4

A Compile-time Model for safe Information Flow in Object-Oriented Databases

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

Abstract

Security is an important topic for Object-oriented databases (OODB). Discretionary authorization models do not provide the high assurance provided by Mandatory models, the latter ones, however, are too rigid for commercial applications. Therefore discretionary, information-flow control models are needed, especially when transactions containing general methods invocations are considered.

This paper first reviews existing security models for object-oriented databases with and without information-flow control. Previous models relied on the run-time checks of every message transferred in the system. This paper uses a simple transaction model and 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. A proof for correctness is given, and the performance implications are discussed.

keywords

Object-oriented Databases, Authorization, Information flow, Transactions, Compile-time checking.

1 Introduction.

Security is an important topic for Databases in general and for Object-oriented databases (OODB) in particular [Kim(90), Kemper(94)]. Commercial multi-user Database Management Systems (DBMSs) thus provide authorization mechanisms supporting the definition and enforcement of authorization rules. 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. 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(96)]. The main problem with the model in [Samarati(96)] 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 [Griffith(76)] in Relational systems, or the model suggested by [Fernandez(94)] for OODBs. In this paper we investigate the problem of ensuring safe information flow for OODBs 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 paper presented here is that the run-time of the Query-language and the DBMS can be trusted. That it, if one composes its transactions only from well-defined Queries and Update, (the exact model for transactions is discussed later), one can rely on the Query translator and on the Access validation associated with it. Clearly, this cannot include an OODBS with the most general *methods*, since some of these methods may not be trusted. (see [Gudes(97)]). In the rest of the paper we assume therefore that transactions contain queries with basic Read/Write (or other trusted methods) operations.

As this paper relies heavily on the two previous papers [Samarati(96)] and [Fernandez(94)], these papers are first reviewed briefly in Section 2 and the definition of safe information flow is given. In Section 3 we present our compile-time model and some examples. The main algorithms and their performance analysis are presented in Section 4. Section 5 is the Summary.

2 Background.

2.1 Fernandez et. al

This model uses the following well known concepts:

Object: A real-word entity with unique identifier.

Attributes & Methods: The components of an object which define its behavior.

Class: Hierarchically structured sets of objects with the same methods and attributes. We can say that an object is an instance of a class or *object instance.*

Generalization: The classes are partially ordered, the relation "*subclass-superclass*" exists (we denote $t_1 \preceq t_2$ if t_1 is subclass of t_2). Both attributes and methods are inherited by subclasses from superclass.

Encapsulation: The only way to access data values of an object is through the methods in its interface. It is assumed that for every attribute there are built-in read/write methods.

This model assumes a simple discretionary Rules-based authorization. The model deals mainly with the impact of inheritance on security and enforces the following basic policies:

P_1 (*inheritance*) – a user that has access to a class is allowed to have similar type of access to the corresponding subclasses attributes inherited from that class.

P_2 (*class access*) – access to a complete class implies access to the attributes defined in that class as well as to attributes inherited from a higher class (but only to the class-relevant values of these attributes).

P_3 (*visibility*) – an attribute defined for a subclass is not accessible by accessing any of its superclasses.

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)]

To enforce the above policies an *Access Validation* algorithm was presented. 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 (see Figure 1.). 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). Initially, all the AT_yes's nodes are set to authorized. After the validation algorithm, the AT_yes contains only the nodes and the attributes to which access is allowed.

- 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 three policies mentioned above and produces the final AT_yes which defines the allowed access. Briefly, for each node and attribute in the AT_yes, the algorithm searches for rules authorizing them. If such an explicit rule is not found, an implicit rule authorizing a node at a higher level is searched for. If such a rule is not found, then a rule authorizing partial access for a node in a descendant of the AT-node is looked for. If no rule is found, then no authorization is given. [1]

[1] the algorithm above assumes the class-hierarchy is a tree, but it can be generalized easily to acyclic graphs and multiple inheritance. It is not a central point in this paper and therefore will not be discussed further.

