Resolving the Tension between Integrity and Security Using a Theorem Prover

Subhasish Mazumdar David Stemple Tim Sheard
University of Massachusetts, Amherst*

Abstract

Some information in databases and knowledge bases often needs to be protected from disclosure to certain users. Traditional solutions involving multi-level mechanisms are threatened by the user's ability to infer higher level information from the semantics of the application. We concentrate on the revelation of secrets through a user running transactions in the presence of database integrity constraints. We develop a method of specifying secrets formally that not only exposes a useful structure and equivalence among secrets but also allows a theorem prover to detect certain security lapses during transaction compilation time.

1. Introduction

Some information in databases and knowledge bases often needs to be protected from disclosure to certain users. In a multilevel-secure system [Fernandez et al. 81] a user is not allowed to see data at security levels higher than his/her own by checking the classification levels of the user and the data at each read/write operation. However, as Morgenstern [Morgenstern 87] points out, " ... a system's security mechanism could not ... prevent the user's ability to infer higher level information based on the semantics of the application — unless the system also had available to it relevant knowledge of the application." In this paper, we consider the semantics defined by the database integrity constraints and show that it is possible during database system design to catch interesting potential disclosures that transactions, when they are run, can make to an intelligent user who is knowledgeable about those integrity constraints. While proving the compatibility of integrity constraints and database transactions has been examined by a few researchers [e.g., [Gardarin and Melkanoff 79], [Casanova and Bernstein 80], [Nicolas 82], [Henschen et al. 84]], we believe this to be the first analysis of using a transaction verifier to discover security lapses caused by the maintenance of integrity.

In the ADABTPL environment [Stemple and Sheard 87], database transactions are analysed in the presence of integrity constraints with the help of a theorem prover. In order to guarantee that they will always respect integrity constraints when they are run, the prover attempts to establish a standard theorem[Sheard and Stemple 86] and give several kinds of feedback to the database designer depending on the way the proof succeeds or fails [Stemple et al. 87].

If successfully run, a transaction does implicitly answer at least two sets of queries — the preconditions of the transactions before the transaction ran, and the integrity constraints afterwards. A transaction whose effect is guaranteed to obey the integrity constraints can only fail to run either because the inputs failed to meet their type constraints or because the query evaluating the transaction's preconditions returned *false*. A malicious user who knows the integrity constraints, the preconditions, and the input may therefore be able to learn secrets through observation and diligent application of transactions. In case a transaction runs successfully, it may be possible to repeat it a finite number of times until it fails and thereby draw further conclusions.

We introduce a formal means of specifying *secrets*, that allows mechanical analysis and detection of certain security lapses resulting from transactions; and we incorporate this specification in ADABTPL, in which one can already express integrity constraints.

In [Bancilhon and Spyratos 77], Bancilhon and Spyratos defined a "protection problem" by the answers to the following questions:

1. What do we protect?

*This paper is based on work supported by the National Science Foundation under grants DCR-8503613 and IRI-8606424, and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764.

To appear in ACM SIGMOD 1988

2. Against whom do we protect?

3. How do we protect?

4. What does *protect* mean?

To put this paper in perspective, we will list our corresponding answers:

1. Information that has been declared secret.

2. Against a properly identified user who can interact with the system by *running transactions.*

3. By checking *at compile time* to see if transactions can reveal information improperly.

4. Making sure that the user cannot deduce secret information from the success/failure of the transaction when it is run.

Since most researchers have concentrated on security lapses through query evaluation, many of them focussing on statistical databases [Dobkin et al. 79], [Denning et al. 79], their answers to the last three questions are different from ours.

2. Overview of our Approach

The system designer using ADABTPL expresses integrity constraints as part of the global schema defining the object types, and specifies transactions as programs in which the input type and preconditions are declared. Designers are also allowed to specify secrets by means of special formulas containing free variables. A breach of security occurs when a user determines bindings for these free variables that makes the formulas true.

It is assumed that the user[1] is aware of the integrity constraints and the preconditions as well as the body of the transaction program. The aim of our system is to detect situations when the user (modelled as an excellent deductive system) is able to learn one of the secrets by virtue of running a transaction. This is done by making our system imitate the user attempting to deduce the secret from knowledge gained by a transaction's success or failure. Note that we are never concerned about the actual data in the database; our analysis is done at system design time, i.e., during compilation of the transaction, at which time the type restrictions and the integrity constraints constitute all the semantic information of the application.

First, we attempt to prove for each of the secrets that it is deducible from the integrity constraints alone. In the (rather pathetic) case of one of these attempts succeeding, either the constraints are too strong and need to be relaxed, or the attempt at secrecy is overly ambitious and needs to be curtailed.

[1] We do not discuss specifically the multi-level nature of the classification of users. We assume just two levels.

Second, for a transaction, the preconditions specified by the user and also the tests generated by the safety theorem prover are analysed for revelatory power. It is quite possible that certain sufficient tests (which are of interest because they suggest cheaper means of run-time testing) turn out to be *secure* while the tests they replace are not. We consider this to be possibly valuable information for the system designer and report it as feedback.

Third, the system helps detect whether or not the user may, by running the transaction iteratively deduce more. This is of particular importance when *partial inference* [Morgenstern 87] is considered. A user is said to make a partial inference about an entity that is being guarded as a secret, if he/she gains knowledge that is capable of reducing the uncertainty of the entity's state by using semantics of the application e.g., the type constraints of the data, the integrity constraints.

