REFUSAL IN INCOMPLETE DATABASES

Joachim Biskup and Torben Weibert

Abstract Controlled query evaluation preserves confidentiality in information systems at runtime. A security policy defines a set of *potential secrets* to be hidden from a certain user. Each time the user issues a query, a *censor* checks whether the correct answer would enable the user to infer any of those potential secrets. Given an incomplete information system, the following problem arises: Is it safe to admit that the database cannot provide an answer to a certain query because it lacks the requested information? We show that the answer needs to be refused more often than necessary at first glance, as otherwise the user would be able to make meta level inferences that would lead to a violation of the security policy. A maximally cooperative censor, which preserves confidentiality but only refuses the answer when absolutely necessary, is presented and analyzed.

1. INTRODUCTION

An important goal of a secure information system is preservation of confidentiality. According to some *security policy,* certain information, often referred to as *secrets,* is to be hidden from certain users. This is particularly important when an information system is situated in an open environment where many different users can access it. Typically, confidentiality is enforced by static access rights. One major disadvantage of static access rights is that they are usually assigned at design time. Thus, the administrator needs to properly survey all imaginable queries to the database. This can easily produce a security hole, as the administrator might accidently overlook certain harmful accesses or query sequences.

Unlike static access rights, *controlled query evaluation* preserves confidentiality at run time. Each time the user issues a query, a *censor* checks whether the answer would enable the user to infer one of the secrets defined by the security policy. If this is the case, the answer is distorted by some *modificator.* Two different kinds of distortion are discussed in literature: The information system can either *refuse* to answer [8] or it can give a false answer, commonly referred to as *lying* [6]. Additionally, there exists a third method combining refusal and lying [3, 5].

Previous work on controlled query evaluation is based on logical databases, using a *model theoretic* approach: The database instance db is considered as a structure of some logic, and a query Φ is a sentence in that logic, being *true* if db is a model of Φ, or *false* otherwise, i. e., if db is a model of $\neg\Phi$. Obviously, such information systems are *complete*: Each sentence is either *true* or *false* in the structure, and thus each query can be answered by the system. Controlled query evaluation for complete databases has been exhaustively studied [2, 4].

Unfortunately, a lot of information systems are *incomplete,* in the sense that some information is missing from the database [7]. Querying this information then results in the database answering "I don't know". For example, it can happen that one or more fields of a dataset contain no data. In relational databases, this is usually expressed by null values. There are several reasons for these null values to occur. For example, sometimes a dataset needs to be added to a table even if some of the attributes are unknown at that time. The missing data is then expressed by null values. Furthermore, null values can emerge from *view updates.* When inserting new datasets through a view, the masked out attributes are filled with null values.

In this paper, we adapt the existing methods for controlled query evaluation to *incomplete* databases. Sticking to logical databases, we use a *proof theoretic* approach: We define a database instance db as a (consistent) set of sentences of some logic, called a *theory.* A query (sentence) Φ is defined to be *true* if Φ is implied by db, *false* if $\neg\Phi$ is implied by db, and *undef* if neither Φ nor $\neg\Phi$ is implied by db.

A security policy is defined as a set *pot_sec* of sentences, called *potential secrets.* A potential secret is a sentence the user is not allowed to infer. Potential secrets are considered harmful only if the secret is actually true in the given database instance. A typical example is a sentence like "person X suffers from aids". If the person does actually suffer from aids, this is to be kept secret from an untrustworthy user. On the other hand, if person X does *not* suffer from aids, this fact may be disclosed, as the information is considered harmless. The goal of controlled query evaluation is as follows: Whatever sequence of queries the user issues, he may not rule out that any potential secret is actually *false* in the database instance. Regarding the above mentioned example, this means: It must always appear possible to the user that person X actually does *not* suffer from aids.

When extending controlled query evaluation to incomplete information systems, the basic question is how to handle the situation when the value of the query is *undef* in the current database instance. A first proposal, based on lying and a specific modal logic framework, is found in [6].

In the present paper, we study refusal in a more general formal framework developed in recent years. More specifically, our work is based on the following assumptions: 1. The only distortion method to be used is refusal. 2. The

user knows the security policy and thereby the set of potential secrets to be protected. 3. The user knows the algorithm of the censor. Thus, he knows on which conditions the answer is refused.

