever inferring that any sentence $\Psi \in pot\_sec[C]$ actually holds, the approach of *lying* [14,3,8] has to protect not only the individual potential secrets but, in fact, the disjunction of all potential secrets $pot\_sec\_disj[C] := \bigvee_{\Psi \in pot\_sec[C]} \Psi$. This requirement for lying reflects the need to avoid "hopeless situations" of the following kind: While already knowing the disjunction of some potential secrets Ψ_i, a client successively queries those sentences and receives lied answers $\neg\Psi_i$, which would lead to an inconsistent log file. (We leave protocols for the approach of refusals and for a combination of lying and refusals [39,3,4,5,6,8] for future research.)

For each client C, the provider maintains a *client log* $log[C]$ for keeping the (postulated) a priori knowledge of that client and the reactions, including answers to queries, returned to him during previous interactions. Without loss of generality, we always assume that the provider communicates the initial value $log[C]_0$ of the log file to the client C at the time of registration. Basically, $log[C]$ is just a set of propositional sentences (whereas in future work for incomplete information systems we have to employ modal logic [7,8]). However, for some purposes, the provider might have to recall some further information, in particular the order in which the client has issued his queries. For simplicity, and by abuse of notations, we refrain from explicitly denoting such additional information in the generic definition given below. Later on, however, we will add more details as particularly needed. *LOG* denotes the collection of all client logs $log[C]$.

Definition 2 (controlled execution). *Let be given a finite set con of sentences as integrity constraints, a current instance db_{i-1}, and for each client C a finite set $pot\_sec[C]$ of sentences as a confidentiality policy, collected by SEC, and a finite client log $log[C]_{i-1}$ with $log[C]_{i-1} \supseteq con$, collected by LOG_{i-1}.*

Then a function $cexec(con, db_{i-1}, SEC, LOG_{i-1}, \Theta_i)$ defines a controlled execution *of an interaction Θ_i by the triple (REA_i, LOG_i, db_i), where*

– REA_i are the collected reactions (possibly) returned to the provider and the clients;

– LOG_i are the collected new client logs; and

– db_i is the new instance produced (satisfying con).

Furthermore, for an initial instance db_0 and initial collected client logs LOG_0 this function is inductively extended to any interaction sequence $Q := \langle\, \Theta_1, \Theta_2, \ldots, \Theta_i, \ldots, \Theta_k \,\rangle$ by applying it stepwise in a straightforward way:

$cexec(con, db_0, SEC, LOG_0, Q)$

$= \big\langle (REA_1, LOG_1, db_1), \ldots, (REA_i, LOG_i, db_i), \ldots, (REA_k, LOG_k, db_k) \big\rangle$

The formal definition of the confidentiality requirement we want to achieve by a controlled execution is expressed in terms of the *indistinguishability* – from the point of view of some client C – of the actual sequence of instances from an alternative sequence whose instances do not satisfy any potential secret – as declared for that client, together with the indistinguishability of the corresponding interaction sequences. To keep the notation simple, we give this definition only in the form tailored for the lying

approach. We also emphasize that we will give a definition that is parameterized with the expressive means of the scenario considered, the clients are assumed to be aware of.

Definition 3 (confidentiality). *Let Int be a subcollection of the interactions in the sense of Def. 1, Con a class of sentences for expressing integrity constraints, Pol a class of sentences for expressing confidentiality policies and Know a class of sentences for expressing further a priori knowledge.* [1] *A controlled execution function cexec* preserves confidentiality *(w.r.t. Int, Con, Pol and Know) iff*
for all sets of integrity constraints con $\subseteq$ *Con, for all initial instances* db_0 *satisfying con, for all collections of confidentiality policies SEC expressed with sentences in Pol, for all collections of initial client logs* LOG_0 *such that for each client C,* $con \subseteq log[C]_0$ *and* $log[C]_0 \setminus con$ *is expressed with sentences in Know and* $log[C]_0 \not\models pot\_sec\_disj[C]$*, for all interaction sequences Q over the underlying subcollection Int, for each client C: there exists an alternative instance* db_0^C *satisfying con and there exists an alternative interaction sequence* Q^C *over Int such that from the point of view of C, as defined by the projection* υ^C *of a sequence of triples* (REA_i, LOG_i, db_i) *to the C-visible parts, in particular the reactions* $ans[C]_i$*, the following two properties hold:*
1. Q with db_0 *and* Q^C *with* db_0^C *produce the same sequence of reactions, i.e.,*

$$\upsilon^C\big(cexec(con, db_0, SEC, LOG_0, Q)\big) = \upsilon^C\big(cexec(con, db_0^C, SEC, LOG_0, Q^C)\big) \quad (2)$$

2. db_0^C *and all* db_i^C *as well do not contain any potential secret* Ψ *in* $pot\_sec[C]$*, i.e.,*

$$eval^*(\Psi)(db_i^C) = \neg\Psi\,, \text{ for all } i = 0, \ldots \quad (3)$$

The general scenario simplifies considerably if we consider a *fixed* instance db_{i-1} and allow only *queries* by clients. Assumed not to be colluding, the clients can then be treated completely separately (ignoring covert channels or related unwanted effects). Moreover, since answers do not age, no refreshments are needed. For this simplified scenario, we can restate a mechanism of "controlled query evaluation" using lies, presented and proved to preserve confidentiality in previous work [14,3,4], as follows.

Protocol 1 (query answering) [2]
client: submit a query request $C_i : que(\Phi_i)$ to the provider.
provider:

1. check whether adding the correct truth $eval^*(\Phi_i)(db_{i-1})$ to the log file $log[C_i]_{i-1}$ maintained by the provider would preserve the invariant derived from the confidentiality policy $pot\_sec[C_i]$, i.e.,

$$log[C_i]_{i-1} \cup \{eval^*(\Phi_i)(db_{i-1})\} \not\models pot\_sec\_disj[C_i]; \quad (4)$$

[1] To denote one sort of item, we select an appropriate identifier. To distinguish to which client C an item refers, we qualify the identifier by a suffix of the form "$[C]$". To indicate the state of an item at a point in time i, we append a subscript "$_i$" to the identifier. Finally, if for a client C a possible alternative "view" is considered, we append a superscript "^{C}".