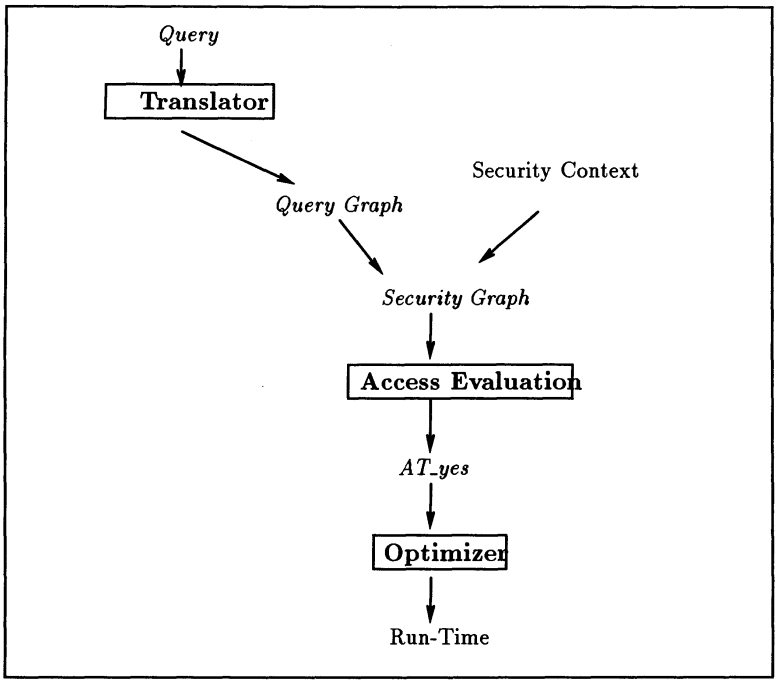

Figure 1: Architecture of access evaluation

2.2 Samarati et. al

The model includes the following main entities:

- **Objects** Objects are identified by a unique identifier, an ordered set of attributes, and an ordered set of methods.
- **Messages** A message is an ordered pair *(name,pars)* Where *name* is the name of the message, and *pars* is its parameters. There may be *Read*, *Write* or *Create* messages.
- **Transaction** A transaction is a sequence 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 can defer its waiting (deferred). 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.

- **Forward and Backward Transmission** Since one method may invoke another method, information may be transferred forward (from the invoker to the invokee) or backwards (from the invokee to the invoker). Computing this information is important for the purposes of computing Information Flow.

- **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(96)] suggests the construction of a *Message Filter* component which intercepts each and *every message* in the system. For each such intercepted message, the Message filter keeps track of:

1. The information that the execution has received from its invoker, through the message parameters.

2. The information that the execution has received from the execution it has invoked through the message replies

3. The RACL of the relevant objects

Using all 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 section, we present a simpler model with a compile-time algorithm.

3 The Object and Transactions Model

Our model include the following concepts:
Object Model The object model is similar to the one in [Fernandez(94)]
Authorization Model - The authorization model is also similar to the one in [Fernandez(94)]. Authorization rules are in the form of triple $(U, A, O.attr.)$ where U - a user or a user group, A - is a basic access type like READ/WRITE, $O.attr$ - stands for an attribute of a class O.
Authorization rules reference Classes, although the model carries over when we

deal with Objects, i.e class instances. In the sequel we will use O to denote classes, and will not make the distinction to objects unless necessary. We also adopt the *inheritance* policies described in Section 2 above.

Transactions A transaction in our model is simpler than in [Samarati(96)]. It consists of two types of methods calls only, i.e Read and Write (we call both of them queries below) and both are called from the transaction level:

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

For every write query all read queries executed before are considered.

Access Lists.