This leads to the problem of *meta inferences* drawn from refusals. As the user knows the algorithm of the censor, he can infer about the reason of a refusal, and thus about the value of the query. The knowledge gained from a meta inference might be partial ("the query value is either *true* or *false,* but not *undef*"), but even such partial inferences can be harmful, as demonstrated in Section 4. To avoid such harmful meta inferences, additional refuse-conditions are introduced, so that the meta inferences drawn from a refusal are turned into harmless partial inferences. Regarding cooperativeness, this leads to a drawback, as the answer is refused more often than necessary at first glance. The censor presented in Section 3, which is derived from the censor for complete databases found in [1], uses a total of three additional refuse-conditions. All of these are proven to be essential in order to ensure confidentiality, so our censor is maximally cooperative with regard to the analyzed constraints, namely refusal, potential secrets and known policies (cf. Section 4).

2. INCOMPLETE DATABASES

First we introduce the concept of logical databases and the model for ordinary (non-controlled) query evaluation. Next, we present the four components of controlled query evaluation: user logs, security policies, the censor and the modificator. Finally, a unified framework of controlled query evaluation is introduced which enables us to state a formal definition for the security of an enforcement method.

2.1 Ordinary Query Evaluation

Given a logic **L**, we define a database schema *DS* as set of predicate and constant symbols, and the instance *db* as (consistent) set of sentences of **L**, using only symbols from *DS.* The set of all instances is denoted by *DS*.* The most elementary kind of query is a sentence Φ. We say that the query Φ is *true* in *db* if *db* implies Φ, *false* if *db* implies $\neg\Phi$, and undefined otherwise, i. e., if neither Φ nor $\neg\Phi$ are implied by *db.* This is formalized by the function *eval:*

$$
\begin{aligned}
&eval(\Phi) : DS^* \rightarrow \{true, false, undef\}\\
&eval(\Phi)(db) := \texttt{if } db \models \Phi \texttt{ then } true\\
&\qquad\qquad\quad \texttt{else if } db \models \neg\Phi \texttt{ then } false\\
&\qquad\qquad\quad \texttt{else } undef
\end{aligned}
$$

The operator $\models$ represents the implication operator in the given logic **L**. Note that we assume that implication is decidable in the logic **L** under consideration,

which is not generally true for all logics. For example, in first order logic, implication is only semi-decidable unless we restrict the database instance and queries to certain kinds of sentences. Nevertheless, we continue to depend on this assumption. The examples presented in this paper use propositional logic.

Given a database instance db and a sequence of queries $Q = \langle \Phi_1, ..., \Phi_n \rangle$, the function *query_eval* returns the resulting sequence of answers:

$$query\_eval(Q, db) := \langle ans_1, ..., ans_n \rangle$$
$$ans_i := eval(\Phi_i)(db)$$

EXAMPLE 1 *Consider the following database:*

$$DS = \{a, b, c, d\},\ db = \{a, \neg b, (b \vee d)\}$$

Given the query sequence $Q = \langle a, b, c, d \rangle$, *the resulting answers are* $ans_1 = true$, $ans_2 = false$, $ans_3 = undef$, *and* $ans_4 = true$.

2.2 Controlled Query Evaluation

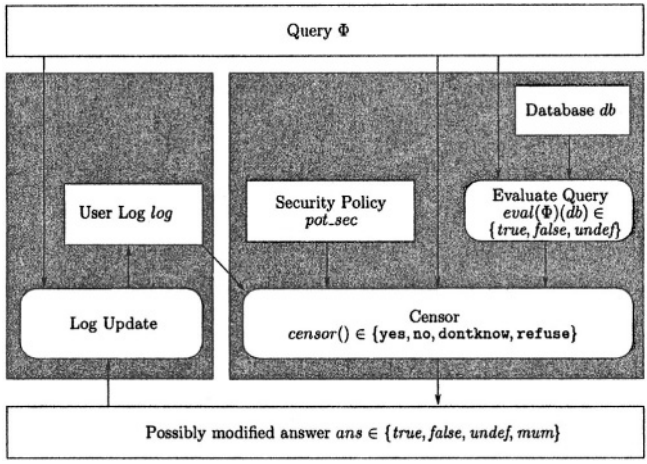

Figure 1. Controlled query evaluation

As outlined in the previous section, ordinary query evaluation returns the correct value of the query in the given database instance, thus providing useful and reliable answers. In presence of a security policy, this might not always be desired. A security policy defines a set of facts the user is not allowed to infer. The goal of controlled query evaluation is to ensure the preservation of the security policy at runtime. Each time the user issues a query, it is checked whether the answer would lead to a violation of the security policy. If this is the case, the original query result is modified.