[2] For saving space, we present all protocols by mixing informal explanations and formal specifications. Note that answers to the provider are not subject to confidentiality constraints. At some places, an answer to a client is explicitly shown only in an informal way; then the formal version is understood to be implicitly specified by the (non)modification of the log file.

2. `if` (4) holds, `then` return the correct truth value $eval^*(\Phi_i)(db_{i-1})$ to C_i
   `else` return the negation $\neg eval^*(\Phi_i)(db_{i-1})$, i.e. a lie, (as justified by a basic lemma showing that in the negative case the lie does preserve the invariant);
   insert the sentence returned into C_i's log.

We concisely summarize the provider's part of the protocols more formally by:

$$\begin{aligned} ans[C_i]_i := \ & \text{if } log[C_i]_{i-1} \cup \{eval^*(\Phi_i)(db_{i-1})\} \not\models pot\_sec\_disj[C_i] \\ & \text{then } eval^*(\Phi_i)(db_{i-1}) \text{ else } \neg eval^*(\Phi_i)(db_{i-1}) \\ log[C_i]_i := \ & log[C_i]_{i-1} \cup \{ans[C_i]_i\} \end{aligned} \tag{5}$$

Alternatively, we might see a pair $(db_{i-1}, log[C_i]_{i-1})$ as a kind of *polyinstantiated instance*: Given the request $C_i : que(\Phi_i)$, the provider first inspects whether the second, potentially distorted (or "polyinstantiated") part $log[C_i]_{i-1}$ already entails an answer; only otherwise, the first, "real" part is employed to dynamically check the correct answer for eligibility, and if this is not the case, the query sentence is "polyinstantiated" by inserting the negation of the correct answer into the second part.

The definition of controlled execution and the protocol of query answering indicate that, in general, achieving inference-proofness require us to accept a high computational overhead, in particular by keeping log files and solving implication problems. However, under some reasonable restrictions substantial optimizations for query answering are possible [9,11] (leaving extensions for update processing for future research).

4 Processing Provider Update Requests and View Refreshing

In this section, we originally introduce inference-proof view refreshments and study their coordination with query answering. More specifically, whenever the provider successfully modifies the instance, a client might be left with an aged view, i.e., for a query previously submitted by him the answer actually obtained on the basis of the instance at the point of time of the submission differs from the answer on the basis of the modified instance. Thus, after a successful modification of the instance, the provider should always refresh the views generated by his previous answers (or other reactions). We will present and analyze two protocols to meet this requirement.

The first protocol deals with update *transactions*, and thus a client, receiving a refreshment notification and then reasoning about the (hidden) actual modification, has to consider the possibility that the real cause has been a *sequence of updates*. Basically, this protocol determines refreshments by a controlled reevaluation of the pertinent queries. The second protocol deals with *elementary* updates, and thus, from a notified client's point of view, a real cause of a notification is restricted to a *single update*. Under this assumption and the further restriction that only the subclass of *literals* (rather than all sentences) is permitted to be used for queries, constraints, a priori knowledge and confidentiality policies, this protocol does not need to perform complete reevaluations; instead, basically, it suffices to just inspect the modified literal of the update request.

The two protocols indicate a tradeoff between expressiveness and efficiency: If we permit unrestricted declarations and interactions, we are faced with the need to perform computationally expensive reevaluations; however, under the restrictions mentioned above, inference-proof view refreshing can be performed highly efficiently.

Protocol 2 (provider update transaction processing with refreshments)
provider: submit a provider update transaction request $P : ptr(\langle \chi_{i,1}, \ldots, \chi_{i,l_i} \rangle)$
(requesting to set each of the $\chi_{i,j}$ to *true*), where the argument sequence consists of literals containing pairwise different atoms; and let $\Delta_i := \{\chi_{i,1}, \ldots, \chi_{i,l_i}\}$.

1. remove all literals $\chi_{i,j}$ from the request Δ_i that are already valid in db_{i-1} and notify the provider;
   `if` the update request is now empty
   `then` do not modify the instance and notify the provider, i.e.,
   - $db_i := db_{i-1}$, for all clients C: $log[C]_i := log[C]_{i-1}$ and $ans[C]_i := \varepsilon$
   - $ans[P]_i :=$ "The requested update is already contained in the database"
2. `else if` the requested update would be incompatible with the constraints, i.e.,

$$eval\big(con\_conj\big)\big((db_{i-1} \setminus \{\neg\chi_{i,j} | \chi_{i,j} \in \Delta_i\}) \cup \Delta_i\big) = false \tag{6}$$

   `then` do not modify the instance and notify the provider, i.e.,
   - $db_i := db_{i-1}$, for all clients C: $log[C]_i := log[C]_{i-1}$ and $ans[C]_i := \varepsilon$
   - $ans[P]_i :=$ "Update of Δ_i inconsistent with integrity"
3. `else` accept the requested update, modify the instance and notify the provider, i.e.,
   - $db_i := (db_{i-1} \setminus \{\neg\chi_{i,j} | \chi_{i,j} \in \Delta_i\}) \cup \Delta_i$
   - $ans[P]_i :=$ "Update of Δ_i successful"

   and,
   for all clients C, perform the following *refreshment subprotocol* for $j_0 := 0$ and the subsequence $Q[C]_{j_0} := \langle\ \Theta_{j_1}, \ldots, \Theta_{j_{k_C}}\ \rangle$ of query requests $C : que(\Phi_{j_l})$ submitted by C previously:
   - using Protocol 1, reevaluate the subsequence using the new instance db_i and the client log $log[C]_{j_0}$ and thereby producing a new current client log[3] $log[C]_i$
   - determine the deviating answers $refresh[C]_i := log[C]_i \setminus log[C]_{i-1}$
   - if there are deviations, notify the client C, i.e.,
     $ans[C]_i :=$ if $refresh[C]_i \neq \emptyset$ then $refresh[C]_i$ else ε