In [Fernandez(94)] the main administration structure was the *authorization rule*. For purposes of flow control we need to define also for each attribute of each class a list of all users authorized to read it. We maintain the structure called *read access list*(RACL) containing the list of users who are allowed Read access 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 users that are authorized to read the attribute either explicitly or via the inheritance policies specified above. [2]

Information Flow Using the concepts of Transactions and Access list we can define information flow. The main idea here is to collect information about read queries: what attributes were read and who may read these attributes. With the aid of this information we can decide whether a write method causes a non-safe information flow. The information flow from object o_i to object o_j is safe if the set of users who can read object o_j is contained in the list of users who can read o_i, i.e.

$$\text{RACL}(o_j) \subseteq \text{RACL}(o_i)$$

As an example, let us consider the transaction T:

$v_1 = read(O_1.Attr_1)$
$v_2 = read(O_2.Attr_2)$
...
$v_n = read(O_n.Attr_n)$
$write(O_j.Attr_j, v_j)$

[2] there may be some rules for some users which negate access to descendants of the current attribute, thus this RACL actually represents the list of users who have either complete or partial access to this attribute.

For this transaction, the flow of information is safe if and only if the union of all lists belongs to RACL($O_j.Attr_j$).

$$\cup RACL(O_i.Attr_i) \subseteq RACL(O_j.Attr_j)$$

However, this is only a strong sufficient condition. A particular user when issueing $Query_i$ gets only part of the query authorized by the AT_yes structure. We can therefore find a better bound for this transaction using Compile-time analysis! In order to apply our compile-time algorithms we need to define several types of Authorization trees:

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 only use the authorization-trees for Read access, and therefore denote them as: AT_yes($u, O.Attr$). Also, in the following, we will use AT_yes(i) to denote the authorization tree of the read query number i in the transaction above. Now we define for any user the structure which is his visible part of database .

User Access Tree (UAT) The set of attributes in the entire database [3] 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 additional data structure $RACL$, but, obviously, it is also true that

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

Common User Access Tree(CUAT). We introduce a new measure for each 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.

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

Safe Information Flow.

Using the definitions above we are now ready to express the criteria for safe information flow. Intuitively, we know that every read query after validation can only read the objects and attributes contained in the 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 that union of information.

[3] the term "entire database" is used here for purposes of definition and correctness, it is not used in this way in the algorithm

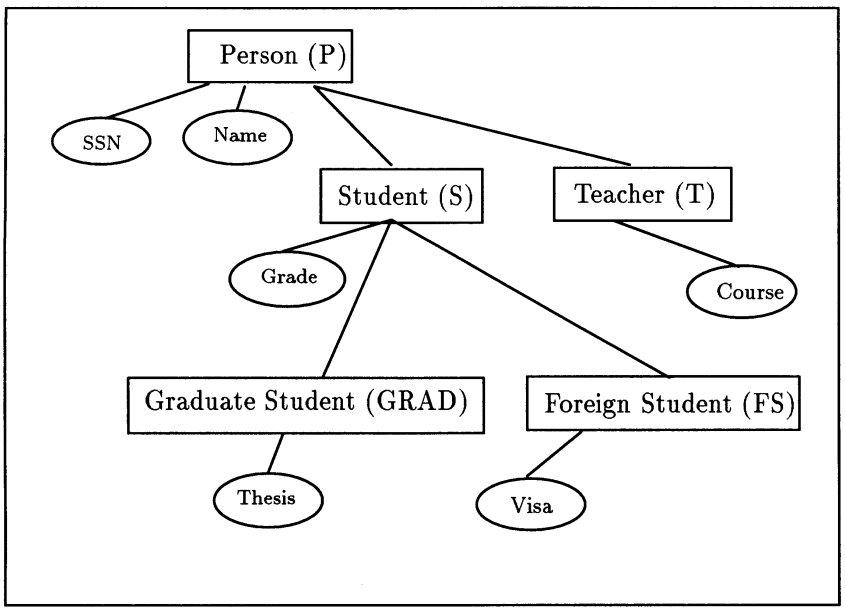

Figure 2: University database

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 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 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. $\Rightarrow$ If CUAT contains the union of AT_yes then there is no user u and attribute a such that u was not authorized to read a and could read the value of a from $O_k.Attr_j$ after write access.
$\Leftarrow$ If the information flow to $O_i.Attr_j$ is safe then there is no way of transmitting secret information to $O_k.Attr_j$. So all data that may be transferred to $O_k.Attr_j$ is accessible for all users of $O_k.Attr_j$. So $\bigcup_{i=1}^{j-1} \text{AT\_yes}(i) \subseteq \text{CUAT}(O_k.Attr_j)$ exists. [4]□