Controlled query evaluation is achieved by adding four components to ordinary query evaluation: 1. A user log *log,* representing the user's assumed belief about the current database instance. 2. A properly formalized security policy *pot_sec.* 3. A censor function *censor* that decides whether a certain answer needs to be modified, 4. A modificator, performing the modification demanded by the censor. Figure 1 shows how the components of controlled query evaluation interact.

2.2.1 The User Log. In order to decide whether a certain answer would lead to a violation of the security policy, the system needs to maintain an image of the user's assumed knowledge about the database instance *db,* i. e., the facts that the user assumes to be true in the given database instance.

Formally, the user log *log* is defined as a set of sentences. The initial user log log_0 contains all semantic constraints the user is assumed to know prior to the first query. After each query Φ_i, the user log is updated with the answer given by the system: if the answer is *true,* Φ_i is added to the log, if the answer is *false,* $\neg\Phi_i$ is added. If the answer is *undef* or was refused, nothing is added to the user log, as the censor presented does not need to remember those answers.

2.2.2 The Security Policy. The security policy is formally defined as a set *pot-sec* of sentences, called *potential secrets.* The user is not allowed to infer any of those potential secrets $\Psi \in pot\_sec$, if the secret is true in the given database instance, i. e., if $eval(\Psi)(db) = true$ holds. On the other hand, if $eval(\Psi)(db) = false$ holds, this information is not considered harmful. The third case, namely $eval(\Psi)(db) = undef$, is considered harmless as well.

EXAMPLE 2 *Imagine a database containing information about applicants for a job vacancy, including information about certain diseases the applicants might suffer from, say cancer. If a certain applicant suffers from cancer, this information must be kept secret, as this knowledge might keep him from being chosen for the job. On the other hand, if an applicant does not suffer from cancer, this fact may be disclosed. Controlled query evaluation must ensure that the user querying the database cannot rule out that a certain person is healthy.*

We assume that the user knows the security policy, i. e., the set of sentences $pot\_sec = \{\Psi_1, ..., \Psi_n\}$. In previous work, both known and unknown policies have been studied. The essential conclusion was that censors for known policies have to be more restrictive than those for unknown policies [2].

2.2.3 The Censor. The censor decides whether the answer needs to be distorted (refused or modified) in order to preserve confidentiality according to the given security policy. Formally, we have a function *censor* with the database *db,* the security policy *pot_sec,* the current user log *log* and the query

Φ as parameters[1]. The censor returns one of the values `yes`, `no`, `dontknow` or `refuse`, instructing the modificator what answer to give instead of the original one.

The algorithm of the censor function depends on the three conditions *awareness* of the security policy (known or unknown policies), *type* of the security policy (potential secrets or secrecies, cf. Section 5) and kind of *distortion* that the censor is allowed to use (refusal, lying or both). For complete information systems, all of the resulting twelve cases have been studied [4]. For incomplete information systems, a secure censor is presented in Section 3, thereby focusing on one of these cases, namely refusal under known potential secrets.

2.2.4 The Modificator. The modificator transforms the original answer to the query into the answer demanded by the censor. It then passes the (possibly modified) answer to the user: *true* if the censor returned `yes`, *false* if the censor returned `no`, *undef* if the censor returned `dontknow`, and the special value *mum* if the censor returned `refuse`.

2.3 Security of an Enforcement Method

In Section 3 a secure censor for refusal under known potential secrets is presented. In order to prove that a certain mechanism for controlled query evaluation preserves confidentiality, we need a proper definition of security.

