1

Compile-time Flow analysis of Transactions and Methods in Object-Oriented Databases

Masha Gendler-Fishman and Ehud Gudes
Department of Mathematics and Computer Science
Ben-Gurion University, Beer-Sheva, Israel.
[1] *e-mail: masha,ehud@cs.bgu.ac.il*

Abstract

Methods are an important characteristics of Object-oriented databases, Previous models for Discretionary access-control in OO databases have considered policies for Methods and Inheritance. However, discretionary authorization models do not provide the high assurance required in systems where Information flow is considered a problem. Mandatory models can solve the problem but usually they are too rigid for commercial applications. Therefore discretionary, information-flow control models are needed, especially when transactions and methods invocations are considered.

This paper first reviews existing security models for object-oriented databases with and without information-flow control. These models rely on the run-time checks of every message transferred in the system. This paper uses a **compile-time** approach and presents algorithms for flow control which are applied at Rule-administration and Compile times, thus saving considerable run-time overhead. Special emphasis is put on the Flow-analysis of Methods and the transactions invoking them.

keywords

Information Flow, Discretionary, Object-oriented, Methods, Compile-time analysis

[1]this paper was partiall supported by GIF grant I-420-041.06

Database Security XI T.Y. Lin and Shelly Qian (Eds)

1 Introduction

Security is an important topic for Databases in general and for Object-oriented databases (OODB) in particular [Kim(90), Kemper(94)]. In general, authorization mechanisms provided by commercial DBMS are *discretionary*, that is, the grant of authorizations on an object to other subjects is at the discretion of the object administrator.

The main drawback of discretionary access control is that it does not provide a real assurance on the satisfaction of the protection requirements, since discretionary policies do not impose any restriction on the usage of information by a subject who has obtained it legally. For example, a subject who is able to read data can pass it to other subjects not authorized to read it. This weakness makes discretionary policies vulnerable to attacks from "Trojan horses" embedded in programs. [2] Access control in *mandatory* protection systems is based on the "no read-up" and "no write-down" principles [Castano(95)]. Satisfaction of these principles prevents information stored in high-level objects to flow to lower level objects. The main drawback of mandatory policies is their rigidity which makes them unsuitable for many commercial environments.

There is the need of access control mechanism able to provide the flexibility of discretionary access control, and at the same time, the high assurance of mandatory access control. A first attempt to do it in the context of OODBs was made by Samarati et al [Samarati(97)]. The main problem with the model in [Samarati(97)] is that all the checks are done at *Run-time* which increases considerably the overhead in the system. Many DBMSs rely on protection which is checked at compile time! For example, Query modification in Ingres [Stonebraker(76)] or View-based mechanisms in System R [Griffth(76)] in Relational systems, or the model suggested by Fernandez et al [Fernandez(94)] for Object-oriented databases. In this paper we investigate the problem of insuring safe information flow for Object-Oriented databases by performing the checks at *Compile time* or at *Rule-definition time*, thus saving considerable overhead at run-time. A very important assumption of the present paper is that the run-time of the DBMS can be trusted. That it, if one composes its transactions only from Queries and Methods which were compiled under the control of the DBMS, one can trust their object-code

[2] the term "Trojan horse" is used here to refer to any illegal leakage of information, not necessarily a destructive one...

and the Run-time system which executes them. A similar assumption is made in View-based systems where views are kept after they are compiled and optimized [Griffth(76)].

In a recent paper by the same authors [Gudes(97)] we presented a simple Transactions model and algorithms to check for information-flow at compile-time for this transaction model. The limitations of the transaction model in [Gudes(97)] was that only the basic READ/WRITE methods were allowed, and no general methods. In this paper we extend the previous transactions model by allowing transactions to invoke any method (with or without parameters) and these methods may further invoke other methods. We put very few restrictions on the type of programming language and constructs used within methods. Using program-flow analysis techniques [Muchnick(81)], we are able to analyze the methods at compile-time when they are entered into the system, and complete the analysis at the time the transactions is compiled. Therefore, this process is very efficient since the compile-time analysis for methods is done only once (provided the method was not changed).

It is important to note that our algorithms provide an *upper-bound* for the information-flow problem. Since program-flow analysis methods cannot know the actual Run-time control-flow, they must consider all branches of an If, WHILE or CASE statements. Thus, when our algorithm reports on safe information-flow the safeness is assured, but when it reports on an unsafe information flow, it actually reports on only a potential unsafe information flow. If one is very concerned about "false" alarms, one can employ a Run-time method in these cases, such as the one in [Samarati(97)]. Another problem in [Samarati(97)] is that within a Method or Transaction all the information associated with a Read query is added into the overall run-time flow, regardless of whether there is an actual flow (between program statements) of this information into the objects which the Method writes. This problem is overcome in our model, because flows within program statements are analyzed within every method.

As this paper relies heavily on three previous papers [Samarati(97)], [Fernandez(94)], and [Gudes(97)] , these papers are first reviewed briefly in Section 2 and the definition of safe information flow is given. The generalized Transactions and Methods model is defined in Section 3 and the overall approach is explained. In Section 4 we present our compile-time analysis of Methods, and in Section 5 we present the Transactions analysis algorithm. Examples are given in both sections. Section 6 is

the Summary.