Now let us make a few remarks regarding the approach. Though speed is generally considered a problem with theorem provers, the safety theorem for the hire transaction with 600 terms took 38 seconds on the TI Explorer II. In general, the efficiency of the theorem prover depends on the lemma base. With an inadequate lemma base, the theorem prover wanders into never-never land trying an unbounded number of possibilities. In short, speed depends on the well-preparedness of the theory the prover is equipped with. In order to make the prover execute efficiently, we proved many theorems that deal with the interaction of the database operations (insert, delete, update) with the typical database constraint predicates so that the bodies of those function and predicate definitions need never be "opened up" during a proof process. Further, we raised the level of generality of theorems by extending the Boyer-Moore theorem prover to deal with higher-order (generic) functions and meta-lemmas (lemmas containing function variables).

Our analysis, focussing on transactions, is orthogonal to standard techniques for detection of compromise through queries. Moreover, by no means do we claim to detect all possible compromises resulting from inference through transactions. We do attempt to show however that an interesting class can be caught and subsequent research may expand on the class of compromises that can be detected in this manner. Though the iterative transaction example involves the sum function, we have not examined statistical databases leaving it for future study since currently our prover is not equipped with a theory for the statistical functions.

3. Transaction Analysis and Theorem Proving

In this section, we give an overview of our approach to transaction analysis based on theorem proving at compile time. We introduce the salient features of the ADABTPL language[Stemple and Sheard 87], followed by the formulation of the safety theorem that the transac-

tion verifier works on. A transaction is *safe* if it produces a database state obeying the integrity constraints whenever its input database obeys those constraints (see [Sheard and Stemple 86] for details). If the transaction cannot be certified safe, several forms of feedback are offered [Stemple et al. 87] including additional tests to be added to the transaction's preconditions, additional updates, and postconditions.

Our inference engine consists of several theorem provers built on top of the kernel of a Boyer-Moore theorem prover [Boyer and Moore 79]. We have built the theories of three abstract data types, finite sets, lists and tuples by proving hundreds of theorems mechanically. Using these theorems as rewrite rules, our safety verifier has proven mechanically the safety, i.e., the property of respecting the integrity constraints of the database, of nontrivial transactions on moderately constrained databases.

A user specifies a database with a schema (as in Figure 1) comprising a set of type definitions: domain types, tuple types, finite set types, and a database type, the last typically consisting of a tuple type whose components are relations. Each type, at any level, can have arbitrary predicates specified as integrity constraints on objects of that type, and implicitly on any higher level type which contains it. The totality of these predicates form a single predicate stating the consistency property of database states. The constraint classes which may be expressed in where clauses include arithmetic constraints, set membership, set containment, null intersection of two sets, universal and existential quantification, redundancy of information in a tuple with a function over a set (e.g., aggregate constraint).

ADABTPL is appropriate as a high level language for users writing transactions. Figure 2 is a transaction written in this language. Note there is a header statement with type declarations for input, and also the preconditions expressed in the constraint language. A translator converts this transaction program into functional[2] form, that takes a typed input and a database state and return an updated database.

A safety theorem for a transaction f (in the functional form) states that if given a consistent database state db and well-typed input i, the transaction returns a consistent database state. If IC is the integrity constraint on the database, the safety theorem for T is:
IC(db) $\wedge$ TYPED(i) $\Rightarrow$ IC(f(i,db)).

The safety theorem prover applies theorems (which we nearly always call lemmas). A lemma is of the form

$$(L = R) \Leftarrow H,$$

where H is a hypothesis (conjunction of zero or more predicates) and (L = R) is a rewrite rule stored as an equation and used in a left to right manner (the left-hand side L is to be replaced by the right-hand side R if the hypothesis H can be proved to be true with the current set of truths).

[2]Our kernel theorem prover uses (heuristic-based) structural induction proof techniques expoiting the recursive nature of functions. For mechanical analysis, we express everything functionally.

```
type
  person= [ pid: number; pname: string;
                placed: boolean ];
  person-rel= set of person
      where key(person-rel, pid);
  job=[ jid: number; jdescript: string ];
  job-rel= set of job where key(job-rel, jid);
  company= [ cid: number; comp-name: string;
                totsal: number];
  company-rel= set of company
    where key(company-rel, cid);
  application= [pid: number; jid: number];
  application-rel= set of application;
  offer= [ cid: number; jid: number;
                num-of-posns: number ]
    where num-of-posns ≠ 0 ;
  offer-rel= set of offer
    where key(offer-rel, cid, jid);
  placement= [ pid: number; jid: number;
    cid: number; salary: number ];
  placement-rel= set of placement
    where key(placement-rel, pid);
database job-agency :
    [ persons: person-rel; jobs: job-rel;
      applications: application-rel;
      offerings: offer-rel;
      companies: company-rel;
      placements: placement-rel ]
    where
      contains(persons.pid, placements.pid) ;
      contains(companies.cid, placements.cid) ;
      contains(jobs.jid, placements.jid) ;
      contains(jobs.jid, offerings.jid) ;
      contains(companies.cid, offerings.cid) ;
      contains(persons.pid, applications.pid) ;
      contains(jobs.jid, applications.jid) ;
      for x in persons
          x.placed is-redundant-with
          x.pid in placements.pid ;
      null-intersection (applications.pid,
                placements.pid);
      for c in companies
          c.totsal is-redundant-with
          sum( (all p in placements
                where p.cid = c.cid),
            salary);
```

Figure 1: Schema of job-agency

```
transaction hire (comp, hiree, jb, sal: number ) ;
preconditions
 hiree in persons.pid ;
 [comp, jb] in offerings.[cid, jid] ;
 hiree not-in placements.pid ;
begin
 update p in persons where p.pid = hiree by
  [ placed := true ] ;
 update o in offerings where
  o.cid = comp and o.jid = jb by
   if o.number-of-positions = 1
    then delete o
    else [ number-of-posns := number-of-posns - 1 ] ;
 insert [hiree, jb, comp, sal] into placements;
 remove p from applications where p.pid = hiree;
 update c in companies where c.cid = comp by
  [ totsal := totsal + sal ]
end;
```

Figure 2: Hire Transaction

Note that a function definition is a kind of rewrite rule for rewriting calls of the function in terms of the function body.