A method for controlled query evaluation can be formalized as a function

$$control\_eval^X(Q, log_0, db, pot\_sec) := \langle (ans_1, log_1), ..., (ans_n, log_n) \rangle$$

where X indicates the name of the method under consideration. In each step, the answer is generated by the modificator according to the decision of the censor, and the user log is updated accordingly:

case $censor^X(db, pot\_sec, log_{i-1}, \Phi_i)$:

`yes`	`then`	$ans_i := true;$	$log_i := log_{i-1} \cup \{\Phi_i\}$ |
`no`	`then`	$ans_i := false;$	$log_i := log_{i-1} \cup \{\neg\Phi_i\}$ |
`dontknow`	`then`	$ans_i := undef;$	$log_i := log_{i-1}$ |
`refuse`	`then`	$ans_i := mum;$	$log_i := log_{i-1}$

Each method comes with an associated precondition $precondition^X$ which defines the "admissible" arguments.

The goal of controlled query evaluation is to hide the fact that a potential secret is actually true in the given database instance. More precisely, given a

[1] As indicated by Figure 1, the censor needs to know the database instance db only to determine the query value $eval(\Phi)(db)$.

potential secret $\Psi \in pot\_sec$, the user must not be able to exclude that Ψ is *false* or *undef* in the actual database instance db_1. In other words: There must be another database instance db_2 in which Ψ is *false* or *undef*, and which would have produced the same answers as db_1 did. From the user's point of view, db_1 and db_2 are indistinguishable. This can be formalized as follows:

DEFINITION 3 (CONFIDENTIALITY FOR KNOWN POTENTIAL SECRETS) *Let* $control\_eval^X$ *be a controlled query evaluation with* $precondition^X$ *as associated precondition for admissable arguments. Then* $control\_eval^X$ *is defined to preserve confidentiality (or, as we say:* $censor^X$ *is safe) iff*

for all finite query sequences Q,
for all security policies pot_sec,
for all potential secrets $\Psi \in pot\_sec$,
for all initial user logs log_0,
for all instances db_1 *so that* (db_1, log_0, pot_sec) *satisfies* $precondition^X$,
there exists db_2 *so that* (db_2, log_0, pot_sec) *satisfies* $precondition^X$
and the following two conditions hold:

(a) *[*db_1 *and* db_2 *produce the same answers]*
$control\_eval^X(Q, log_0, db_1, pot\_sec) =$
$control\_eval^X(Q, log_0, db_2, pot\_sec)$

(b) *[*Ψ *is not true in* db_2*]*
$eval(\Psi)(db_2) \in \{false, undef\}$

3. REFUSAL FOR KNOWN POTENTIAL SECRETS

In this section, we present a censor for refusal under known potential secrets for incomplete databases. We start with a discussion on refusal as an enforcement method for controlled query evaluation and its advantages and disadvantages. Next, the censor for complete information systems, as found in literature, is reviewed. Then a censor for incomplete information systems is presented. Finally, we consider the quality of the presented censor by analyzing its cooperativeness.

3.1 Outline of Refusal

In this paper, we focus on refusal as a means to distort harmful answers, that means: 1. The censor may refuse the answer in order to hide possibly dangerous answers. 2. The censor may not give false answers, i.e., it may return `yes` only if $eval(\Phi)(db) = true$ holds, no only if $eval(\Phi)(db) = false$ holds, and `dontknow` only if $eval(\Phi)(db) = undef$ holds.

What's the advantage of refusal? Even if some answers may be refused, the information system does only provide reliable information, i.e., facts that are actually true in the database instance. This can be important when the database deals with sensitive information, for example in military applications or in a

hospital, where doctors and nurses need reliable information to choose the right medication.

The main disadvantage of refusal is that the user immediately notices that an answer has been distorted. This might not always be desired. Moreover, the user can (on meta level) infer about the reason of the refusal. We will see that a secure censor has to refuse the answer more often than necessary at first glance in order to avoid these meta inferences. This leads to a loss of cooperativeness, which is analyzed in Section 4.