2 Background

2.1 Fernandez et. al

The first model [Fernandez(94)], assumes a simple discretionary Rules-based authorization. The model deals mainly with the impact of inheritance on security and enforces several inheritance-based policies. In following papers, policies were proposed for negative authorization, content-dependent restrictions, and for resolving conflicts between several implied authorizations (see [Larrondo(90)]). Another paper extended the basic model to include treatment of general methods [GalOz(93)] (As an example to the inheritance policies, consider the database in Figure 1. A rule giving a user Read access to all attributes of Student, implies also a Read access to Foreign Student's Social security number (SSN), but not to his/her Visa...)

Because of space problems we will not review this paper here. The most relevant point is the discussion of an Access Validation algorithm The validation algorithm is applied at *Compile-time* in that it works after the Query translator and its output is entered to the Optimizer and run-time system. The Access-validation algorithm accepts two major inputs:

- The original query after translation in form of a tree. This query is further extended using the inheritance hierarchy to something called *Authorization Tree (AT_yes).* (the AT_yes will be redefined in the next section, therefore we do not detail its structure here). Initialy, all the AT_yes's nodes are authorized. After the validation algorithm, the AT_yes contains only the nodes and the attributes to which access is allowed (see an example in Section 3.1).

- The rules which are relevant to this query are extracted from a tree called the *Security Graph* which is an extension of the AT_yes upwards and downwards to include all relevant rules.

The algorithm scans in parallel the query nodes and security graph nodes, applies the inheritance policies mentioned above and produces the final AT_yes which defines the allowed access.

2.2 Samarati et. al

The second model by Samarati et al. [Samarati(97)] describes a run-time architecture (Message filter) for checking for information flow. Again, for reasons of space we cannot describe the model in detail. The most important concepts are:

- **Transaction.** A transaction is the set of methods invocations caused by a user sending a message. The first message invokes a method which invokes other methods by sending messages to it and waiting for replies. The invoking method may in turn wait for the reply (synchronized) or deferred its waiting. A user executing a transaction is called the *Transaction initiator* .

- **Access lists.** There are several access lists associated with each object including
  RACL(o) - the list of users which can read from object o,
  WACL(o) - the list of users which can write into object o.

- **Information flow.** There exists a flow between O_i and O_j in a transaction if and only if a write or create method is executed on O_j, and that method had received information (via forward or backward transmission) on O_i. When a method A sends a message to another method B, then all the information which flowed into A is assumed to flow into B. Similarly, if a method A receives a reply from B, the information that flows into B is assumed to flow into A.

- **Safe Information flow.** Information flow is safe only if there is information flow from O_i to O_j and all users which can read O_j can also read O_i, i.e. RACL(O_j) is contained in RACL(O_i).
  To enforce only safe information flows, [Samarati(97)] suggests the construction of a *Message Filter* component which intercepts each and *every message* in the system. Using this information it is possible to enforce safe information flow and **disallow** transferring of information which may cause an unsafe flow (i.e an empty reply is returned in that case...)

Although the above algorithm is very general and works for various types of methods and executions, it requires the check and filtering of every message in the system. This is a considerable overhead! In the next sub-section, we discuss a simpler model, of Read/Write methods

only, but on which a compile-time algorithm based on [Fernandez(94)] is used.

2.3 Gendler & Gudes Model

The model by Gendler & Gudes [Gudes(97)] provides compile-time checking for information-flow which is based on the AT_yes idea of [Fernandez(94)]. The following concepts are used:

- **Object Model and Authorization Model.** Both models are similar to the ones in [Fernandez(94)]

- **Access Lists.** In [Fernandez(94)] the main administration structure was the *authorization rule* placed at special class/node in the object-hierarchy tree. For purposes of flow control we need to define also for each attribute a list of all users authorized to read it. We maintain the structure called *read access list*(RACL) containing the list of users who have read privileges to the attribute. The RACL of-course can be obtained using the inheritance policies mentioned above:

$$\text{RACL}(O.Attr) = \{u : (\exists\ O' | O \preceq O' \text{ and } \exists \text{ rule } (u, R, O'.Attr))$$

$$\wedge\ (\nexists\ O'' | O \preceq O'' \preceq O' \text{ and } \exists \text{ rule } (u, -R, O''.Attr))\}$$

  i.e. this list contains the users that the authorization rules permit them a read access to the attribute, either explicitly or via the inheritance policies specified above.

- **Authorization Tree.** Each query of the type above is validated against the initiator (U) authorization rules using the model and algorithm presented in [Fernandez(94)]. The result of such validation is the set of objects (classes) and their attributes which is authorized for this query. Basically, this set is a sub-tree of the query graph rooted at $O.Attr$ and is called *authorization tree*, denoted AT_yes$(u, A, O.Attr)$. In the sequel we will usually use the authorization-trees for Read access, and therefore denote them as: AT_yes$(u, O.Attr)$.

- **User Access Tree (UAT).** The set of attributes in the "entire database" that user u is allowed to access for reading is called *user*

access tree.

$$\text{UAT}(u) = \{(O.Attr) : u \in \text{RACL}(O.Attr)\}$$

The above UAT is computed from the data structure $RACL$, but, obviously, it is also true that

$$\text{UAT} = \cup_{\forall i,j} AT\_yes(O_i.Attr_j)$$

- **Common User Access Tree(CUAT).** We introduce a new measure for every attribute A_j: the intersection of UATs of all users who are permitted to read it. This intersection expresses the set of all attributes which is allowed to be read by all users who are allowed to read attribute A_j.

$$\text{CUAT}(O.Attr) = \bigcap_{\forall u \in \text{RACL}(O.Attr)} \text{UAT}(u)$$

As will be shown below, the CUAT is a critical component in computing safe information flow. Detailed algorithms for its computation were given in [Gudes(97)] .