Example 1. We consider a vocabulary *schema* and, for the sake of simplicity, only one client C with confidentiality policy $pot\_sec[C]$ and initial log file $log[C]_0$, and an initial instance db_0 as follows: $schema := \{a,b,c,d,e,f,s_1,s_2,t_1,t_2\}$, $pot\_sec[C] := \{s_1, s_2, (t_1 \wedge t_2)\}$, $log[C]_0 := con := \{a \vee b \vee s_2\}$, $db_0 := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, t_2\}$. Table 1 exhibits an interaction sequence and the resulting effects.
As seen to be possible by the client C, an alternative instance is given by
$db_0^C := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, \neg t_2\}$
and an alternative interaction sequence by
$Q^C := \langle C : que((c \wedge d \wedge e \wedge f) \vee s_1), C : que(t_1), C : que(t_2), P : ptr(\langle \neg t_1, t_2, a, d \rangle), C : que(s_2) \rangle$.

[3] Using the parameter $j_0 := 0$, the refreshment subprotocol does not change any sentence of the initial log file. Seeing the integrity constraints *cons* as schema data, we have to keep them invariant. Seeing an update request to refer only to the instance, we obtain the option to introduce a separate control operation to modify the a priori knowledge in $log[C]_0 \setminus cons$, which we do not treat further in this paper. However, dealing with view updates, we will enable a client to modify the a priori knowledge.

Table 1. An interaction sequence and the resulting effects for Protocol 2

interaction	effect
$P : ptr(\langle \neg b, \neg e \rangle)$ invisible incompatibility	$db_1 := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, t_2\}$ $ans[C]_1 := \{\}$ $log[C]_1 := \{(a \vee b \vee s_2)\}$
$C : que((c \wedge d \wedge e \wedge f) \vee s_1)$ distorted answer	$db_2 := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, t_2\}$ $ans[C]_2 := \{\neg((c \wedge d \wedge e \wedge f) \vee s_1)\}$ $log[C]_2 := \{(a \vee b \vee s_2), \neg((c \wedge d \wedge e \wedge f) \vee s_1)\}$
$C : que(t_1)$ correct answer	$db_3 := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, t_2\}$ $ans[C]_3 := \{t_1\}$ $log[C]_3 := \{(a \vee b \vee s_2), \neg((c \wedge d \wedge e \wedge f) \vee s_1), t_1\}$
$C : que(t_2)$ distorted answer	$db_4 := \{\neg a, b, c, \neg d, e, f, \neg s_1, \neg s_2, t_1, t_2\}$ $ans[C]_4 := \{\neg t_2\}$ $log[C]_4 := \{(a \vee b \vee s_2), \neg((c \wedge d \wedge e \wedge f) \vee s_1), t_1, \neg t_2\}$
$P : ptr(\langle \neg t_1, s_1 \rangle)$ refreshment	$db_5 := \{\neg a, b, c, \neg d, e, f, s_1, \neg s_2, \neg t_1, t_2\}$ $ans[C]_5 := \{((c \wedge d \wedge e \wedge f) \vee s_1), \neg t_1, t_2\}$ $log[C]_5 := \{(a \vee b \vee s_2), ((c \wedge d \wedge e \wedge f) \vee s_1), \neg t_1, t_2\}$
$C : que(s_2)$ correct answer	$db_6 := \{\neg a, b, c, \neg d, e, f, s_1, \neg s_2, \neg t_1, t_2\}$ $ans[C]_6 := \{\neg s_2\}$ $log[C]_6 := \{(a \vee b \vee s_2), ((c \wedge d \wedge e \wedge f) \vee s_1), \neg t_1, t_2, \neg s_2\}$
$P : ptr(\langle s_2 \rangle)$ hidden update	$db_7 := \{\neg a, b, c, \neg d, e, f, s_1, s_2, \neg t_1, t_2\}$ $ans[C]_7 := \{\}$ $log[C]_7 := \{(a \vee b \vee s_2), ((c \wedge d \wedge e \wedge f) \vee s_1), \neg t_1, t_2, \neg s_2\}$

Theorem 1 (inference-proof provider update transactions with refreshments). *For Int being the subcollection of* queries and provider update transactions *in the sense of Def. 1 and Con, Pol and Know being the full class of all sentences, the controlled execution function that is based on Protocol 1 (queries) and Protocol 2 (provider update transactions)* preserves confidentiality *in the sense of Def. 3.*

Proof. We focus on one of the clients, say client C, and omit the qualification "[C]" for components of policies, reactions and log files related to C. As required by Def. 3, we start with a given general situation relevant for C, namely the integrity constraints *con*, the potential secrets *pot_sec* and the initial log file log_0 with $con \subseteq log[C]_0$ and $log_0 \not\models pot\_sec\_disj$, and the initial instance db_0 and the original sequence $Q = \langle\, \Theta_1, \Theta_2, \ldots, \Theta_i, \ldots, \Theta_k \,\rangle$ of interactions, w.o.l.g. of the kind $P : ptr(\langle \chi_{i,1}, \ldots, \chi_{i,l_i} \rangle)$ or $C : que(\Phi_i)$, which iteratively produce a sequence of instances db_i as defined by the protocols. We will construct an alternative instance db_0^C and an alternative interaction sequence $Q^C = \langle\, \Theta_1^C, \Theta_2^C, \ldots, \Theta_i^C, \ldots, \Theta_k^C \,\rangle$, which generates a sequence of alternative instances db_i^C. We will proceed inductively, for each interaction distinguishing its kind, and prove the properties described further in Def. 3. To elaborate on the induction, we even achieve the following stronger properties:

1. The subsequence $Q_{que} := \langle\, \Theta_{j_1}, \ldots, \Theta_{j_{k_C}} \,\rangle$ of Q formed by the query requests $C : que(\Phi_{j_i})$ is identical with the subsequence of Q^C formed by the query requests.
2. For all $i = 1, \ldots, k$, the original reaction ans_i and the alternative reaction ans_i^C returned to C are identical, i.e., $ans_i = ans_i^C$, and thus we also have that the original

and the alternative log files are identical, i.e., $log_i = log_i^C$. By definition, we also have $log_0 =: log_0^C$.