3.2 Refusal in Complete Databases

Previous work on controlled query evaluation deals with complete databases, i.e., databases in which every query has a value of *true* or *false*. For such complete databases, the following *complete censor* preserves confidentiality [1]:

$$censor^{\texttt{complete}}(db, pot\_sec, log, \Phi) =$$
$$\texttt{if } (\exists \Psi \in pot\_sec)\ [\ log \cup \{\Phi\} \models \Psi \text{ or } log \cup \{\neg\Phi\} \models \Psi\ ]$$
$$\quad \texttt{then refuse}$$
$$\texttt{else if } eval(\Phi)(db) = true \texttt{ then yes else no}$$

Table 1 shows the functioning of the complete censor. The decision of the censor depends on two factors: First a *security configuration* (represented by a line in the table) is identified by checking which of the possibly resulting user logs ($log \cup \{\Phi\}$ if the answer *true* is given, or $log \cup \{\neg\Phi\}$ if the answer *false* is given, respectively) would enable the user to infer any of the potential secrets. Then the decision is determined by the actual query result (*true* or *false,* represented by a column in the right part of the table).

Clearly, the answer has to be refused if the resulting user log ($log \cup \{\Phi\}$ if $eval(\Phi)(db) = true$, $log \cup \{\neg\Phi\}$ if $eval(\Phi)(db) = false$) would imply a potential secret. These *real* `refuse`*-conditions* are marked black in the table.

Unfortunately this is not sufficient. Imagine the user issues a query Φ for that only $log \cup \{\Phi\}$ would imply a potential secret but not $log \cup \{\neg\Phi\}$. As the

$(\exists \Psi \in pot\_sec)$		$eval(\Phi)(db) = \ldots$	
$[log \cup \{\Phi\} \models \Psi]?$	$[log \cup \{\neg\Phi\} \models \Psi]?$	*true*	*false*
X	X	refuse	refuse
X	–	refuse	refuse
–	X	refuse	refuse
–	–	yes	no

Table 1. Refusal censor for complete databases

$(\exists \Psi \in pot\_sec)$		$eval(\Phi)(db) = \ldots$		
$[log \cup \{\Phi\} \models \Psi]$?	$[log \cup \{\neg\Phi\} \models \Psi]$?	*true*	*false*	*undef*
X	X	refuse	refuse	refuse
X	-	refuse	no	refuse
-	X	yes	refuse	refuse
-	-	yes	no	dontknow

Table 2. Refusal censor for incomplete databases

user knows the user log and (as we suppose) the set of potential secrets, he is able to determine the security configuration, i. e., the line of the table the answer must originate from, in this case the second line. Furthermore, we assume that the user knows the algorithm of the censor, so he knows what answers the censor gives under this security configuration. If there was only a single (real) `refuse`-condition in the second line of the table (for $eval(\Phi)(db) = true$), the user could figure from the answer *mum* that $eval(\Phi)(db) = true$ must hold, because there is no other query value that could have led to this answer. This problem of *meta inferences* is solved by adding some *additional* `refuse`-*conditions,* marked gray in the table. Now the censor answers `refuse` even if $eval(\Phi)(db) = false$ holds. As a result, the user cannot infer about the value of Φ in db anymore.

3.3 Refusal in Incomplete Databases

The censor for complete databases presented in the previous section only handles query values of *true* and *false.* In incomplete databases, there is a third possible value for a query, namely *undef.* When developing a censor for incomplete databases, the main problem is how to deal with these *undef* values.

As stated in Section 2.3, it is regarded harmless if the user knows that a potential secret is undefined in the current database instance. Nevertheless, answering *undef* on an arbitrary query is not necessarily safe. Imagine a situation where the user can infer that "if the database does not know whether b or $\neg b$ holds, then the potential secret s must hold". Although the logic exploited for the user log is not powerful enough to express such sentences, such inferences could be made on *meta level.* As a result, the system must not generally admit that it does not know the value of a query.