3.1 Example.

Consider the university database shown in Figure 2. Assume the following authorization rules are defined:

(u_1,R,S.SSN) (u_1,R,T.SSN)
(u_2,R,P.SSN) (u_2,-R,T.SSN)
(u_3,R,P.SSN) (u_3,-R,GRAD.SSN) (u_3,W,FS,SSN)

[4]as stated in the introduction the query optimizer and run-time system is trusted and only those accesses allowed at compile time are actually executed

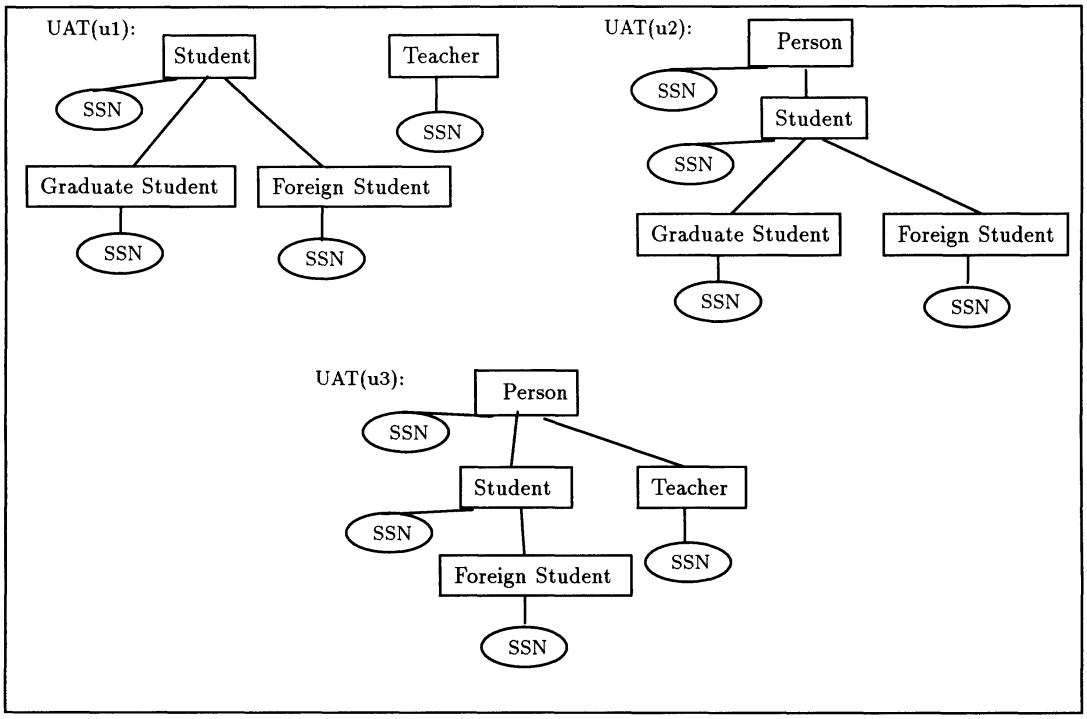

Figure 3: UAT example

The user access trees for users u_1,u_2,u_3 are shown in Figure 3. Now consider the following transaction T_1:

$v_1 = read(S.SSN)$
$v_2 = read(T.SSN)$
...
$write(FS.SSN)$

Assume that T_1 is executed with u_3 privileges (i.e u_3 is the initiator.) The Authorization trees for the two read queries are shown in Figure 4. Now consider the RACL of the attribute of the last write query:

$$\text{RACL(FS.SSN)} = \{U_1, U_2, U_3\}$$

The CUAT for this attribute (which is the intersection of the 3 UATs in Figure 3) is shown in Figure 5. Now let us look at the situation that can happen after the transaction execution. Intuitively, the attribute FS.SSN is accessible to more users than T.SSN. Therefore the SSN value of a teacher may be read during the transaction and written to the SSN attribute of foreign student. The user u_2 has now access to the value previously unaccessible to him. So unsafe information flow may occur during the transaction execution. As can be seen from the figures, the union of the authorization trees is not contained within the relevant CUAT, as the theorem requires.