Higher order functions are used to encapsulate structured updates and integrity constraint primitives. Lemmas about higher-order functions are called meta-lemmas and in part of the hypotheses. See [Sheard and Stemple 86] for more on our use of the higher-order theory.

If the safety theorem prover is unable to establish the safety theorem, it attempts to generate necessary and sufficient tests followed by a phase for better tests, some of which may be only sufficient. For example, consider the integrity constraint:

contains(r1 r2),

where r1 and r2 are relations, and a transaction that deletes an element from r1. We need to prove that the integrity constraint is still valid after the deletion, i.e., that the *updated integrity expression*:

contains(delete(a r1) r2)

holds given that the integrity constraint held on the original database. Thus, we need to prove the following:

contains(delete(a r1) r2) $\Leftarrow$ contains(r1 r2).

To prove the above formula, the system applies the rewrite rule

(contains(delete(a r1) r2)) = (not(member(a r2)))
$\Leftarrow$ (contains(r1 r2)).

and arrives at the unsimplifiable formula

not(member(a r2))

which is also the necessary and sufficient test for the validity of the updated integrity expression given that the integrity constraint held in the original database.

It may however be cheaper (in terms of run-time testing) to arrive at a *sufficient condition* that assures the same predicate if true. Towards this end, the system attempts to *back-chain* from the residue and output a list of sufficient conditions.

Consider a variation of the previous example in which the transaction remains as before, but the constraint becomes

contains(project(r1 pid) project(r2 pid)).

The test obtained is

not(member(pid(a) project(r2 pid))).

If there is a lemma of the form
((test = true) $\Leftarrow$ hypoth), then hypoth can be a sufficient test. Now consider a situation where r2 has a type restriction (e.g., the pid of each element must be less than 1000) as part of the integrity constraint. The process of back-chaining leads to the following sufficient tests:

1. project(r2 pid) = emptyset,and
2. r2 = emptyset.
3. not(lessp(pid(a) 1000)).

4. Detection of Insecurity

4.1 Modelling the User

We view the user as a deductive system $\mathcal{U}$ equipped with a finite set of axioms (Horn clauses with equality) and a sound but incomplete proof procedure. The axioms consist of rewrite rules defining the basic data types and the integrity constraints, and also rules if the database in question is a deductive database or an expert database. Following standard notation, we will write $\vdash_{\mathcal{U}}$ p to mean that the user is able to deduce the sentence (a formula with no free variables) p from the axioms, and we will write (q $\vdash_{\mathcal{U}}$ p) if the user can deduce p by assuming q in addition to the axioms.

At a given moment, let us assume the database to be in state s_0. The contents of the database may be expressed as ground (terms with no variables are called *ground*) predicate terms. The database, i.e., the set of these predicate terms is a model $\mathcal{D}$ of the axioms (the same axioms that are known to the user). However the user may or may not be aware of any of these ground terms — here we assume that in state s_0 the user is *not* aware of any of these ground terms (except those entries in the database that are derivable from the axioms/rules). We will write $\models_{\mathcal{D}}$ p to mean that p is a sentence that holds in the model $\mathcal{D}$, i.e., is true in the database. We should really indicate the state of the database and write $\models^{s_0}_{\mathcal{D}}$ p.

Now the database is capable of making transitions from state s_i to s_{i+1} as a result of the success of transaction T_j. If at state s_0, the user runs a transaction (that is guaranteed

to respect the integrity constraints if they were true to start with) with precondition p (a formula with free variables), then the user can add the sentence p(i) (i being the inputs to the transaction) to $\mathcal{U}$ (in deductive style we will write $\vdash^{s_0}_{\mathcal{U}} p(i)$); if not successful, (not p(i)) can be added instead. For this paper, we will concentrate on additions to the knowledge of the user *about state* s_0 and hence for clarity, we will simply use the notation $\vdash$ in place of $\vdash^{s_0}_{\mathcal{U}}$. Similarly we will use $\models$ instead of $\models^{s_0}_{\mathcal{D}}$.

In addition to learning that preconditions hold by successfully executing a transaction, the user can learn from the integrity constraints as postconditions of a successful transaction. By repeating a transaction, the user may incrementally extend his/her knowledge about the state s_0. For example, consider a database entity x that satisfies a constraint predicate q (i.e., q(x) = true). Let a transaction update x by applying function f to it (and nothing else). If this is successful, the sentence $q(f(x))$ is true in state s_0, and if iterated n times, then the formula $q(f^n(x))$ is also true in state s_0. We will illustrate later an application that uses this kind of knowledge increase.

4.2 Specifying secrets

Secrets are specified as formulas with free variables (these variable names are prefixed with an asterisk). The set $\mathcal{S}$ of such formulas defines the set of secrets.

For example, in a relation r1, let the instances of a key attribute c1 be a secret. This is expressed by the formula $s \in \mathcal{S}$:

s = for-some(project(r1 c1) p)
where p(x) $\equiv$ (x = *val)

which has the free variable *val. If the user can establish in $\mathcal{U}$ the truth of the above formula (or some other formula $s \in \mathcal{S}$) by obtaining an answer substitution (binding *val to a term with no free variables) that makes the above formula true, the secret is revealed.

4.3 Orderings on Formulas

We wish to demonstrate a structure on the universe of specifiable secrets. We will show that there is an equivalence class on secret formulas based on inference. This notion is useful since it formally captures the idea that some secrets are related in that if one is broken into, each of the others is violated as well, and also because it cuts the work needed to detect revelation since the specification set can be trimmed by including only one member from each partition generated by the equivalence relation in $\mathcal{S}$. Then we show a partial order that captures the structure existing among some secrets in which revelation of one leads to revelation of the other but not necessarily vice-versa. We will show later how this is useful in capturing the notion of partial inference.