Table 2 shows a censor $censor^{incomplete}$ for refusal under known potential secrets in incomplete databases. We will prove that the censor is secure in the sense of Definition 3 if the following precondition holds: The initial user log log_0 does not entail any of the potential secrets, and the database and the initial user log are consistent which each other, i. e. the user does not initially believe

facts that are false in the database[2]. So an argument (db, log_0, pot_sec) must satisfy the following precondition:

$$precondition^{incomplete}(db, log_0, pot\_sec) := \\ (db \cup log_0) \text{ is consistent and } (\forall \Psi \in pot\_sec)[\, log_0 \not\models \Psi \,]$$

It is obvious that the censor keeps the second condition as an invariant for all of the following user logs, so we have

$$(\forall \Psi \in pot\_sec)[log_i \not\models \Psi] \text{ for all } i \leq n. \tag{1}$$

THEOREM 4 (SECURITY OF REFUSAL CENSOR) *$control\_eval^{incomplete}$ preserves confidentiality in the sense of Definition 3.*

We only give a rough sketch of the proof. Given a database instance db_1, an initial user log log_0, a security policy *pot_sec* so that $precondition^{incomplete}$ is satisfied, and a query sequence $Q = \langle \Phi_1, ..., \Phi_n \rangle$ resulting in the answers $\langle ans_1, ..., ans_n \rangle$, we define a second database instance by gathering all sentences that were added to the user log throughout the query sequence:

$$db_2 = \left(\bigcup_{1 \leq i \leq n,\ ans_i = true} \{\Phi_i\} \right) \cup \left(\bigcup_{1 \leq i \leq n,\ ans_i = false} \{\neg \Phi_i\} \right)$$

As db_2 only contains sentences that are implied by db_1, db_2 is consistent, so $precondition^{incomplete}(db_2, log_0, pot\_sec)$ is satisfied by db_2. As db_2 is a subset of log_n, and by (1), db_2 does not imply any of the potential secrets, thus satisfying condition (b) of Definition 3. Finally, it can be shown by induction that the same answers are given under db_1 and db_2, satisfying condition (a).

EXAMPLE 5 *Recall Example 2 and the database containing information about what diseases a certain person suffers from. Limiting the diseases under consideration to aids, cancer and influenza, the database schema might contain the following atoms*

$$DS = \{AIDS, CANCER, FLU, DISEASED\}$$

where DISEASED indicates that the person suffers from any of the three explicitly named diseases. We specify a security policy that disallows the user to infer that the person suffers from aids or cancer, so we have the potential secrets

$$pot\_sec = \{AIDS, CANCER\}.$$

[2]The latter condition is not essential for the security of the censor but yet reasonable to presume. Otherwise, all queries regarding the conflicting facts would result in a refusal, as adding the correct answer would make the log inconsistent, and then all of the potential secrets would be implied by the user log.

The user knows that if the person is diseased, it must suffer from aids, cancer or influenza, so we have the following initial user log:

$$log\_0 = \{DISEASED \rightarrow (AIDS \vee CANCER \vee FLU),\\ AIDS \rightarrow DISEASED,\ CANCER \rightarrow DISEASED,\\ FLU \rightarrow DISEASED\}$$

Now imagine the database knows that the mentioned person suffers from cancer but not from aids, whereas the database does not know whether the person has influenza. So the database instance is as follows:

$$db = \{\neg AIDS, CANCER, DISEASED\}$$

Table 3 shows the answers given by the censor for the query sequence

$$Q = \langle AIDS, DISEASED, CANCER, FLU\rangle.$$

The l *and* c *values indicate the line and column of Table 2 the answer originates from.*

According to Theorem 4, for any of the potential secrets, there exists a database instance that would have produced the same answers, and in which this potential secret is either false or undef. It can easily be verified that

$$db_2 = \{\neg AIDS, DISEASED\}$$

satisfies this condition for each of the two potential secrets.

4. COOPERATIVENESS

In the previous section we have presented a secure censor for refusal under known potential secrets. In order to ensure the highest possible cooperativeness, we are interested in finding a censor that only distorts the answer in case it is absolutely necessary. We have found that sometimes the answer needs to be refused even if the otherwise resulting user log wouldn't have implied a potential secret, protecting the *real* `refuse`-*conditions* against possible meta inferences. Surely, these *additional* `refuse`-*conditions* have an impact on the cooperativeness of the method, as the answer is more often refused than it is originally necessary. This leads to the idea of defining the quality of a censor by the number of additional `refuse`-conditions in the decision table. The censor presented in Section 3.3 imposes a total of three additional `refuse`-conditions. In this section we show that this is the least possible amount and thereby that this censor is maximally cooperative.