3. For all $i = 0,\ldots,k$, the alternative instance db_i^C satisfies *con*, but it does not satisfy *pot_sec_disj* and thus makes all potential secrets Ψ in *pot_sec false*, i.e., $eval^*(\Psi)(db_i^C) = \neg\Psi$.
4. Moreover, for all $i = 0,\ldots,k$, the alternative instance db_i^C satisfies all answers that would be returned if the subsequence Q_{que} of *all* submitted queries was evaluated by Protocol 1 for the potential secrets *pot_sec*, the initial log file log_0 and the instance db_i. This "look-back-and-ahead" property implies that the result of this fictitious evaluation, denoted by $log_{que,i}$ is identical with the corresponding result for the alternative instance db_i^C, denoted by $log_{que,i}^C$. Thus we have $log_{que,i} = log_{que,i}^C$.

The actual construction of the alternative instances is based on the enforced invariant expressing that a client's log file never implies *pot_sec_disj*: the alternative instances are taken as appropriate witnesses for such non-implications. The details of the construction and the verification of the claimed properties are omitted for the lack of space. □

Protocol 3 (elementary provider update processing with refreshments)
provider: submit an elementary provider update request $P : pup(\chi_i)$ to set χ_i to *true*
Essentially, same as Protocol 2, with some straightforward simplifiations and the following *optimized refreshment subprotocol*, performed for all clients C:

`if` either the client C is prohibited to learn the update performed
or the client is eligible but so far has "no belief" on $\neg\chi_i$, i.e.,
$\chi_i \models pot\_sec\_disj[C]$ or $\big(\chi_i \not\models pot\_sec\_disj[C]$ and $log[C]_{i-1} \not\models \neg\chi_i\big)$
`then` the update remains invisible to that client, i.e.,
– $ans[C]_i := \varepsilon$, and $log[C]_i := log[C]_{i-1}$
`else` notify that client and log the notification, i.e.,
– $ans[C]_i := \chi_i$
– $log[C]_i := (log_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}$

Example 2. We consider a vocabulary *schema* and, for simplicity, only one client C with confidentiality policy *pot_sec* and initial log file log_0, and an initial instance db_0 as follows: $schema := \{a,b,c,d,s_1,s_2\}$, $pot\_sec[C] := \{s_1,s_2\}$, $log[C]_0 := con := \{a\}$, $db_0 := \{a,b,\neg c,d,\neg s_1,s_2\}$. Table 2 exhibits an interaction sequence and the resulting effects, for which $db_0^C := \{a,b,\neg c,d,\neg s_1,\neg s_2\}$ is an alternative instance and $Q^C := \langle C : que(c), C : que(s_1), C : que(d), P : pup(c)\rangle$ is an alternative interaction sequence.

Theorem 2 (inference-proof elementary provider updates with optimized refreshments). *For Int being the subcollection of* queries with a literal and elementary provider updates *in the sense of Def. 1 and Con, Pol and Know being the class of literals, the controlled execution function that is based on Protocol 1 (queries) and Protocol 3 (elementary provider updates)* preserves confidentiality *in the sense of Def. 3.*

Proof. The omitted proof follows the inductive structure employed for Theorem 1. Alternatively, we could profit from that proof as follows. By definition, Protocol 3 is a specialization of Protocol 2 regarding Cases 1 and 2. Regarding Case 3, it is a specialization as well, since switching the truth value of a literal χ_i cannot affect the truth

Table 2. An interaction sequence and the resulting effects for Protocol 3

interaction	effect
$P : pup(b)$	$db_1 := \{a, b, \neg c, d, \neg s_1, s_2\}$
already contained update	$ans[C]_1 := \{\}$, $log[C]_1 := \{a\}$
$P : pup(\neg a)$	$db_2 := \{a, b, \neg c, d, \neg s_1, s_2\}$
invisible incompatibility	$ans[C]_2 := \{\}$, $log[C]_2 := \{a\}$
$C : que(c)$	$db_3 := \{a, b, \neg c, d, \neg s_1, s_2\}$
correct answer	$ans[C]_3 := \{\neg c\}$, $log[C]_3 := \{a, \neg c\}$
$C : que(s_1)$	$db_4 := \{a, b, \neg c, d, \neg s_1, s_2\}$
correct answer	$ans[C]_4 := \{\neg s_1\}$, $log[C]_4 := \{a, \neg c, \neg s_1\}$
$P : pup(s_1)$	$db_5 := \{a, b, \neg c, d, s_1, s_2\}$
hidden update	$ans[C]_5 := \{\}$, $log[C]_5 := \{a, \neg c, \neg s_1\}$
$C : que(d)$	$db_6 := \{a, b, \neg c, d, s_1, s_2\}$
correct answer	$ans[C]_6 := \{d\}$, $log[C]_6 := \{a, \neg c, d, \neg s_1\}$
$P : pup(c)$	$db_7 := \{a, b, c, d, s_1, s_2\}$
refreshment	$ans[C]_7 := \{c\}$, $log[C]_7 := \{a, c, d, \neg s_1\}$
$P : pup(\neg b)$	$db_8 := \{a, \neg b, c, d, s_1, s_2\}$
hidden update	$ans[C]_8 := \{\}$, $log[C]_8 := \{a, c, d, \neg s_1\}$

values of other literals. Theorem 1 then states that Protocol 3 preserves confidentiality if the client sees alternative *transactions* as "possible". Thus, it suffices to verify that an original *one-step* transaction always permits an alternative *one-step* transaction. □

5 Processing View Update Requests

We will now treat view updates in the context of our scenario, which includes queries, provider updates and transactions. Our main protocol is based on the following ideas.