We start by introducing an ordering on formulas — the ordering being defined by logical implication.

4.3.1 Ordering on Predicates

We first define an ordering on predicate expressions. The equivalence relation on closed formulas (i.e., predicate expressions with no free variables) is straightforward. Two closed formulas p and q are equivalent when p $\Rightarrow$ q and q $\Rightarrow$ p. Now we define a relation $\geq_*$ on the partition generated by this equivalence class. Define
$q \geq_* p \leftrightarrow q \Rightarrow p$.
As usual, $q >_* p$ iff $q \Rightarrow p$ but $p \not\Rightarrow q$.

Claim: $\geq_*$ is a partial order.

Proof: omitted. □

4.3.2 Ordering on Secrets

Let a secret be specified by a formula s having n free variables x_1, ..., x_n and written $s(\overline{x})$. Given a secret, an *answer substitution* is defined as a set of ordered pairs, the first of each pair being a variable (here belonging to the set of free variables) and the second a term, such that there is only one ordered pair with a given variable as first element. Given a secret $s(\overline{x})$, call an answer substitution θ *total* if each variable x_i in $\overline{x}$ appears as a first element of a pair in θ. The application of an answer substitution θ to a secret $s(\overline{x})$ is denoted as $s(\overline{x}).\theta$. Clearly $s(\overline{x}).\theta$ is a closed formula (i.e., one without free variables) if θ is total for $s(\overline{x})$. From now we will assume that answer substitutions are total for the relevant formula expressing a secret. We will often use the term "solution" in place of total answer substitution for brevity.

Now let us define the equivalence of two secrets. Two secrets $s(\overline{x})$ and $t(\overline{y})$ are said to be *equivalent*($\sim$):
$s(\overline{x}) \sim t(\overline{y}) \leftrightarrow$
user-derives($s(\overline{x})$, $t(\overline{y})$) $\wedge$ user-derives($t(\overline{y})$, $s(\overline{x})$)
where user-derives($s(\overline{x})$, $t(\overline{y})$) =
$\forall\phi \ \exists\theta$: $(s(\overline{x}), \phi) \hookrightarrow (t(\overline{y}), \theta)$,
and $\hookrightarrow$ is defined as a relation between pairs of secrets and solutions by
$((s(\overline{x}), \phi) \hookrightarrow (t(\overline{y}), \theta)) \leftrightarrow [\ (\models s(\overline{x}).\phi) \Rightarrow (s(\overline{x}).\phi \vdash t(\overline{y}).\theta)]$

If for ϕ a solution to s, a solution θ can be derived for t, we say that the pair (s,ϕ) *leads to a solution* pair (t,θ), or using the relation $\hookrightarrow$ we write $(s,\phi)\hookrightarrow(t,\theta)$. The predicate user-derives says that for *any solution* ϕ to s, a solution θ can be derived for t. Intuitively, if it is possible to derive a solution to t for every solution to s and vice-versa, the two secrets are equivalent. Consider for example the two secret formulas even(*x) and odd(*x) where *x has type number. If the following rules are known about these predicates:
even(x) $\Rightarrow$ even(x+2), and
even(x) $\Rightarrow$ odd(x+1),
then even(*x) $\sim$ odd(*x); these two predicates being interdependent and mutually definable, it is intuitively clear that these two secrets should indeed be equivalent.

Observation 1: If $(s,\phi) \hookrightarrow (t,\theta)$ and $(t,\theta) \hookrightarrow (u,\alpha)$, then $(s,\phi) \hookrightarrow (u,\alpha)$.

Proof: Let $(s,\phi) \hookrightarrow (t,\theta)$,
i.e., $[\ (\models s(\overline{x}).\phi) \Rightarrow (s(\overline{x}).\phi \vdash t(\overline{y}).\theta)]$
Assuming $s(\overline{x}).\phi$, $t(\overline{y}).\theta$ must be derivable. But $\mathcal{U}$ is sound. Hence it follows that $\models t(\overline{y}).\theta$
Since $(t,\theta) \hookrightarrow (u,\alpha)$, it follows that
$[\ (\models t(\overline{y}).\theta) \Rightarrow (t(\overline{y}).\theta \vdash u(\overline{z}).\alpha)]$
By Modus Ponens, it follows that $(t(\overline{y}).\theta \vdash u(\overline{z}).\alpha$
Hence we have $[\ (\models s(\overline{x}).\phi) \Rightarrow (s(\overline{x}).\phi \vdash u(\overline{z}).\alpha)]$
i.e., $(s,\phi) \hookrightarrow (u,\alpha)$. []

Observation 2: If user-derives(s,t) and user-derives(t,u) then user-derives(s,u).

Proof: Let ϕ_0 be any solution to s.
Since user-derives(s,t), i.e., $\forall\ \phi \exists\ \theta$: $(s,\phi) \hookrightarrow (t,\theta)$,
we have $\exists\ \theta_0$ such that $(s,\phi_0) \hookrightarrow (t,\theta_0)$.
Since user-derives(t,u) i.e., $\forall\ \alpha \exists\ \beta$: $(t,\alpha) \hookrightarrow (u,\beta)$,
we have $\exists\ \beta_0$ such that $(t,\theta_0) \hookrightarrow (u,\beta_0)$.
By Observation 1, it follows that $(s,\phi_0) \hookrightarrow (u,\beta_0)$.
i.e., $\forall\ \phi \exists\ \beta$: $(s,\phi) \hookrightarrow (u,\beta)$,
i.e., user-derives(s,u). []

Claim: $\sim$ is an equivalence class.

Proof: omitted. []

Now we will define an ordering on secrets. Let $\mathcal{S}$ be the set of secrets for the given database, and $S = \mathcal{S}/\sim$, i.e., the set $\mathcal{S}$ under the quotient of $\sim$. and let [s] denote the partition containg s.