- **Safe Information Flow.** We are now ready to define the criteria for safe information flow. Intuitively, we know that every read query after validation can only read the objects and attributes contained in that's query authorization tree. Therefore, the union of these trees expresses all the information to which this transaction has read access. We must make sure that the users who have access to the object into which this transaction writes, are allowed to access all the information that has flowed into the transaction upto the Write query.

Theorem 1 (Safe Information Flow) The information flow to the attribute $O_k.Attr_j$ caused by the write access $write(O_k.Attr_j, v)$ in a transaction, is safe, if and only if, the Common Users Access Tree of the attribute $O_k.Attr_j$ contains the union of the authorization trees of all the previous read queries.

$\bigcup_{i=1}^{j-1} \text{AT\_yes}(i) \subseteq \text{CUAT}(O_k.Attr_j) \iff$ the information flow to $O_k.Attr_j$ is safe

Proof see [Gudes(97)]. Intuitively, each object read by the transaction and potentially written into $O_k.Attr_j$, must be contained in the set of objects allowed Read access by all users who can read object $O_k.Attr_j$.

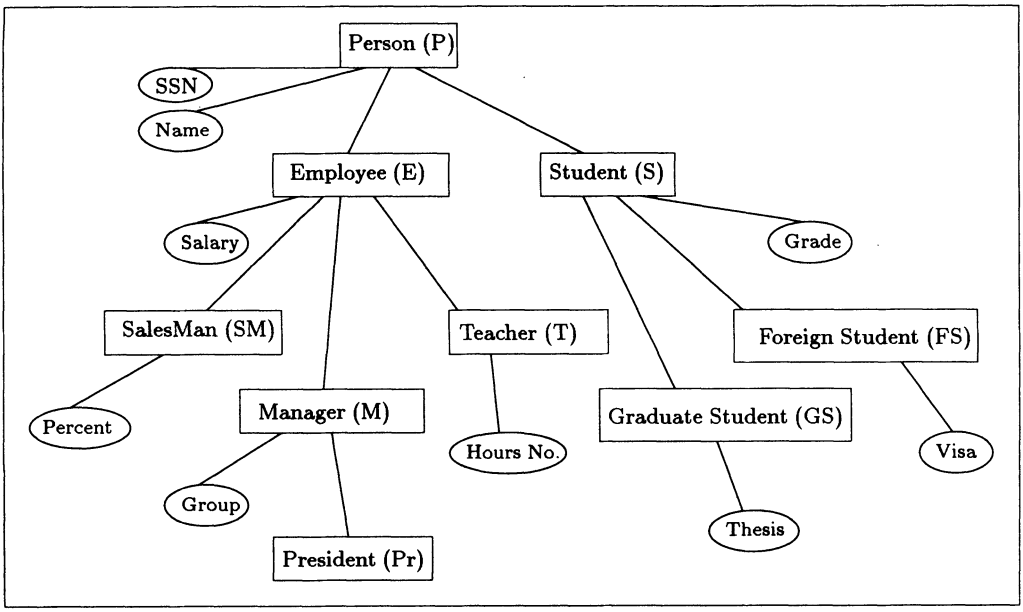

Figure 1: University database

3 The generalized Transactions & Methods Model

The transactions model presented in this section, is a generalization of the model in [Gudes(97)] discussed in above, in that a transaction may now call methods and pass to them parameters. Such a method may further call other methods and in return may return a value or an object to the calling method or transaction. Both the transaction and the method it calls may issue Reads and Writes to the database, therefore, information flow may occur within methods and it must be checked for. Formally, we define:

Transaction. A transaction consists of a sequence of Read or Write queries, or Method calls, where:

Read query. $val = read(O.Attr)$ where O is a database object/class , $Attr$ is an attribute and val is the variable that receives the result.

Write query. $write(O.Attr, val)$ where O and $Attr$ are as before and val is the value (or variable) to be written into the object attribute.

Method with return value. $val = meth(p_1, \ldots p_n)$ where $meth$ is the name of a called method, $p_1 \ldots p_n$ are the method's parameters (in

a form of single instances or AT_yes hierarchical trees) and *val* is the variable that receives the return-value.

We also assume without loss of generality that a transaction does not contain any control-flow statements (they can be embedded within a method) and therefore contain only calls to the methods of the above type. As explained in the introduction, the analysis of transactions is done in two phases:

1. The Method's compile-time phase
2. The Transaction's compile-time phase.

Each method is analyzed separately at the time it is compiled and stored in the database, and the analysis results are stored in specialized security-related tables. The transaction analysis phase uses the information in these tables at the time the transaction is compiled.

3.1 Analysis of methods

In this section we discuss the compile-time analysis of a single method. This analysis is done once when the method's code is inserted into the database, and the results are stored in several tables associated with the method. The results of this analysis are composed of three parts:

1. The information flow to objects accessed for writing from within the current method.
2. The information flow to the return value of the method (if one exists).
3. The forward information flow via the parameters to methods which are called from within the current method.

These results are stored in special database tables and are used in the next stages of the analysis. Since the actual parameters to the method are not known, we use "virtual" symbols in the tables. At transaction analysis time the real information flow is substituted for these virtual symbols.