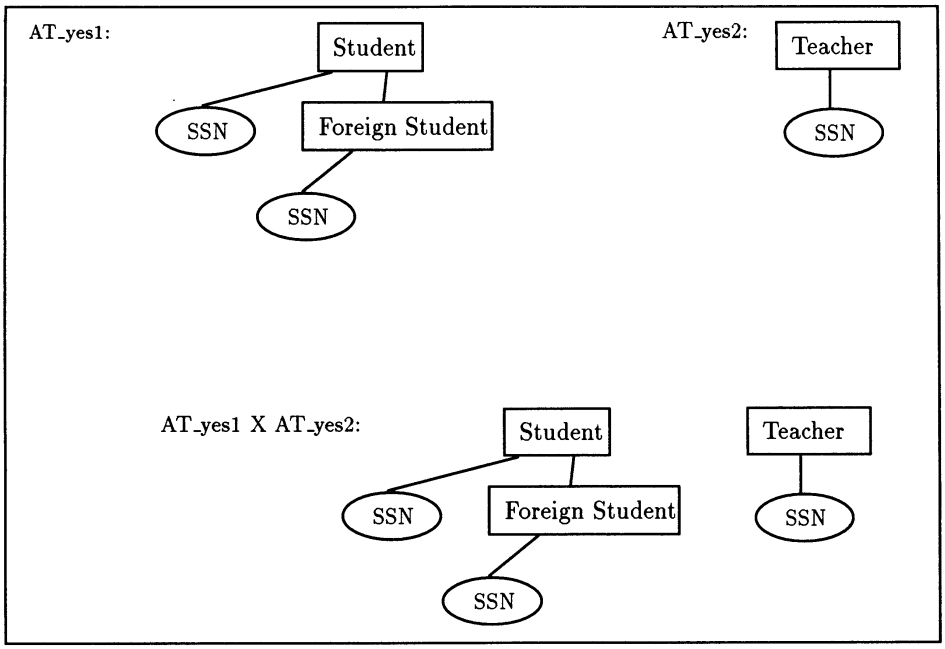

Figure 4: Example Authorization Trees

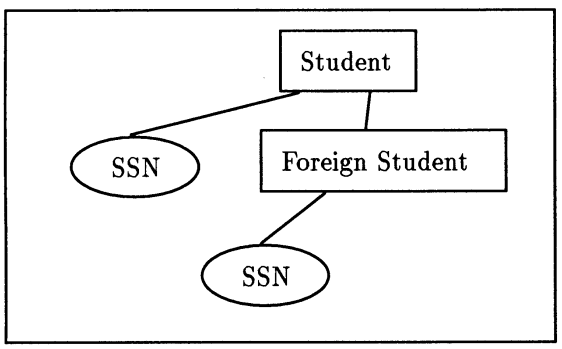

Figure 5: CUAT example

4 The Algorithms.

4.1 The CUAT management algorithms.

There are two possibilities to construct and manage the CUAT structures: to calculate CUAT at compile-time or to store and maintain all the CUAT global structures in the system. We first consider the second method because it allows to update the necessary structures without complex calculation. Once the CUATs are created, we must update them only at the moment of adding/removing of authorization rules. Note, that the system must maintain the CUAT structures of only those attributes which may be accessed for writing. A similar approach, i.e that of storing data structures at Rule Administration time, is suggested in [Bertino(96)].