Given [s],[t] $\in$ S, [s] $\geq_s$ [t] if for s $\in$ [s], t $\in$ [t] user-derives(s, t). In other words s is a *stronger* secret than t if whenever a solution to s is found, one to t is also deducible.

Claim: $\geq_s$ is a partial order.

Proof: omitted. []

An ordering on answer substitutions is also induced. Two substitutions θ and ψ total for a secret s are equivalent:
$\theta \simeq \psi \ \leftrightarrow\ ((s, \theta) \hookrightarrow (s, \psi)) \wedge ((s, \psi) \hookrightarrow (s, \theta))$.
In other words, whenever θ is a solution to s, so is ψ and vice-versa. For example, consider a secret s(*n,*m) where it is known that:
prime(a) $\wedge$ prime(b) $\wedge$ s(a,v) $\Rightarrow$ s(b,v)
In other words, s is a relation in which all prime first arguments have the same second argument. Now if θ returns the solution ((n 3) (m 55)), i.e., {n$\leftarrow$3, m$\leftarrow$55}, and ψ returns ((n 7) (m 55)), i.e., {n$\leftarrow$7, m$\leftarrow$55}, then the two answers are equivalent. Now in the equivalence class of these bindings, two bindings are ordered:
$[\psi] \succeq [\ \phi\ ]$ if $\psi \in [\ \psi\ ]$, $\phi \in [\ \phi\ ]$ $(s, \psi) \hookrightarrow (s, \phi)$
where both answers are total for a secret s. In other words, ψ is a *better* answer than ϕ if whenever ψ is a solution, ϕ also is. For example, in the above example with even(*x), the solution *x$\leftarrow$2 is a (strictly) better solution than *x$\leftarrow$4.

Claim: $\succeq$ is a partial order.

Proof: omitted. []

4.4 Relationship between Integrity Constraints and Secrets

Integrity constraints expressing invariants of the modelled world can be assumed to be known by anyone. When a transaction is run successfully, it is known that the subsequent state of the database satisfies the integrity constraints. Also when a certified transaction fails, it can be inferred that the preconditions failed with the input it was given. Thus transaction successes and failures give a view of the database in the light of the integrity constraints and therefore potentially of the secrets.

4.5 Checking for Revelation

Checking for revelation of secrets is an exercise in logic programming in which an attempt is made to find bindings for the free variables in the secret formulas. With the secret in question as the goal, the logic program attempts to find bindings for the free variables using a set of formulas as the program. The set of formulas comprises the integrity constraints, the precondition of the transaction, and the type-predicates that the input must satisfy. If the transaction takes an input a of type t, then a is treated as a constant, with the ground axiom type-t(a) assumed to hold. Note that the traditional logic programming approach to databases differs from ours primarily in that we do not use the ground facts in the database. Now let us discuss how our system is designed to check for revelation. This description is followed by an example.

First, to discover whether or not the integrity constraints reveal secrets by themselves, we check to see if any of the secrets are derivable (in the sense explained) from the integrity constraints alone. In this case the logic program does not include a transaction's preconditions or type predicates — only the integrity constraints are assumed as truths as each secret is tried as a goal. If this is successful, it means that the integrity constraints "contain" some of the secrets.

Next, transactions are analysed for their secret disclosure potential. As noted above, once a transaction runs, the user learns that the preconditions to the transactions must be true. During the safety proof of a transaction (certification of nonviolation of integrity constraints), tests are generated that are then added by the system designer as preconditions. Even if the user is not aware of these tests, it must be assumed that he/she will be able to infer these from his/her knowledge of the integrity constraints. The question is, "Can the user deduce a secret after a transaction is run?" To check this, the prover takes the role of the user, assumes the integrity constraints and the above tests as truths, and attempts to find an answer substitution for the secret. Note again that no actual data in the database

needs to be known at his stage. All input to the transaction may be assumed to be constants and all semantic knowledge in an expert database (Horn clause rules perhaps) may be used.

Further, recall that during test generation, cheaper (typically with better run-time performance) tests which are only sufficient are also generated. In case the transaction is found to be insecure, these sufficient tests are examined at the request of the designer. Some of these tests may turn out to be secure in which case the system designer has the option of including the sufficient test for the version of the transaction that the low-classification user may run. Thus the suggested sufficient tests have a utility in terms of security enhancement as well. (Of course, they will preclude the transaction's being run in certain safe cases for the user in question, but that may be acceptable to the designer.)

Thus, at this stage the designer may learn that though safe in terms of meeting integrity constraints, the transaction is potentially insecure. There are several steps he/she may take at this stage — the most obvious one being a change in the transaction. Other steps include reclassification of users, and a change in the schema (for example a change in key attributes or a change in the integrity constraints). All this depends on the ingenuity of the system designer and our system makes no effort in that direction.

We now illustrate the above procedure using simple examples.

Let us assume that the formula s expressed in the last section is indeed a secret. Assume also that there is a relation r2 which has no associated secrets, but is related to r1 through an integrity constraint IC where
IC = contains(project(r1 c1) project(r2 c2))
Now consider a transaction that simply inserts an element a into r2. This transaction may be run by a user at a lower level of classifiction than that of r1 and so any knowledge gleaned by that user about r1 could be damaging. The safety theorem prover that attempts to certify that the transaction meets the integrity constraints detects a necessary and sufficient test

member(c2(a) project(r1 c1)), or a.c2 $\in$ project(r1,c1).

To check for secret revelation, the prover takes the role of the user, assumes the integrity constraints and the above test as truths[3], and attempts to find an answer substitution for the secret.

First, using a rewrite rule, the secret is rewritten to

member(*val project(r1 c1)), or *val $\in$ project(r1,c1)

and then the logic programming technique leads to a binding *val$\leftarrow$c2(a), i.e., the c2-component of a, the input to the transaction under consideration. This illustrates how the successful running of a transaction can reveal the secret and how this fact may be detected during transaction/system design.