The method analysis itself uses program-flow analysis techniques which are common in Compiler and program Optimization ([Aho(86), Muchnick(81), Denning(86)]). We assume that we have a parser that

can generate a syntax tree of the method, and the flow-analysis phase operates on this tree. We assume that the method is written in a programming language like Pascal or C (C++) with some restrictions. We assume that all variables in the methods are strongly-typed, i.e. pointers and memory management operators are either strongly typed too or forbidden. Another limitation is that there are no static or global variables, i.e all information between methods is passed via the parameters, and the **goto** statement is also forbidden. We also use the following notation:

1. FLOW(a), where a is an OODB object or an attribute of an object or a local variable of method - a list b of OODB objects, local variables and virtual symbols, such that there is a potential information flow to a, $b \rightsquigarrow a$.

2. IN stores the potential flow to the current block of statements ([Muchnick(81)]). [3]

3.2 Assignment statement

Let us begin with the simple case of assignment statement:

$$S \rightarrow \mathbf{id} = E$$

Assignment is the simplest way for information flow. All information from expression E flows into variable **id**. Note that syntax analysis of the expression E is required to find all variables (or functions) participating in the expression. We use the notation $a \in E$ to specify the variables involved in the expression E.

$$\text{FLOW}(\mathbf{id}) = \bigcup_{a \in E} \{\{a\} \cup \text{FLOW}(a)\} \cup \text{IN} \qquad (3.1)$$

We must include IN, if this statement is part of an IF or a LOOP block as explained below.

3.3 Block of statements

We define the relation *before* between two statements as follows: S_1 *before* $S_2 \Rightarrow$ if S_1 leads to an informati on flow FLOW_{S1} to variable x and S_2 leads

[3] we will use in the rest of the paper the term "information flow", although it should be clear that we mean essentially only "potential information flow"

to an information flow $x \rightsquigarrow y$ such that: $\text{FLOW}_{S1} \rightsquigarrow y$. Formally, S_1 *before* S_2 $\Rightarrow$ if $\exists$ FLOW(x) updated in S_1 - FLOW_{S1} and $\exists$ flow $x \rightsquigarrow y$ in S_2, such that $\text{FLOW}_{S1} \rightsquigarrow y$ exists, and statement S_1 is executed before S_2. Thus, the statement S_1 *must* be analyzed before S_2. Consider the following statements block:

$$S \rightarrow S_1;\ S_2; \ldots\ S_n$$

$$S_1\ before\ S_2\ before\ \ldots\ before\ S_n \tag{3.2}$$

We say that inside a block of statements there are *sequential flows*. The analysis of a block of statements must therefore pass sequentially thorough all the statements in the block. We shall see in following subsections another order of flows for loop statements.

3.4 Read/Write queries

A method may contain Read queries and Write queries. Such queries will cause information flow as follows. Assume a Read query : $a = read(O.Attr)$ then

$$\text{FLOW}(a) = O.Attr \cup \text{FLOW}(O.Attr) \cup \text{IN} \tag{3.3}$$

We say that as a consequence of a read query the information flows to a from $O.Attr$ as well as from all objects and variables that the information flowed from them to $O.Attr$ before the Read. For a Write query: $write(O.Attr, a)$ then

$$\text{FLOW}(O.Attr) = a \cup \text{FLOW}(a) \cup \text{IN} \tag{3.4}$$

The information flows to $O.Attr$ from a, as well as from all objects and variables that the information flowed from them to a and to the Write statement.

Now, for every Write query we must record the flow that was caused by the Write in a table entry, since later on we will analyze whether that flow was safe or not. Each compiled method will have its own WRITES table containing one entry for every Write query in that method. An entry in the WRITES table has the format: $(Obj, WrInFlow)$, where Obj is OODB object accessed for writing and $WrInFlow$ is set of OODB objects and virtual symbols that is contained in FLOW(a) (Note that local variables are not inserted into this table).

3.5 If statement

Consider the following example:

if $(a > 1)$
$b = 1;$
else $b = 2;$

There is no direct information flow from a to b in this example. However, it is possible after the execution of the code to draw a conclusion from the value of b about the value of a. So there is a potential information flow from the **if** condition to both the **then** and the **else** branches. (see [Denning(86)] for an extensive discussion of this example). Formally,

$$S \rightarrow \textbf{if}(E)\ S_1;\ \textbf{else}\ S_2;$$
$$\mathrm{IN}(S_1) = \mathrm{IN}(S_2) = \bigcup_{a \in E} \{\{a\} \cup \mathrm{FLOW}(a)\} \cup \mathrm{IN} \tag{3.5}$$

3.6 Loop Statement

Loop statement is a more complicated case than **if** statement. Consider the following example:

$x = 1;\ a = 1;$
for$(i = 0;\ i < n;\ i++)$
$x = x * a++;$

Again, there is no direct flow from n to x, but after the loop execution, x is equal to $n!$. Another problem is the repeating execution of the loop body:

while $(\ldots)$
$\{\ a = b;$
$b = c;$
$\ldots\ \}$

One-pass analysis finds the flows $b \rightsquigarrow a$, $c \rightsquigarrow b$. If the loop body is executed more than once, the flow $c \rightsquigarrow a$ also exists. In the previous analysis of a statements block, we only considered *sequential flows*. Loop statements have the property of *cyclic flow*, i.e succeeding statements also affect preceding ones. Thus, for loops a two pass analysis is needed. Formally,

$$S \rightarrow \textbf{while}(E)\ S_1$$

z	t
t	$\{z,\_\$1\}$

Table 1: Result of one pass analysis of loop

z	$\{t,\_\$1\}$
t	$\{z,\_\$1\}$

Table 2: Result FLOW table

$$\mathrm{IN}(S_1) = \bigcup_{a \in E} \{\{a\} \cup \mathrm{FLOW}(a)\} \cup \mathrm{IN}, \textbf{analyze\_twice}S_1 \qquad (3.6)$$

where **analyze_twice** is a command to the Analyzer to scan the statement twice. A similar analysis is done for a **for** statement.