Imagine the user issues a query that is refused by the censor. The user can then make the following inferences on meta level:

	Φ_i	$eval(\Phi_i)(db)$	censor result	log_i
$i = 1$	$AIDS$	*false*	`no` $l = 2,\ c = 2$	$log_0 \cup \{\neg AIDS\}$
	Giving the correct answer is harmless.			
$i = 2$	$DISEASED$	*true*	`yes` $l = 4,\ c = 1$	$log_0 \cup \{\neg AIDS, DISEASED\}$
	Still no need to refuse. The user can now infer from his log that either $CANCER$ or FLU must hold.			
$i = 3$	$CANCER$	*true*	`refuse` $l = 2,\ c = 1$	$log_0 \cup \{\neg AIDS, DISEASED\}$
	Clearly, the answer must be refused.			
$i = 4$	FLU	*undef*	`refuse` $l = 3,\ c = 3$	$log_0 \cup \{\neg AIDS, DISEASED\}$
	The database does not know whether the person has influenza or not. Admitting that would not be harmful. Actually, we have an additional `refuse`-condition here. The answer is only refused for the sake of the real `refuse`-condition in this line (in the second column), to make these two cases indistinguishable to the user.			

Table 3. Controlled query evaluation for Example 5

- As the user knows the set of potential secrets, he can determine the security configuration of his query and thereby identify the line of the decision table the answer originates from.

- As the user knows the algorithm of the censor and the decision table, he can compare the answer to the entries found in the corresponding line of the decision table. He can then identify the column(s) the answer might originate from.

- If there is exactly one `refuse` in the line under consideration, the column can be fully identified, and so can the query value.

- If there are two fields containing a `refuse`, the user can still gain partial knowledge about the query value, i. e. a disjunction "the query value is either a or b", where a and b are the values of the corresponding columns.

- If there are `refuses` in all three columns of this line, no information about the query value can be gained.

Obviously, the possibility of gaining information on meta level depends on the number of `refuse`-conditions in a given line, and the columns where these are located.

The second and the third line of the decision table contain only one real `refuse`-condition each. Without the additional `refuse`-conditions introduced by our censor, these single `refuses` would enable the user to gain full knowledge of the query value. So additional `refuse`-conditions are unquestionably necessary in those lines. Now, with a `refuse` in both the first (or second, respectively) and third column, the user can only infer that either $eval(\Phi)(db) = undef$ or $eval(\Phi)(db) = true$ (or *false,* respectively) must hold, but not *which* of these alternatives. This disjunction is safe according to Theorem 4.

In the first line of the decision table, the situation is slightly different. There are two real `refuse`-conditions, so even without an additional `refuse`, the user can only gain partial knowledge about the query value. But the resulting disjunction "the query is either *true* or *false"* is not necessarily safe, as demonstrated by the following example.

EXAMPLE 6 *Imagine a weakened censor that lacks the additional* `refuse`*-condition in the third column:*

$(\exists\Psi \in pot\_sec)$		$eval(\Phi)(db) = \ldots$		
$[log \cup \{\Phi\} \models \Psi]?$	$[log \cup \{\neg\Phi\} \models \Psi]?$	*true*	*false*	*undef*
X	*X*	`refuse`	`refuse`	`dontknow`

Consider the following situation:

$$DS = \{s, b\},\ db = \{s\},\ pot\_sec = \{s, \neg s\},\ log_0 = \emptyset$$

When the user issues the query sequence $Q =< b, s \vee b, s >$*, the system answers as follows:*

	Φ_i	$eval(\Phi_i)(db)$	*censor result*	ans_i	log_i
$i = 1$	b	*undef*	`dontknow` $(l = 4, c = 3)$	*undef*	$\emptyset$
$i = 2$	$s \vee b$	$s \vee b$	`yes` $(l = 3, c = 1)$	$s \vee b$	$\{s \vee b\}$
$i = 3$	s	s	`refuse` $(l = 1, c = 1)$	*mum*	$\{s \vee b\}$