First, consider an adding rule situation. Assume we add a new rule : $(w, R, O.Attr_1)$. What is $CUAT(O.Attr_1)$ now? Recall that it is equal to the intersection of all UAT's of all users who are granted read access to $(O.Attr_1)$.
Formally,

$$\mathrm{CUAT(O.Attr_1)} = \bigcap_{\mathrm{u \in RACL(O.Attr_1)}} \mathrm{UAT(u)} = \bigcap_{\mathrm{u \in RACL'(O.Attr_1)}} \mathrm{UAT(u)} \cap \mathrm{UAT(w)} =$$

$$= \mathrm{CUAT'(O.Attr_1)} \cap \mathrm{UAT(w)}$$

where RACL' and CUAT' are the corresponding structures before adding of the authorization rule. Therefore, it is quite easy to compute the new CUAT in this situation. The new CUAT depends on the new RACL list , which is the old list plus the user w. So we just need to intersect the old UAT's of all users from the old RACL with UAT(w).
The rule adding algorithm also tries to update the CUAT structures of attributes from existing UAT tree. There is the chance for expanding of CUAT by adding attribute $O.Attr$ if all users of some attribute except w had access to $O.Attr$.

AddRule$(w, R, O.Attr)$
  /* *first, check chance for expanding of existing CUAT structures* */
  for each $O_i.Attr_j \in$ UAT(w)
   common := O.Attr
   for each $v \in$ RACL($O_i.Attr_j$), $v \neq w$
    common := *common* $\cap$ UAT(v)
   CUAT($O_i.Attr_j$) := CUAT($O_i.Attr_j$) $\cup$ *common*
   /* *calculate CUAT* */
  UAT(w) := UAT(w) $\cup$ $O.Attr$
  for each node $o.Attr \in O.Attr$
   if (RACL($o.Attr$) = $\emptyset$)
    CUAT($o.Attr$) := UAT(w)
   else
    CUAT($o.Attr$) := *Intersect*(UAT(w),CUAT($o.Attr$))

$\text{RACL}(o.Attr) := \text{RACL}(o.Attr) \cup w$

The removal of an authorization rule is more complex. The only case when the CUAT structure must be changed after deleting an authorization rule is that all users except w may read some attribute $Attr_{all}$. After deleting user w from $\text{RACL}(Attr_1)$ the attribute $Attr_{all}$ will belong to $\text{CUAT}(Attr_1)$ (while before it didn't). But the implementation of this property requires to keep the information about all *candidates* to a new CUAT and the storage of this information is too large. We therefore prefer the simpler implementation, that of re-building the CUAT from scratch - i.e. calculating the intersection of UAT of all other users. (a reasonable assumption is that deletion of authorization rules occur much less often than adding new ones...). The algorithm is as follows:

RemoveRule$(w, R, O.Attr)$
$\text{UAT}(W) := \text{UAT}(W) - O.Attr$
for each $O_i.Attr_j \in \text{UAT}(w)$
$\text{CUAT}(O_i.Attr_j) := \text{CUAT}(O_i.Attr_j) - O.Attr$
for each node $o.Attr \in O.Attr$
$\text{RACL}(o.Attr) := \text{RACL}(o.Attr) - w$
if $(\text{RACL}(o.Attr) = \emptyset)$
$\text{CUAT}(o.Attr) := \emptyset$
else
begin
$\text{CUAT}(o.Attr) := \Omega$ $\quad /* \textit{universal set} - \textit{all OODB} */$
for each $v \in \text{RACL}(o.Attr)$
$\text{CUAT}(o.Attr) := \textit{Intersect}(\text{UAT}(v), \text{CUAT}(o.Attr))$
end

Both the above two algorithms require the intersection of two authorization trees, such intersection algorithm is presented in [Gendler(97)] **Comment**. A problem may arise, when a rule authorizing a Read access to a user U on an object O is added after a transaction is completed. The transaction may have updated the object and caused information flow into it. This flow was valid before the rule was added. However, with the addition of the rule, that past flow may not be valid anymore! There is no easy solution to this problem (it will also occur in [Samarati(96)]...). The best way is to verify before adding rule that allows access to object O, that object O does not contain any unsafe information as far as user U is concerned.