First, we reconsider the protocol for a special case studied in [12]. For a restricted scenario of only one client and without provider updates, this protocol processes an elementary view update $C_i : vup(\chi_i)$. The protocol consists of four, subsequently considered steps, which represent four disjunct cases for the response to the client C_i. These cases capture the intuition that the client's request to set the truth value of the literal χ_i to *true* implicitly contains several queries that are answered by the provider's reactions. These implicit queries include whether χ_i is already *true* and whether the constraints would be valid after switching χ_i to *true*. Obviously, we have to identify all implicit queries and then control them as if they were explicitly submitted. The protocol for the general case, presented in this work, keeps the overall structure of the specialized one, but substantially extends it regarding refreshments for other clients and transactions.

We need the following tools: For a set Δ of sentences, $neg(\Delta)$ negates each sentence in Δ; for a sentence ϕ and a literal χ, $neg(\phi, \chi)$ replaces every occurrence of the atom specified by the literal χ in the formula ϕ by the negated atom; the latter function handles a set of sentences and a set of literals, respectively, element-wise. For example, $neg\big(\neg(a \wedge b) \vee \neg a, \neg a\big) = \neg(\neg a \wedge b) \vee a$; and we obtain a basic property:

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

Thus, we obtain the same results evaluating a sentence on an instance and evaluating the χ-negated formula on the instance created by negating the atom specified by χ.

Second, as in [12], we have to suitably resolve conflicts between integrity and confidentiality, well-known from polyinstantiation for mandatory access control. More specifically, on the one hand, a requested update could violate integrity but, on the other hand, a notification of this fact to the requesting client C_i would endanger confidentiality. Under polyinstantiation, the conflict is handled by keeping both the original value, classified to be employed for sufficiently cleared users, and an updated value, classified to be employed in particular for the requestor. In our discretionary approach, we will elaborate a similar solution: Roughly, the provider claims to perform the update but actually leaves the instance unmodified and only reflects the update in the log file of C_i. Thus, described alternatively in terms of seeing the pair $(db_{i-1}, log[C]_{i-1})$ as a kind of *polyinstantiated instance*, the update sentences are going to be "polyinstantiated".

Third, as a new feature, we have to add refreshments for the other clients $C \neq C_i$ and, for our broader context, to take care about all reactions received by such a client. Basically, there are three cases: answers to explicit queries, answers to implicit queries as discussed above, and refreshments of both kinds of answers. However, these cases are uniformly represented by the current log file $log[C]_i$ maintained by the provider. Thus, essentially, the provider has to refresh this log file, in general respecting the insertion sequence. Note that in general a client receiving a refreshment notification will not be able to distinguish whether the underlying update originates from another client or the provider (if the underlying update would be permitted for all participants involved).

Fourth, as an additional challenge, transactions raise the problem that some of the included requests might be harmful whereas others are not. We solve this problem by iteratively splitting the set Δ_i of all literals involved into two parts $Com\Delta_i$ and $Inc\Delta_i$, where $Com\Delta_i$ contains the literals identified to be *compatible* to the client's view and $Inc\Delta_i$ the *incompatible* ones.

Protocol 4 (view update transaction processing)
client: submit a view update transaction request $C_i : vtr(\langle \chi_{i,1}, \ldots, \chi_{i,l_i} \rangle)$ to the provider to set each of the $\chi_{i,j}$, containing pairwise different atoms, to *true*.
provider:

1. initialize the literal sets $Com\Delta_i$ and $Inc\Delta_i$, i.e., $Com\Delta_i := \emptyset$, $Inc\Delta_i := \emptyset$,
   and then iteratively inspect each literal for compatibility as follows:
   for $j = 1, \ldots, l_i$,
   `if` the request to update $\chi_{i,j}$ is compatible with the client's view (corresponding to the concept of "acceptability" in [2]; meaning that the request either needs not to be performed or should not be performed for the sake of confidentiality), i.e.,

$$[\, eval^*(\chi_{i,j})(db_{i-1}) = \chi_{i,j} \text{ and } log[C_i]_{i-1} \cup neg(Inc\Delta_i) \cup Com\Delta_i \cup \{\chi_{i,j}\} \not\models pot\_sec\_disj[C_i] \,] \text{ or} \tag{7}$$

$$[\, eval^*(\chi_{i,j})(db_{i-1}) = \neg\chi_{i,j} \text{ and } log[C_i]_{i-1} \cup neg(Inc\Delta_i) \cup Com\Delta_i \cup \{\neg\chi_{i,j}\} \models pot\_sec\_disj[C_i] \,] \tag{8}$$

   `then` $Com\Delta_i := Com\Delta_i \cup \{\chi_{i,j}\}$ `else` $Inc\Delta_i := Inc\Delta_i \cup \{\chi_{i,j}\}$;

`if` $Inc\Delta_i = \emptyset$ (i.e., all requests are seen as compatible)
`then` do not modify the instance, log the request like a query response, and notify the client C_i, i.e.,
- $db_i := db_{i-1}$
- $log[C_i]_i := log[C_i]_{i-1} \cup Com\Delta_i$
- $ans[C_i]_i :=$ "The requested update is already contained in the instance"

2. `else if` allowing the incompatible part would infer a secret or violate the constraints and this fact is known to the client C_i a priori, i.e.,

$$neg(log[C_i]_{i-1}, Inc\Delta_i) \cup Inc\Delta_i \cup Com\Delta_i \cup con \models pot\_sec\_disj[C_i] \tag{9}$$