From the first answer, the user can infer that the database knows nothing definite about b*. From the second answer, the user can infer that the database knows that* $s \vee b$ *holds, i.e.,* $s \vee b$ *or even* s *must be implied by* db*. The third*

query leads to a security configuration where both $log \cup \{\Phi\}$ *and* $log \cup \{\neg\Phi\}$ *would imply a potential secret, so the answer must originate from the first line of the censor table. As the answer is refused, the user can infer from the weakened decision table that either* s *or* $\neg s$ *must hold in* db*. From the first two answers he knows that* $\neg s$ *cannot be true. So it must hold that* $db \models s$*.*

In order to avoid this meta inference, the safe refusal censor from Section 3 introduces the additional `refuse`-condition in the rightmost column of the first line. When the user receives a refusal from this line now, he cannot infer anything about the query value anymore, as the resulting disjunction "the value is either *true, false* or *undef*" contains no information.

As we have pointed out in this section, additional `refuse`-conditions prevent the user from making such inferences on meta level by either turning lines that formerly contained harmful single `refuses` into lines with harmless disjunctions (as in the second and third line), or by turning lines that formerly contained possibly harmful disjunctions into lines that contain a harmless total of three `refuse`-conditions (as in the first line).

As we have shown, under known potential secrets for incomplete databases, a minimum of three additional `refuse`-conditions is required in order to preserve confidentiality. Thus, with regard to cooperativeness, the censor presented in Section 3.3 can be considered ideal.

5. CONCLUSION

We have developed a censor for refusal under known potential secrets for incomplete databases. As we have shown, the resulting enforcement method for controlled query evaluation preserves confidentiality according to the security definition in Section 2.3. There are two kinds of cases where the censor refuses the answer: Four real `refuse`-conditions prevent the user from inferring potential secrets on logical level. Three additional `refuse`-conditions avoid inferences possibly made on meta level. We have shown that the censor needs a minimum of three additional `refuse`-conditions in order to preserve confidentiality. Thus, we have found an ideal censor for the given constraints (potential secrets, known policy, refusal).

As pointed out in Section 4, the most challenging part of designing a safe censor is the identification and treatment of harmful meta inferences. Successful efforts have been made to use a modal logic representation for the user log, which enables us to formalize sentences like "the database knows that Φ holds" or "the database does not know whether Φ or $\neg\Phi$ holds". Inferences on meta level can then be expressed in formal logical sentences and can be handled much easier. This will be covered by future work.

In the present paper, we only consider security policies based on potential secrets. There is a second kind of security policies called *secrecies* [1], where

pairs of complementary sentences are protected so that the user cannot decide which of the alternatives holds. For complete databases, secrecies can easily be transformed into a set of potential secrets and then handled by a censor designed for potential secrets, if certain requirements are met [2]. It is still an open question if this reduction can also be made for incomplete databases.

Finally, we have only studied known policies so far. Given an unknown policy, the user cannot determine the line of the table an answer originates from. It is still to be analyzed how the censor can take advantage of this.

References

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

[2] Joachim Biskup and Piero A. Bonatti. Confidentiality policies and their enforcement for controlled query evaluation. In *Proc. of ESORICS 02, Zürich, Switzerland, October 14-16, 2002,* volume 2502 of *Lecture Notes in Computer Science,* pages 39–54. Springer, 2002.

[3] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. In *Proc. of FoIKS 02, Schloss Salzau, Germany, February 20-23, 2002,* volume 2284 of *Lecture Notes in Computer Science,* pages 49–66. Springer, 2002.

[4] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for enforcing confidentiality in complete information systems. *To appear in International Journal of Information Security,* 2004.

[5] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Annals of Math. and Artificial Intelligence,* 40:37–62, 2004.

[6] P. A. Bonatti, S. Kraus, and V.S. Subrahmanian. Foundations of secure deductive databases. *IEEE Transactions on Knowledge and Data Engineering,* 7(3):406–422, 1995.

[7] J. Chomicki and G. Saake, editors. *Logics for Databases and Information Systems,* chapter 10. Kluwer Academic Publishers, 1998.

[8] George L. Sicherman, Wiebren de Jonge, and Reind P. van de Riet. Answering queries without revealing secrets. *ACM Transactions on Database Systems,* 8(1):41–59, 1983.