4.2 Compile-Time Algorithm

Now, after the CUAT trees are constructed, we are ready to check for information flow. The information flow control is processed after the compilation of each trans-

action. First, each query is validated using the transaction initiator privileges, [5] and constructing the AT_yes trees. Then the FlowControl algorithm is applied to verify that the privileges of the transaction initiator are sufficient for all read/write queries within the transaction.

FlowControl(*transaction*, *initiator*)
AT_all := $\emptyset$
for $\forall$ method *meth* invoked by the *transaction*
if *meth* is read primitive: $read_i(O.Attr_i)$
AT_i is query graph of *meth* with regards to *initiator*
AT_all := AT_all $\cup$ AT_i
if *meth* is write primitive: $write(O.Attr_j, val)$
if not *Contain*(AT_all,CUAT($O.Attr_j$))
return FALSE
return TRUE

Both the *Union* and *Contain* algorithms of authorization-trees are quite simple and can be found in [Gendler(97)].

Comment The algorithms above assumed that the UAT is constructed for the entire database and is stored that way. Such a structure is associated with every user in the system and may take considerable space, and also the intersection of two such UATs may be quite long. In practice though users work within a particular *view* or *context* [Fernandez(94)]. Therefore, the UATs are usually constructed from distinct trees, and within a specific view or context, only the portions of the tree relevant to that context need to be managed (or intersected). This will require much less overhead.

4.3 Analysis

Considering the worst-case scenario of a CUAT containing all nodes in the database, the complexity of our compile-time algorithm is in the worst case $\mathbf{O}(r * n + w * n^2)$ where r is the number of the read queries in the transaction, w is the number of write queries and n is the number of attributes of all nodes in AT_yes tree (in the worst case the number of attributes in the entire database schema).

The complexity of the CUAT managing algorithm is: adding a rule is performed in time $\mathbf{O}(n^2)$ - complexity of intersection algorithm , while rule deleting is performed in $\mathbf{O}(u * n^2)$, where u is a size of RACL list, in the worst case , the number of users.

On the other hand, the message filter algorithm described in [Samarati(96)] analyses forward information (i.e. the objects that were read) after each write access, so its performance is strongly dependent on the huge number of database

[5] It is assumed that a transaction is associated with an initiator. If another user executes the same transaction, at some other time, the transaction needs to be recompiled and validated with the new user's privileges.

objects (not object *types*) accessed. Therefore the performance of run-time message filter algorithm is in the worst case $\mathbf{O}(R * u^2 + w * R * u^2)$,
where R and w are the *number of database objects* accessed for reading/writing, and u - as before is the size of the RACL list - i.e. number of users. Usually, $R >> n^2$ which shows the clear advantage of the compile-time approach. The advantages of our approach are even larger considering the average case (and the comment on contexts above).

5 Summary

The problem of Information-flow in object-oriented databases was discussed. It was argued that the approach suggested by Samarati et. al. [Samarati(96)] which requires the checking and storing information for every message in the system carries too much overhead at run-time. Instead, another model was suggested, where some data structures are constructed and maintained at Rule-administration time, and the rest of the checks are done at compile-time only, no Run-time checks are needed. This saves considerable overhead, and also potential information flow problem are discovered earlier.

The problem with the model presented here is that the methods called within each transaction are *trusted*!. This does not constitute a problem for the common Read/Write methods, but may be a problem with more complex methods. Furthermore, if a transaction contains control structures such as: Loop or If/Then/Else then our approach is too conservative and can be improved. Both problems can be addressed by looking at techniques for Dataflow analysis [Denning(86)] or program verification (see, e.g. the approach advocated by Java [Java(96)]). These are issues of our current research [Gudes(97)].

References

[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.

[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.

[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.

[Gendler(97)] Gendler, M. "A Model for secur Information-flow in Object-oriented databases," MSc Thesis, Ben-Gurion University, 1997.

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

[Gudes(97)] Gudes E., Gendler, M. "Compile-time Flow analysis of Transactions and Methods in Object-Oriented Databases," submitted.

[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

[Samarati(96)] 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.

[Java(96)] F.Yellin "Low Level Security in Java", Unpublished Report, Sun corp, 1996.