`then` do not modify the instance, log the compatible and the negated incompatible parts like query responses, and notify the client C_i, i.e.,
- $db_i := db_{i-1}$
- $log[C_i]_i := log[C_i]_{i-1} \cup neg(Inc\Delta_i) \cup Com\Delta_i$
- $ans[C_i]_i :=$ "The part $Com\Delta_i$ of the requested update is already contained in the instance, and updating the part $Inc\Delta_i$ is inconsistent with secrets or integrity"

3. `else if` allowing the requested update would violate the constraints and this is unknown to the client a priori but not harmful, i.e,

$$eval\big(con\_conj\big)\big((db_{i-1} \setminus neg(Inc\Delta_i \cup Com\Delta_i)) \cup Inc\Delta_i \cup Com\Delta_i\big) = false \text{ and} \tag{10}$$

$$log[C_i]_{i-1} \cup neg(Inc\Delta_i) \cup Com\Delta_i \cup \big\{neg(\neg con\_conj, Inc\Delta_i)\big\} \not\models pot\_sec\_disj[C_i] \tag{11}$$

`then` do not modify the instance, log the negated incompatible part of the request, the compatible part of the request and a sentence expressing the incompatibility like query responses, and notify the client C_i, i.e,
- $db_i := db_{i-1}$
- $log[C_i]_i := log[C_i]_{i-1} \cup neg(Inc\Delta_i) \cup Com\Delta_i \cup \big\{neg(\neg con\_conj, Inc\Delta_i)\big\}$
- $ans[C_i]_i :=$ "The part $Com\Delta_i$ of the requested update is already contained in the instance, and updating the part $Inc\Delta_i$ is incompatible with integrity"

4. `else`
accept the requested update and notify the client C_i and, if the instance is actually changed, refresh the views of all other clients, i.e.,
- `if` $eval\big(con\_conj\big)\big((db_{i-1} \setminus neg(Inc\Delta_i \cup Com\Delta_i)) \cup Inc\Delta_i \cup Com\Delta_i\big) = false$
  `then` $db_i := db_{i-1}$
  (thus the update is *not* performed in the actual instance and some kind of "polyinstantiation" will occur when the update is performed in the log file)
  `else` $db_i := (db_{i-1} \setminus neg(Inc\Delta_i \cup Com\Delta_i)) \cup Inc\Delta_i \cup Com\Delta_i$
- $log[C_i]_i := neg(log[C_i]_{i-1}, Inc\Delta_i) \cup Inc\Delta_i \cup Com\Delta_i \cup con$
  (thus the update comprises an implicit refreshment[4] of the user log $log[C_i]$ which can be computed by the client C_i himself or be communicated to him)

[4] Notably, this refreshment includes the part $neg(log[C]_0 \setminus con, Inc\Delta_i)$ which represents the updated apriori knowledge.

- $ans[C_i]_i :=$ "The part $Com\Delta_i$ of the requested update is already contained in the instance, and the update of the part $Inc\Delta_i$ is *successful*"
- `if` $db_i \neq db_{i-1}$
  `then`, for all $C \neq C_i$, process the refreshment subprotocol of Protocol 2 for the user log $log[C]_{j_0}$ and the sequence of query requests $Q[C]_{j_0}$ constructed from the actual sequence of previous interactions $Q := \langle\, \Theta_1, \ldots, \Theta_{i-1} \,\rangle$ as follows:
  - let j_0 be the largest $j < i$ such that Θ_j is a *successful* (i.e., Case 4 of Protocol 4 applies) view update transaction issued by the client C, if such a j exists; otherwise let j_0 be 0;
  - to form $Q[C]_{j_0}$, first skip all interactions Θ_j up to j_0;
  - then, starting from j_0, if a subsequent interaction Θ_j of Q returned a nonempty answer $ans[C]_j$ to the client C, then add the query request $C : que(ans[C]_j)$ to $Q[C]_{j_0}$; otherwise skip that interaction.

Example 3. We consider a vocabulary *schema* and, again for the sake of simplicity, only one client C with confidentiality policy *pot_sec* and initial log file log_0, and an initial instance db_0 and a view update transaction request as follows: $schema := \{a, b, c, s_1, s_2\}$, $pot\_sec := \{s_1, s_2\}$, $log_0 := con := \{\neg a \vee s_1, \neg c \vee b, \neg s_2 \vee \neg c\}$, $db_0 := \{a, \neg b, \neg c, s_1, s_2\}$, $\Theta := C : vtr(\langle \neg a, c, b \rangle)$.
Since the literal $\neg a$ satisfies (8), $\neg a$ becomes an element of $Com\Delta$. Subsequently, neither the literal c nor the literal b satisfies (7) or (8) and thus they become members of $Inc\Delta$. Thus, at the end of Case 1 we have obtained $Com\Delta = \{\neg a\}$ and $Inc\Delta = \{c, b\}$.
In Case 2, the condition (9) is not satisfied, since

$$\{\neg a \vee s_1, c \vee \neg b, \neg s_2 \vee c\} \cup \{c, b, \neg a\} \cup \{\neg a \vee s_1, \neg c \vee b, \neg s_2 \vee \neg c\} \not\models s_1 \vee s_2.$$

In Case 3, since (10) holds, i.e.,

$$eval\big([\neg a \vee s_1] \wedge [\neg c \vee b] \wedge [\neg s_2 \vee \neg c]\big)\big(\{\neg a, b, c, s_1, s_2\}\big) = false\ ,$$

an incompatibility with the integrity constraints is detected, but this fact must be hidden, since (11) does not hold, i.e.,

$$\{\neg a \vee s_1, \neg c \vee b, \neg s_2 \vee \neg c\} \cup \{\neg c, \neg b, \neg a\} \cup \{\neg([\neg a \vee s_1] \wedge [c \vee \neg b] \wedge [\neg s_2 \vee c]\} \models s_1 \vee s_2.$$

Finally, in Case 4 we obtain

$db_1 := db_0$, since (10) holds, and

$$log_1 := \{\neg a \vee s_1, c \vee \neg b, \neg s_2 \vee c\} \cup \{c, b, \neg a\} \cup \{\neg a \vee s_1, \neg c \vee b, \neg s_2 \vee \neg c\},$$

which is the antecedent of condition (9) already checked to be harmless in Case 2. There are no refreshments, since the instance has not actually been changed.