[3]if there is more than one precondition, their conjunction is assumed true.

Further, recall that during test generation, sufficient tests are also generated for better performance in the sense that it is cheaper in terms of run-time testing. For the above example, a *sufficient test*:
t1 = member(c2(a) project(r2 c2))
is generated. Although t1 will rule out the transaction being run under certain conditions, the original test can be factored into a cheap test t1 and a more expensive test
t2 = member(c2(a) project(r1 c1)) $\wedge$
not(member(c2 (a) project(r2 c2)))
It is found that t1 does not reveal a secret but t2 does. In other words, one sufficient test (i.e, t1) is secure, whereas another (i.e., t2) is insecure.

4.6 Partial Inference and Iteration of Transaction

Morgenstern [Morgenstern 87] explains the need for taking into account *partial* inferences. Partial inferences are said to be made about an entity y if the user's uncertainty in y's value is reduced by his/her knowledge of another entity x. He cites the example of the reduction in the uncertainty about the zip code if the area code part of the telephone number is given.

Here, we attempt to take into account this notion of partial inference especially when secrets involve equalities. We will also show how repeated executions of a transaction can increase a user's partial inference power. In case this sequence of transaction executions terminate owing to a violation of an integrity constraint, the knowledge of when this limit is reached may sharpen the user's partial inference.

First secrets involving equalities are marked as candidates for partial inference. Let s be such a secret of the form (d = *v). The system examines the type of the term d — if it is of type number or an enumerated type, or a user-defined type with a total order, the system can offer the designer a sequence of progressively weaker formulas say s', s'', such that s $>_s$ s' $>_s$ s''. Selection of one of the formulas, say s', indicates that detection of this would constitute an unacceptable partial inference.

Now suppose it is found that an integrity constraint q returns bindings for s''. This in itself is not worrisome since that is not a powerful enough inference, but let there be a transaction with a precondition p that refines q. This would mean that by running this transaction, a user may know more about some entity than was apriori evident from the integrity constraint alone. It could well be that by repeatedly running this transaction, it would be possible to refine this knowledge even further and eventually reveal s'. We will show that such indeed is the case. We will examine a sequence in which a transaction is repeated until an integrity constraint is violated. The following lemmas will make this more precise.

Definition: q *directly reveals* s *through* θ if q contains no free variables and can be rewritten to r, i.e., q $\Leftrightarrow$ r, such that r unifies with s returning an answer substitution θ.

Definition: p *reveals* s *through* θ if p $\Rightarrow$ q, and q directly reveals s through θ.

Definition: p *(directly) reveals* s if for some θ, p (directly) reveals s through θ.

In other words, q directly reveals s through θ if q can be eventually rewritten (using two-way equality rewrite-lemmas) to a formula r such that the latter unifies with s using θ. We drop the qualifier "directly" if q is allowed to rewrite to r using one-way rewrites, i.e., q implies r through a chain of implications.

The two following lemmas state that if q offers a solution to a secret s, and p is a formula that implies q, then knowing p one knows at least as much as knowing q (Lemma 1), and the solution through p could be more informative (Lemma 2).

Lemma 1: *If* $\models p$, $\models q$, $p \geq_* q$, *and q reveals s through* θ, *then p reveals s through* θ.

Proof: Since $p \geq_* q$, we have $p \Rightarrow q$; since q reveals s through θ, we must have $q \Rightarrow r$, $r \Leftrightarrow s.\theta$. Hence it follows that $p \Rightarrow r$, $r \Leftrightarrow s.\theta$, i.e., p reveals s through θ.[]

Lemma 2: *If* $\models q$, $\models p$, *q directly reveals s through* θ, $p \geq_* q$ *(respectively* $p >_* q$*), and p directly reveals s through* ψ, *then* $\psi \succeq \theta$ *(respectively* $\psi \succ \theta$*).*

Proof: Since q reveals $s(\overline{v})$ through θ, q must be equivalent (through equality rewrites) to some formula q_0 such that q_0 unifies with $s(\overline{v})$ returning $\theta=\{\overline{v}\leftarrow\overline{z}\}$, where $\overline{z}$ is a vector of constant terms. Now let us denote by $q_0(\overline{y})$ the term we get from q_0 by replacing all occurrences of $\overline{z}$ by variables from $\overline{y}$. Then we have $q \Leftrightarrow q_0(\overline{z})$.

The case $p \equiv q$ is trivial, so we will focus on $p >_* q$. Since $p >_* q$, $p \Rightarrow q$, and $q \not\Rightarrow p$. Since p directly reveals s through ψ, $p \Leftrightarrow q_0(\overline{c})$, where $\overline{c}$ is a vector of constants and $\psi=\{\overline{v}\leftarrow\overline{c}\}$, Then we must have $q \not\Rightarrow q_0(\overline{c})$, and hence, $q_0(\overline{z}) \not\Rightarrow q_0(\overline{c})$, i.e., $\theta \not\succeq \psi$.

Also we must have, $q_0(\overline{c}) \Rightarrow q_0(\overline{z})$, i.e., $\psi \succeq \psi$. Hence $\psi \succ \theta$. Referring to the diagram below,

p	$\Rightarrow$	q
$\Updownarrow$		$\Updownarrow$
$q_0(\overline{c})=s.\psi$	?	$q_0(\overline{z})=s.\theta$

the ?-entry can only be filled in with $\Rightarrow$, not with $\Leftarrow$.[]

Before we state the main theorem of this section, let us have one definition.

Definition: Two (secret) formulas s_0, $s_1 \in \mathcal{S}$ are said to be *complementary* if s_0 is of the form (not s_1') where s_1' is s_1 apart from a renaming of free variables.