3.7 Example

Let us consider the following example of a partial method's code, and analyze the flows occuring within it.

```
Copy(int x) { int z,t;
  z = 0; t = 1;
  while (t == 1)
  { z = z + 1;
    if (z == x)
      t = 0;   } }
```

At the beginning the flows to local variables z and t are equal to $\emptyset$. The result FLOW of the first pass of the analysis of the **while** statement is shown in the Table 1. The flow IN of the loop body is variable t, then analysis of the first assignment statements finds FLOW(z)=$\{t\}$. The analysis of the **if** statement finds FLOW(t)=$\{z,\_\$1\}$.

The complete FLOW table of the method is shown in Table 2. The second pass analysis of the **while** statement, finds the flow $\_\$1 \rightsquigarrow z$. It is essential because during execution of the loop the value of the parameter x is copied into the local variable z.

3.8 Method Calls

In this section we discuss the analysis of methods calls. Since every method is analyzed independently, separate tables are created for each method. One table, called the CALLS table is used to store the flow into the called method parameters. The other table, called the RETURN table, contains the flow returned from a method call.

The CALLS table contains an entry for each method called from within the current method. Generally when a method A calls another method B, information may flow in both directions: from A to B via the Input or Input/Output parameters (note, we forbid the use of global variables...), and from B to A via the Output or Input/Output parameters or via the Return value. In this paper we restrict the discussion to *Input* parameters and *Return* value only, the case of Output and Input/Output parameters is discussed in [Gendler(97)].

Basically, an entry in the CALLS table is $(method, ParInFlow_1 \ldots ParInFlow_n)$ where *method* is the name of the called method and *Pars* are sets of objects and virtual symbols that contain the information flow into the *method*'s parameters. If the called method returns a value via the Return statement, then this value is denoted also as a virtual symbol. Virtual symbols are used to denote information flow sets which are not known at Method compilation time and are instantiated only at *Transaction compilation time.* There are two kinds of virtual symbols: $\_\$i$ denotes information flow into parameter number i and $\_@j$ stands for information flow from the called method (via return value) and corresponds to entry number j in the CALLS table.

Assume now a transaction that invokes the method A. As the transaction is being compiled, the actual parameters passed to the methods are known. Let assume that real objects o_1 and o_2, stand for the parameters of A and therefore, the virtual symbols $\_\$i$ are *binded* to them. When B is processed, flows to the actual parameters substitute the virtual symbols $\_\$i$ of B. The union of flows to all return values of B substitute the virtual symbol $\_@n$ of A. Figure 2 demonstrates this process (see a full discussion in Section 5).

We now specify precisely the information that is entered into the tables. The first case is when the method is called within an arithmetic or another expression (i.e a function).

$$b = E \rightarrow method(E_1, E_2 \ldots E_n)$$

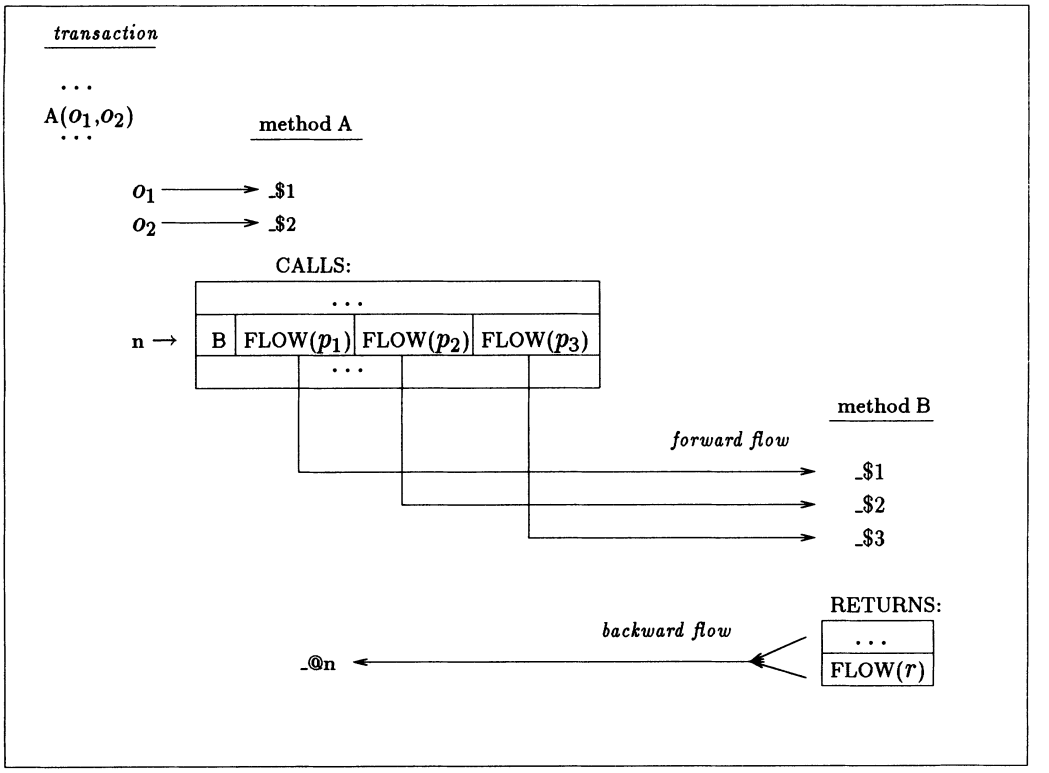

Figure 2: Analysis of transaction

The new entry in the CALLS table is:

$$(method, \text{GlFlow}(\bigcup_{a \in E_1} \{\{a\} \cup \text{FLOW}(a)\}) \ldots \text{GlFlow}(\bigcup_{a \in E_n} \{\{a\} \cup \text{FLOW}(a)\} \cup \text{IN}) \tag{3.7}$$

where GlFlow(a) is the set FLOW(a) without local variables. To store the return value, e.g. within the flow of the expression E (which flows into b), the virtual symbol _@j is used. A similar case is a call of the method which returns nothing.

Return value

The return value provides the means for backward information flow from the called method, to the method or the transaction which has called it.

$$S \rightarrow \textbf{return}(E)$$

An entry in the the RETURN table is inserted for each Return statement and it contains the following information flow

$$\text{GlFlow}(\bigcup_{a \in E} \{\{a\} \cup \text{FLOW}(a)\} \cup \text{IN}) \tag{3.8}$$

Note that since there may be multiple Return statements, several entries will be created, however, since we do not know which Return will be taken and we are interested in potential flow, these entries will be merged into a single entry at the Transaction analysis time.

4 Example

Consider the university database shown in Figure 1. Let us suppose that the salary of the university president is the most secret piece of information in the database. It is also reasonable to assume that the president of the university has the greatest salary between all university employees. We will investigate an example of illegal information flow caused by the called methods, in consequence of which the president's salary is compromised.

The methods code is shown below. In this code we have not used any special query language for read/write queries. Inside a method we just use the statement *read_object*$(o.A)$ where o is a database object and A is an attribute. It is important to note that at Method analysis time, since in a statement such as *read_object*$(o.A)$ the specific instance of o is not known, we use instead the root Class O of that object. In actuality, when the transaction analysis is performed, this root class is replaced with the restricted set of sub-classes and objects allowed access to the transaction's initiator (i.e the *AT_yes* subtree). In the tables, though we denote it as the root class, or as a virtual symbol.

The example below shows two methods, one calling the other. For each method we show the important tables generated by the method analysis phase.

```
  Max_Payed_Employee(Employee_Hierarchy emp_tree)
{ int Max = 0;
  int ESSN, ESalary;
  Employee emp;
```

emp	_$1
ESalary	{_$1.Salary,*emp*.Salary }
ESSN	{ _$1.SSN,*emp*.SSN }
Max	{ ESalary , _$1.Salary, *emp*.Salary,_@1}

Table 3: Max_Payed_Employee.FLOW

```
for ∀emp ∈ emp_tree
{ ESalary = read_object(emp.Salary);
  ESSN = read_object(emp.SSN);
  if (ESalary > Max)
   Max = Store_Results(ESSN, ESalary)
}
return Max; }

int Store_Results(int res1, int res2)
{  Dummy dummy;
  write(dummy.val1, res1);
  write(dummy.val2, res2);
  return res2; }
```

First, as a result of the analysis of the method Max_Payed_Employee, the table Max_Payed_Employee.FLOW is generated as shown in Table 3. The first row shows the fact that local object *emp* contains the flow received via the parameter. The next two rows represent the flows caused by the read queries: The fourth row shows the information flow to local variable Max via the statement

$$Max = \text{Store\_Results}(ESSN, ESalary) \tag{4.1}$$

This is the first case of a *method-call* discussed above. It is composed from the flow from the **if** statement and the flow returned by the called Method. The virtual symbol _@1 contains the flow returned from the Store_Results method. The flow into Max contains the IN flow and the flow resulted from the expression **if** $(ESalary > Max)$, so all the information from the conditional expression flows to Max. It is $ESalary$ and its FLOW which is {_$1.Salary,*emp*.Salary }. That explains the three components of the entry for Max in the table.

Store_Results	{ _$1.SSN, _$1.Salary} , { _$1.Salary}

Table 4: Max_Payed_Employee.CALLS

Dummy.val1	{_$1}
Dummy.val2	{_$2}

Table 5: Store_Results.WRITES

Now, the calling of method Store_Results results with a new entry in table CALLS as shown in Table 4. The entry contains the name of the called method and its forward information flow - i.e. the information flowing into the parameters - $ESSN$ and $Esalary$ plus information flow to the statement (flow IN), recall that the statement is part of an **if** structure, so as seen above, { _$1.Salary} is added to the flow of each of the parameters.

Now let us see the analysis of the method Store_Results. The FLOW table is empty and is not interesting. The WRITES table contains two entries as shown in Table 5. The RETURN table contains just the entry *(_$2)*. In the next section we shall see how the information contained in the above tables is combined for the detection of non-safe information flow.

5 Analysis of transactions

The analysis of a transaction is the final stage of the analysis described in this paper. Transactions may contain queries to the database - i.e. Read/Write queries, and calls to various methods. We assume that all components of a transaction are executed sequentially - i.e. no control flow statements are allowed. (this is not a real limitation since analysis similar to the one described for methods can be done, or alternatively they can be inserted within another method.) In terms of Access Control, we assume the authorization model for methods in which a method corresponds to an Access Type, i.e internal actions of the method do not require additional authorization (see [GalOz(93)]).