Theorem 3 (inference-proof view update transactions). *For Int being the subcollection of* queries, provider update transactions and view update transactions *in the sense of Def. 1 and Con, Pol and Know being the full class of all sentences, the controlled execution function that is based on Protocol 1 (query answering), Protocol 2 (provider update transaction processing), modified such that in Case 3 the refreshment subprotocol is performed with the parameters j_0 and $Q[C]_{j_0}$ as described in Case 4 of Protocol 4, and Protocol 4 (view update transaction processing)* preserves confidentiality *in the sense of Def. 3.*

Proof. The omitted proof extends the arguments sketched for Theorem 1. □

To finish this section, we sketch a protocol that combines elementary view updates with elementary provider updates under the restriction to only deal with literals. Omitting the proof, we claim that we can then perform refreshments in the optimized form.

Protocol 5 (elementary update processing with optimized refreshments)
We take the specialized protocol presented in [12] and add refreshments performed with the *optimized* refreshment subprotocol declared in Case 3 of Protocol 3, suitably modified to consider the parameters j_0 and $Q[C]_{j_0}$.

Theorem 4 (inference-proof elementary updates with optimized refreshments). *For Int being the subcollection of* queries with a literal and elementary provider updates and elementary view updates *in the sense of Def. 1 and Con, Pol and Know being the class of literals, the controlled execution function that is based on Protocol 1 (query answering), Protocol 3 (elementary provider update processing), suitably modified to consider the parameters j_0 and $Q[C]_{j_0}$ in the optimized refreshment subprotocol, and Protocol 5 (elementary update processing)* preserves confidentiality *in the sense of Def. 3.*

6 Related Work and Conclusion

We provided a thorough proof of concept for dynamic, instance-dependent inference control of both querying and updating including enforcing integrity constraints within a provider-client architecture of an information system. Basically, the results suggest the following: Once a provider can control a client's ability to gain forbidden information based on answers to arbitrary query sequences, then the provider can extend the inference-proofness achieved to interaction sequences containing updates as well. We formally demonstrated this feature for a specifically instantiated model, focusing on propositional logic, closed queries and lying as a distortion mechanism. We conjecture that similar results can be obtained for first-order logic, open queries and refusals, as studied in previous work on querying. Practically, we somehow have to restrict the expressiveness of some suitable parts of the model, see [9,11], in order to escape from the infeasible algorithmic complexity or even undecidability of solving arbitrary implication problems in the underlying logic. Moreover, to stay within the realm of practicality, we deliberately refrained from considering probabilities and quantifying information gains in terms of information theory. Accordingly, our contribution is in line with many other studies on "possibilistic secrecy", see e.g., [19,24,36,32,42,26,40]. Often such work was extended to "probabilistic secrecy", see, e.g., [25,29,35,38,33,34,21,26]. However, similar to Shannon's perfect encryption, "perfect probabilistic secrecy" seems to be achievable only at a price one cannot afford in general, and practical special cases tend to have a characterization in purely possibilistic terms.

We see the main differences with other approaches to "possibilistic secrecy" as follows. First, while many approaches look for "overall" confidentiality, we achieve confidentiality *discretionarily selected at the finest granularity*, by declaring the concrete sentences that need protection. Second, while many approaches study abstract concepts of confidentiality for some system, we design *concrete protocols* to guarantee discretionary, fine-granulated confidentiality as a control mechanism, to be integrated into an information system and to be inference-proof regarding an "attacker" who is fully aware of the design. Third, while many approaches prefer a static analysis of all potential behaviors of a global system, e.g., [24,36,32,26], or of all potential instances of an information system for a query, e.g., [42,33], in contrast, for favoring availability, we explore a *dynamic approach* to control the interactions that actually take place,