Observation: If two (secret) formulas s_0, $s_1 \in \mathcal{S}$ are complementary, then whenever some p directly reveals s_0, $\sim$p must directly reveal s_1.

Theorem 1: *If* $s, s_0, s_1, s_2 \in \mathcal{S}$, *such that* s_0, s_1 *are complementary,* $s = s_0 \wedge s_1 \wedge s_2$, *and* s_2 *contains no free variables other than those in* s_0 *and* s_1, *then if formulas* p_0, ..., p_n *are such that* $p_{i+1} \geq_* p_i$, *(respectively* $>_*$, $0 \leq i \leq n$*), and* $\vdash p_0$, ..., $\vdash p_n$, *and* $\vdash \sim p_{n+1}$, *and* p_i *directly reveals* s_0 *through* ψ_i, *(*$0 \leq i \leq n+1$*), then*
(i) $\psi_n \succeq \psi_{n-1} \succeq \ldots \succeq \psi_0$ *(respectively* $\succ$ *),*
(ii) $\sim p_{n+1}$ *reveals* s_1 *(through* α*, say), and*
(iii) $\{p_0, \ldots, p_{n+1}\}$ *reveals s provided* $s_2.(\psi_n \cup \alpha)$ *holds.*

Proof: Since $\vdash p_0$, we have $\models p_0$. Since p_0 directly reveals s_0, and $p_{i+1} \geq_* p_i$, $(0 \leq i \leq n)$, by Lemma 1 we have p_i reveals s_0, for $0 \leq i \leq n$. In fact, we are given the stronger condition that p_i directly reveals s_0. By Lemma 2, we have $\psi_n \succeq \psi_{n-1} \succeq \ldots \succeq \psi_0$ (respectively $\succ$).

Since s_0 and s_1 are complementary, and p_{n+1} directly reveals s_0, it follows from the above observation that $\sim p_{n+1}$ directly reveals s_1.

We have $s_0.\psi_n$ = true, and $s_1.\alpha$= true. Now $(\psi_n \cup \alpha)$ is well defined provided there is no conflict in assignments to variables. If well defined, s_0, s_1 are both true under the answer substitution $(\psi_n \cup \alpha)$. Hence s would be true provided $s_2.(\psi_n \cup \alpha)$ is true.[]

This theorem says that if a secret is of the form above, and the component s_0 can be solved by formula p_0, and if the user is able to derive progressively stronger formulas p_0 ... p_n (with more informative answers by Lemma 1), the process terminating with the formula $\sim p_{n+1}$, then the secret s is revealed with the last two answer substitutions giving the condition through s2. Note that s_2 has no free variables other than those in s_0 and s_1, and hence $s_2.(\psi_n \cup \alpha)$ is a ground term and it can be viewed as a condition on the inputs to the transaction that allow the secret to be revealed.

To illustrate the use of the above theorem, consider the schema and transaction in figures 1 and 2. Assume there is an integrity constraint:
for-all(companies, p_0) where
$p_0(x) \equiv$ (totsal(x) $\leq$ maxsalary) meaning that the total salary for any company is bounded above. Let the total salary for each company also be a secret. Since the total salary for a company is also equal to the sum of the salaries for the company in the placements relation, the formula for the secret takes the form:

s = (sum(salary r1) = *val)

where *val is a variable, and r1 is the selection of one company from the placements relation. Now when the hire transaction inserts a tuple in the placements relation, and increments the totsal field of the companies relation by sal, the input to the transaction, the integrity constraint above must hold for the transaction to be successful. The secret can be revealed by iterating the hire transaction, as we will show using Theorem 1.

As an aside, this situation would also occur if there are two relations r1 and r2 with r2 contained within r1, salary

being a common attribute; The total salary computed by summing the salaries of all the tuples in relation r1 is a secret (as expressed by a formula such as s above); and the user runs a transaction that inserts a tuple into the smaller relation r2.

Now let us examine how the theorem can be applied. First, s is weakened to the formula
$s' = (\text{sum(salary r1)} \leq *v1) \wedge (\text{sum(salary r1)} > *v2) \wedge ((*v1 - *v2) \leq d)$.
In other words, the formula s' with two free variables *v1 and *v2 states that the sum of salaries is bounded on two sides by *v1 and *v2 and the bounds are less than d apart, where d is a constant. Note that s' is of the form $s_0 \wedge s_1 \wedge s_2$, where s_0 and s_1 are complementary.

Let us replace the term sum(r1 salary) by totsal and consider p_0 to be to be an integrity constraint:
(totsal $\leq$ maxsalary). Note that p_0 directly reveals s_0, and s_0 can be viewed as s'', a weaker version of s'. Now if the transaction runs to completion, then the user knows that the formula
$p_1 = ((\text{sal} + \text{totsal}) \leq \text{maxsalary})$,
is true and that its complement is true if it does not. Note that $p_1 >. p_0$ if sal $\neq 0$.

Now consider the iteration of this transaction. It turns out that $p_{i+1} >. p_i$ if the (i+1)th transaction runs successfully and the user is able to deduce p_{i+1}, i.e., $\vdash p_{i+1}$. Next we check to see if for some positive integer n it is possible for the transaction to be successful n times and then fail. One way of showing this is to prove the termination of the recursive function g that takes parameters x, p, and f and is defined by

```
termination-test (x p f) =
    if p (x)
        then termination-test (f(x) p f)
        else x
```

where we have:
$p(y) \equiv (y \leq \text{maxsalary})$,
$f(z) \equiv (\text{sal} + z)$.
Our prover being based on the Boyer-Moore theorem-prover, it can certify functions as provably terminating if it can find a measure function M that reduces on every recursive call. It must prove that
$p(x) \Rightarrow \text{lessp}(M(f(x))\ (M\ x))$.
Detecting such a measure function through inspection of f is not easy and we know of no general method. With the help of heuristics and help from a system expert, it is possible to devise a handful of measure functions that work for most simple update functions. In this case the measure function that works is given by
M(x) = difference(maxsalary x).
provided that sal $\neq 0$.