The transaction analysis uses the auxiliary results of the methods

analysis - i.e the tables WRITES, CALLS and RETURN discussed above. We add a new table called TR_FLOW to store all information flows caused by the transaction at any point of time. The table TR_FLOW will be used for computing the actual flows to the parameters of the methods. Note, that information flow has the property of transitivity: if flows $a \rightsquigarrow b$ and $b \rightsquigarrow c$ are safe, then the flow $a \rightsquigarrow c$ is safe too. This means that to verify safety of information flow $o_1 \rightsquigarrow o_2$ caused by writing within a method, there is no need to record all flows $o_i \rightsquigarrow o_1$ caused by previous write queries within the transaction and which were analyzed to be safe. In checking for safeness we use the ideas discussed in Section 2.3. That is, for Read Queries we first obtain by the authorization algorithm the subtree AT_yes and use it as the actual flow. The actual information flow is computed by *binding* the flow from the transaction with the virtual symbols recorded within the methods analysis tables. The analysis algorithm **TranFlowControl** is as follows:

TranFlowControl(*transaction*, *initiator*)
for ∀ method *meth* invoked by the *transaction*
switch (*meth*)
case read query $a = read(O.Attr_i)$
Generate AT_yes using Initiator privileges
TR_FLOW(a) = TR_FLOW(a) ∪ $AT\_yes(query)$
break;
case write query $write(O.Attr_j, a)$
Generate AT_yes using Initiator privileges
if not **Safe**(TR_FLOW(a), $AT\_yes(O.Attr_j)$, *initiator*)
return FALSE
break;
case method $a = meth(p_1 \ldots p_n)$
if *initiator* ∉ RACL(*meth*)
break; /* no need to check flow */
if not MethFlowControl(*meth*, TR_FLOW(p_1)...TR_FLOW(p_n)
return FALSE
TR_FLOW(a) =TR_FLOW(a) ∪ (∪ $meth.RETURN$)
break;
return TRUE

The **MethFlowControl** algorithm works as follows: first, it substitutes the virtual symbols with real values, then it verifies all write

queries within the method and checks for safe information flow. To verify the consistency of information flow in *all* invoked methods the algorithm works in a recursive manner. The decision about safety of a particular information flow is made during the process of the WRITES table binding:

MethFlowControl($m, fl_1 \ldots fl_n$)
bind(_$1, fl_1)
...
bind(_$n, fl_n)
for ∀ entry i in CALLS table ($meth, Par_1 \ldots Par_n$)
if not **MethFlowControl**($meth, Par_1 \ldots Par_n$)
return FALSE
bind(_@i, ∪ *meth*.RETURN) /*all entries taken */
for ∀ entry in WRITES table ($Obj, WrFlow$)
if not **Safe**($WrFlow$,Obj)
return FALSE
return TRUE

Note that the *bind* of Return value is possible, since it is known when we return from the recursion.

The **Safe** algorithms checks whether a given information flow is safe using the same algorithm as described in Section 2.3 (and in details in [Gudes(97)]). Remember, $CUAT(obj)$ stands of the intersection of all objects which can be read by users who can also read object obj.

Safe($flow, obj$)
if Contain($flow$,CUAT(obj))
return TRUE;
else
return FALSE;

The proof of correctness for the above algorithms is quite obvious and is given in [Gendler(97)].

5.1 Example - continued

Let us illustrate the execution of the algorithm by the example of the method
Max_Payed_Employee discussed in the previous section. Assume the following transaction which invokes the method Max_Payed_Employee.

begin transaction
1.*create Private*

2.$emp\_tree = read(Employee.\{Salary, SSN\})$
3.a =Max_Payed_Employee(emp_tree)
4.$write(Private, a)$
end transaction

The object *Private* is the private object created by the initiator of the transaction. The AT_yes result of the read query is returned to *emp_tree*. This tree is a sub-tree of the Employee root class, and includes the objects permitted to the initiator of the transaction. (Note, different sub-trees may be authorized for different users). The method Max_Payed_Employee receives the tree as a parameter and returns the maximal payed employee within it. At the first glance the transaction seems legal. Even if the method will return the salary of the president (assuming the initiator has access to it), the initiator will write it to his private object which nobody has access to, and no non-safe information flow will occur. However, to see the whole picture we must analyze the information flows caused by the method Max_Payed_Employee. Applying the algorithm **TransFlowControl** we get the following results (refer to the numbered items in the transaction):

1. No flow recorded yet.

2. TR_FLOW(*emp_tree*)=AT_yes(Employee.{Salary,SSN},*initiator*)

3. The flow from the method is computed by calling:
   MethFlowControl(Max_Payed_Employee,TR_FLOW(*emp_tree*)).
   and binding the parameters in the table Max_Payed_Employee.FLOW:

   bind(*_$1*, TR_FLOW(*emp_tree*))
   Next, the following recursive call is made:

   MethFlowControl(Store_Results, {TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary}, {TR_FLOW(*emp_tree*).Salary})

   The substitution of real data inside the virtual symbols is performed:
   bind(*_$1*, {TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary})
   bind(*_$2*, {TR_FLOW(*emp_tree*).Salary })

   Now we have to analyze the WRITES table of this method. The decision about the safety of the information flow is made by the two calls to the algorithm **Safe**:

   Safe({TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary}, Dummy.val1)

Safe({TR_FLOW(*emp_tree*).Salary}, Dummy.val2)

Clearly, the information flows:
{TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary} $\rightsquigarrow$ Dummy.val
and
{ TR_FLOW(*emp_tree*).Salary } $\rightsquigarrow$ Dummy.val2 are detected.