at the price of having to maintain log files in general and to anticipate future interactions at runtime. Fourth, while many approaches employ an abstract notion of a system in terms of abstract traces or states, in contrast (but similar to, e.g., [42,33,26,40]), we deal with the particularities of *logic-oriented information systems*. Finally, our approach has some obvious relationships to the work on mandatory control of information systems with polyinstantiation, see, e.g., [20,31,28,37,16,17,41]. Our approach shares with polyinstantiation the basic underlying idea, but elaborates it in a substantially different way: We declare the specific confidentiality requirements in a discretionary form of finest granularity; in the first place, we materialize the versions only by the provider's reactions to a client; complementary, however, we have to require that the provider maintains a log file for each client; we deal with the problem of inference-proof refreshments of aged views (also treated in [22]); we prove our protocols as secure with regard to an explicitly stated and elaborated notion of confidentiality preservation.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. 6(4), 557–575 (1981)
3. Biskup, J., Bonatti, P.A.: Lying versus refusal for known potential secrets. Data Knowl. Eng. 38(2), 199–222 (2001)
4. Biskup, J., Bonatti, P.A.: Controlled query evaluation for enforcing confidentiality in complete information systems. Int. J. Inf. Sec. 3, 14–27 (2004)
5. Biskup, J., Bonatti, P.A.: Controlled query evaluation for known policies by combining lying and refusal. Ann. Math. Art. Intell. 40, 37–62 (2004)
6. Biskup, J., Bonatti, P.A.: Controlled query evaluation with open queries for a decidable relational submodel. Ann. Math. Art. Intell. 50, 39–77 (2007)
7. Biskup, J., Weibert, T.: Confidentiality policies for controlled query evaluation. In: Barker, S., Ahn, G.-J. (eds.) Data and Applications Security 2007. LNCS, vol. 4602, pp. 1–13. Springer, Heidelberg (2007)
8. Biskup, J., Weibert, T.: Keeping secrets in incomplete databases. Int. J. Inf. Sec. 7, 199–217 (2008)
9. Biskup, J., Embley, D., Lochner, J.-H.: Reducing inference control to access control for normalized database schemas. Information Processing Letters 106, 8–12 (2008)
10. Biskup, J.: Security in Computing Systems – Challenges, Approaches and Solutions. Springer, Heidelberg (2009)
11. Biskup, J., Lochner, J.-H., Sonntag, S.: Optimization of the controlled evaluation of closed relational queries. In: Proc. IFIP/SEC 2009, IFIP Series 297, pp. 214–225. Springer, Heidelberg (2009)
12. Biskup, J., Seiler, J., Weibert, T.: Controlled query evaluation and inference-free view updates. In: DBSec 2009. LNCS, vol. 5645, pp. 1–16. Springer, Heidelberg (2009)
13. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: PODS 2006, pp. 338–347. ACM, New York (2006)
14. Bonatti, P.A., Kraus, S., Subrahmanian, V.S.: Foundations of secure deductive databases. IEEE Trans. Knowledge and Data Eng. 7(3), 406–422 (1995)
15. Brodsky, A., Farkas, C., Jajodia, S.: Secure databases: constraints, inference channels and monitoring disclosure. IEEE Trans. Knowledge and Data Eng. 12(6), 900–919 (2000)
16. Cuppens, F., Gabillon, A.: Logical foundation of multilevel databases. Data Knowl. Eng. 29, 259–291 (1999)
17. Cuppens, F., Gabillon, A.: Cover story management. Data Knowl. Eng. 37, 177–201 (2001)

18. Dayal, U., Bernstein, P.A.: On correct translation of update operations on relational views. ACM Trans. Database Systems 8, 381–416 (1982)
19. Denning, D.E.: Cryptography and Data Security. Addison-Wesley, Reading (1982)
20. 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(2), 129–140 (1987)
21. Evfimieski, A., Fagin, R., Woodruff, D.: Epistemic privacy. In: PODS 2008, pp. 171–180. ACM, New York (2008)
22. Farkas, C., Toland, T.S., Eastman, C.M.: The inference problem and updates in relational databases. In: Proc. DBSec 2001, IFIP Conf. Proc., vol. 215, pp. 181–194. Kluwer, Dordrecht (2001)
23. Farkas, C., Jajodia, S.: The inference problem: a survey. SIGKDD Explor. Newsl. 4(2), 6–11 (2002)
24. Goquen, J.A., Mesequer, J.: Unwinding and inference control. In: Proc. IEEE Symp. on Security and Privacy, Oakland, pp. 75–86 (1984)
25. Gray III, J.W.: Toward a mathematical foundation for information flow properties. In: Proc. IEEE Symposium on Security and Privacy, Oakland, pp. 21–34 (1991)
26. Halpern, J.Y., O'Neill, K.R.: Secrecy in multiagent systems. ACM Trans. Information and Systems Security 12(1), Article 5, 5.1–5.47 (2008)
27. Hegner, S.J.: An order-based theory of updates for relational views. Ann. Math. Art. Intell. 40, 63–125 (2004)
28. Jajodia, S., Sandhu, R.S.: Towards a multilevel secure relational data model. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 50–59 (May 1991)
29. Kenthapadi, K., Mishra, N., Nissim, K.: Simulatable auditing. In: PODS 2005, pp. 118–127. ACM, New York (2005)
30. Langerak, R.: View updates in relational databases with an independent scheme. ACM Trans. Database Systems 15, 40–66 (1990)
31. Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R.: The SeaView security model. IEEE Trans. Software Eng. 16(6), 593–607 (1990)
32. Mantel, H.: On the composition of secure systems. In: Proc. 2002 IEEE Symp. on Security and Privacy, Oakland, pp. 88–101 (2002)
33. Miklau, G., Suciu, D.: A formal analysis of information disclosure in data exchange. J. Computer and System Sciences 73, 507–534 (2007)
34. Motwani, R., Nabar, S.U., Thomas, D.: Auditing SQL queries. In: Proc. Int. Conf. on Data Eng., ICDE 2008, pp. 287–296. IEEE, Los Alamitos (2008)
35. Nabar, S.U., Narthi, B., Kenthapadi, K., Mishra, N., Motwani, R.: Towardsa robustness in query auditing. In: VLDB 2006, VLDB Endowment, pp. 151–162 (2006)
36. Ryan, P.: Mathematical models of computer security. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 1–62. Springer, Heidelberg (2001)
37. Sandhu, R.S., Jajodia, S.: Polyinstantiation for cover stories. In: Deswarte, Y., Quisquater, J.-J., Eizenberg, G. (eds.) ESORICS 1992. LNCS, vol. 648, pp. 307–328. Springer, Heidelberg (1992)
38. Santen, T.: A formal framework for confidentiality-preserving refinement. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 225–242. Springer, Heidelberg (2006)
39. Sicherman, G.L., de Jonge, W., van de Riet, R.P.: Answering queries without revealing secrets. ACM Trans. Database Systems 8(1), 41–59 (1983)
40. Stouppa, P., Studer, T.: Data privacy for ALC knowledge bases. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 309–421. Springer, Heidelberg (2008)
41. Winslett, M., Smith, K., Qian, X.: Formal query languages for secure relational databases. ACM Trans. Database Systems 19(4), 626–662 (1994)
42. Zhang, Z., Mendelzon, A.O.: Authorization views and conditional query containment. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 259–273. Springer, Heidelberg (2004)