At this point the prover has established that if the transaction is repeated, the sequence must terminate for some n. From Lemma 2, it follows that it is enough to concentrate on $p(f^n(\text{totsal}))$ and $\sim p(f^{n+1}(\text{totsal}))$ the last two of the progressively more informative formulas which give solutions for s_0 and s_1 respectively. These get rewritten through the use of rewrite rules pushing the composition of the addition function as multiplication into:
totsal $\leq$ (maxsalary - (n * sal)), and
totsal $\geq$ (maxsalary - (add1(n) * sal)) respectively. These give solutions for s_0 and s_1 as promised by the theorem and s_2 yields the condition for revelation of s':
(sal $\leq$ d). In other words, if the input (to the transaction) sal is chosen to be d or less, then the secret s' is revealed after n iterations.

We have shown by this example, how by iterating a transaction, it is possible for the user to sharpen his/her knowledge about a secret, i.e., infer its value partially, and how it is possible to detect this in some cases.

5. Conclusion and Future Work

We have taken a deductive approach to the problem of security enforcement and have shown how secrets can be specified formally and how such a specification leads to an interesting and useful structure among secrets — a structure based on the notion of equivalence and partial ordering. Our focus has been on a diligent user who can observe the success/failure of transactions and uncover secrets by making inferences based on the rich semantics offered by the database integrity constraints and rules in a deductive/expert data/knowledge base. The instrument we use is a theorem prover and all analysis is done at compilation time. We have shown also that when partial inference is not tolerable, that too can be specified. Our last theorem shows how by repeating transactions, incremental partial inference may lead to complete revelation, and this case can be detected. This method of secrecy specification and detection of compromise needs a theorem prover, but is not otherwise restricted to the ADABTPL environment.

In the ADABTPL environment, database transactions are analysed in the presence of integrity constraints and feedback is given to the designer about whether the transaction respects those constraints, possible tests that could be incorporated to correct for violations, if any, and some simple postconditions. The extension proposed in this paper would give feedback regarding security from an analysis of the transaction, but apart from some simple hand-testing, has not been implemented at the moment of writing.

REFERENCES

[Bancilhon and Spyratos 77] Bancilhon, F.M., and Spyratos, N., "Protection of information in relational data bases", Proceedings of the 3rd Intl. Conf. on Very Large Data Bases, Tokyo, Japan, 1977. pp. 494-500.

[Boyer and Moore 79] Boyer, R. S. and Moore, J. S. *A Computational logic*, Acedemic Press, New York, 1979.

[Casanova and Bernstein 80] Casanova, M.A., and Bernstein, P.A., "A Formal System for Reasoning about Programs Accessing a Relational Databse", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, July 1980.

[Denning et al. 79] Denning, D.E., Denning, P.J., and Schwartz, M.D., "The Tracker: A Threat to Statistical Database Security", ACM Transactions on Database Systems, Vol 4, No. 1, 1979. pp. 76-79.

[Dobkin et al. 79] Dobkin, D., Jones, A.K., and Lipton, R.J., "Secure Databases: Protection against User Influence", ACM Transactions on Database Systems, Vol 4, No 1. pp. 97-106.

[Gardarin and Melkanoff 79] Gardarin, G. and Melkanoff, M. "Proving the Consistency of Database Transactions", Proceedings of the 5th International Conference on Very Large Databases, Rio de Janiero, Brazil, 1979.

[Fernandez et al. 81] Fernandez, E.B., Summers, R.C., and Wood, C., "*Database Security and Integrity*", Addison-Wesley, Mass, 1981.

[Henschen et al. 84] Henschen, L.J., McCune, W.W., and Naqvi, S.A., "Compiling Constraint Checking Programs from First-Order Formulas", in *Advances in Database Theory*, Vol 2, Edited by Gallaire, Minker, and Nicolas. Plenum Press, New York, 1984.

[Nicolas 82] Nicolas, J.M., "Logic for Improving Integrity Checking in Relational Databases", Acta Informatica, Vol. 18, No. 3, Dec. 1982. pp. 227-254.

[Sicherman et al. 83] Sicherman, G.L., de Jonge, W., and van de Riet, R.P. "Answering Queries without Revealing Secrets", ACM Transactions on Database Systems, v.8, no. 1, March 1983. pp.41-59.

[Morgenstern 87] Morgenstern, M., "Security and Inference in Multi-level Database and Knowledge-Base Systems", Proceedings of the ACM-SIGMOD International Conference on Management of Data. San Francisco, California. May, 1987. pp. 35-373.

[Sheard and Stemple 86] Sheard, T. and Stemple, D., "Automatic Verification of Database Transaction Safety",COINS Technical Report 86-30, University of Massachusetts, Amherst, 1986.

[Stemple and Sheard 85] Stemple, D. and Sheard, T. "Database Theory for Supporting Specification-Based Database System Development", Proceedings of the 8th International Conference on Software Engineering. Imperial College, London, U.K. August, 1985. pp. 43-49.

[Stemple and Sheard 87] Stemple, D., and Sheard, T., "The Construction and Calculus of Types for Database Systems", Proceedings of the Workshop on Database Programming Languages, Roscoff, Finistere, France, September, 1987.

[Stemple et.al. 86] Stemple, D., Sheard, T., and Bunker, R.E., "Incorporating Theory into Database System Development", Information Processing and Management, Vol 22, No. 4, 1986. pp. 317-330.

[Stemple et al. 87] Stemple, D., Mazumdar, S., and Sheard, T. "On the Modes and Meaning of Feedback to Transaction Designers", Proceedings of the ACM-SIGMOD International Conference on Management of Data. San Francisco, California. May, 1987. pp. 374-386.