4. The flow into the object *private* is computed, but since this is a private object, its CUAT is at least *AT_yes*, therefore this write is obviously safe.

The safeness of this transaction is therefore dependent on the safeness of step 3. Let us imagine that Dummy is a public object open to all users (i.e. we can assume that CUAT(Dummy) may contain very few objects, maybe the salary of the students employees only). So the safety of the information flow depends on the privileges of the transaction initiator u. If the transaction initiator is a user allowed to read students' salaries only, i.e.

$$\text{TR\_FLOW}_u(emp\_tree) = \text{Student.\{SSN,Salary\}}$$

then there is no non-safe information flow. But, if the transaction initiator is a user allowed to read the president's, i.e.

$$\text{TR\_FLOW}_u(emp\_tree) = \text{\{Manager,President\}.\{SSN,Salary\}}$$

then the method Max_Payed_Employee plays the role of a **Trojan Horse** [Samarati(97)]. Clearly, this illegal flow is discovered, since TR_FLOW is not contained in CUAT(Dummy).

5.2 Authorization Rules Maintenance

As was explained in Section 2.3 and used in the **Safe** algorithm, the main data structure with which the flow computed at compile-time is checked, is the structure: *CUAT* - the common user access tree. This structure can be uniquely determined for a given set of authorization rules. However, when authorization rules change, this structure has to be recomputed.

There are basically two approaches. One is to recompute it every time a new transaction is compiled. This carries heavy computational overhead but saves storage. The other option is to compute it once and stores it for each object (class), and maintain it when authorization rules

are added or deleted. Since the events of changes in authorization rules are much less frequent, this is much better computationally. Algorithms to maintain the CUAT structure were also presented in [Gudes(97)] . A similar approach is advocated by [Bertino(96)].

6 Conclusions

In this paper we discussed the problem of protecting against unsafe information flow in object-oriented databases. Previous models for such protection provide a run-time mechanism which carries a considerable run-time overhead. This paper presents a compile-time solution to the problem. It relies on the compile-time analysis of methods which uses well known program analysis techniques. It then uses the Methods analysis phase in analyzing the transactions and deciding at transaction compile-time whether an unsafe information flow exists or not. In the future we like to consider the case of distributed transactions (or autonomous objects) when no single-centralized control is available for the analysis phase.

References

[Aho(86)] A.V.Aho,R.Sethi,J.D.Ullman *Compilers, principles, techniques and tools*, Addison-Wesley, 1986.

[Bertino(96)] Bertino, E., Bettini, C., Ferrari, E., Samarati, P., "A Temporal Access Control Mechanism for Database systems," *IEEE Trans. on Knowledge and Data Engineering*, Vol 8, No. 1, pp. 67-80.

[Castano(95)] Castano, S., M. Fugini, G. Martella, P. Samarati, *Database Security*, Addison-Wesley, 1995.

[Denning(86)] D.E.Denning *Cryptography and Data Security*, Addison-Wesley, 1983.

[GalOz(93)] N.Gal-Oz, E.Gudes and E.B.Fernandez "A Model of Methods Access Authorization in Object-Oriented Databases.",*Proc. of the 19th VLDB Conference*, Dublin,Ireland,1993.

[Gudes(97)] M.Gendler-Fishman,E.Gudes, "A Compile-time Model for safe Information Flow in Object-Oriented Databases", *Information*

Security in Research and Business, Edited by L. Yngstrom and J. Carlsen, Chapman & Hall, 1997, pp. 41-55.

[Gendler(97)] M.Gendler-Fishman*A Compile-time Model for safe Information Flow in Object-Oriented Databases*,thesis for MSc, Ben-Gurion University.

[Griffth(76)] Griffith, P., Wade B., "An Authorization Mechanism for a Relational Database System," *ACM Trans. on Database Systems*, Vol 1, No. 3, September, 1976.

[Fernandez(94)] E.B.Fernandez, E.Gudes, H.Song "A Model for Evaluation and Administration of Security in Object-Oriented Databases.",*IEEE Trans. on Knowledge and Data Engineering*, Vol.6. No.2., April 1994, pp. 275-292.

[Kemper(94)] Kemper A., G. Moerkotte, *Object-oriented Database Management*, Prentice-Hall, 1994.

[Kim(90)] Kim W., *Introduction to Object-Oriented Databases*, The MIT Press, 1990.

[Larrondo(90)] Larrondo-Petrie M., Gudes E., Song, H., Fernandez E B., "Security Policicies in object-oriented databases," *Database Security IV: Status and Prospectus*, D. L. Spooner C. E. Landwehr (Ed.), Elsevier Science Publishers, 1990, pp. 257-268

[Muchnick(81)] S.S.Muchnick,N.D.Jones *Program Flow Analysis: theory and applications*, Prentice-Hall,1981.

[Samarati(97)] Samarati P., E.Bertino, A.Ciampichetti and S.Jajodia "Information Flow Control in Object-Oriented Systems," to appear in IEEE Trans. on Knowledge and Data Engineering, 1996.

[Stonebraker(76)] Stonebraker, M., Wong, E., Kreps, P., Held, G., "The Design and Implementation of Ingres", *ACM Trans. on Database Systems*, Vol 1, No. 3, September, 1976.