IMPLEMENTING REAL-TIME UPDATE OF ACCESS CONTROL POLICIES

Indrakshi Ray and Tai Xin *

Abstract *Real-time update* of access control policies, that is, updating policies while they are in effect and enforcing the changes immediately, is necessary for many security-critical applications. In this paper, we consider real-time update of access control policies that arise in a database system. Updating policy while they are in-effect can lead to potential security problems. In an earlier work, we presented an algorithm that not only prevents such security problems but also ensures correct execution of transactions. In the current work we extend that algorithm to handle addition and deletion of access control policies and provide the implementation details of the algorithm. We also describe properties of histories generated by this algorithm.

1. INTRODUCTION

Since security policies are extremely critical for an enterprise, it is important to control the manner in which policies are updated. Updating policy in an ad-hoc manner may result in inconsistencies and problems with the policy specification; this, in turn, may create other problems, such as, security breaches, unavailability of resources, etc. In other words, policy updates should not be through ad-hoc operations but done through well-defined *transactions* that have been previously analyzed.

An important issue that must be kept in mind about policy update transactions is that some policies may require *real-time updates.* We use the term real-time update of a policy to mean that the policy will be changed while it is in effect and this change will be enforced immediately. An example will help motivate the need for real-time updates of policies. Suppose the user *John,* by virtue of some policy *P,* has the privilege to execute a long-duration transaction that prints a large volume of sensitive financial information kept in file *I.* While *John* is executing this transaction, an insider threat is suspected and the policy *P* is changed such that *John* no longer has the privilege of executing this

*This work was partially funded by the US AFOSR under contract number FA9550-04-1-0102. The opinions expressed are those of the authors and do not necessarily reflect those of the AFOSR.

transaction. Since existing access control mechanisms check *John*'s privileges *before John* initiates the transaction and not *during* the execution of the transaction, the updated policy *P* will not be correctly enforced causing financial loss to the company. In this case, the policy was updated correctly but not enforced immediately resulting in a security breach. Real-time update of policies is also important for environments that are responding to international crisis, such as relief or war efforts. Often times in such scenarios, system resources need reconfiguration or operational modes require change; this, in turn, necessitates policy updates.

In this paper we consider real-time policy updates in the context of a database system. A database consists of a set of objects that are accessed and modified through transactions. Transactions performing operations on database objects must have the privilege to execute those operations. Such privileges are specified by access control policies; access control policies are stored in the form of *policy objects*. Transactions executing by virtue of the privileges given by a policy object is said to *deploy* the policy object. In addition to being deployed, a policy object can also be accessed and modified by transactions. We are considering an environment in which different kinds of transactions execute concurrently some of which are policy update transactions. In other words, a policy may be updated while transactions are executing by virtue of this policy.

To prevent security breaches caused by real-time update of access control policies, a simple solution is to abort all transactions that are executing by virtue of the policy that is being updated. Unfortunately, this results in unnecessary transaction aborts. This is because not all updates to a policy object are problematic. For instance if a policy object is updated such that the rights given by the policy are increased, then the transactions executing by virtue of the policy need not be aborted.

In an earlier work [14] we described a syntactic approach for classifying policy update transactions and proposed a concurrency control mechanism supporting this approach. In this paper, we analyze the characteristics of histories produced by the concurrency control mechanism and provide implementation details of this mechanism. We extend our approach to handle real-time creation and deletion of access control policies as well.

The rest of the paper is organized as follows. Section 2 gives our definition of policy updates and shows how we can classify a policy update as a relaxation or restriction. Section 3 proposes our transaction processing model for policy updates. Section 4 illustrates how the semantics of the policy update operation can be exploited to increase concurrency. Section 5 describes the implementation details of the policy update algorithm. Section 6 highlights the related work. Section 7 concludes our paper with some pointers to future directions.

2. DEFINING POLICY UPDATES

We consider policy updates in the context of a database system. A *database* is specified as a collection of objects together with a set of *integrity constraints* defined on these objects. At any given time, the *state* of the database is determined by the values of the objects in the database. A change in the value of a database object changes the state. A database state is said to be *consistent* if the values of the objects satisfy the given integrity constraints.

A *transaction* is an operation that transforms the database from one consistent state to another. To prevent the database from becoming inconsistent, transactions are the only means by which data objects are accessed and modified. A transaction can be initiated by a user, a group, or another process. A transaction inherits the access privileges of the entity initiating it. A transaction can execute an operation on a database object only if it has the privilege to perform it. Such privileges are specified by access control policies.

In this paper, we consider only one kind of access control policies: authorization policies. Henceforth, we use the term policy or access control policy to mean authorization policy. An authorization policy specifies what operations an entity can perform on another entity. We focus our attention to systems that support positive authorization policies only. This means that the policies only specify what operations an entity is *allowed* to perform on another entity. There is no explicit policy that specifies what operations an entity is *not allowed* to perform on another entity. The absence of an explicit authorization policy authorizing an entity A to perform some operation O on another entity B is interpreted as A not being allowed to perform operation O on entity B. We also assume that there is at most one policy specified over any given subject and object pair.

We consider simple kinds of authorization policies that are specified by *subject, object,* and *rights.* A subject can be a user, a group of users or a process [12]. An object, in our model, is a data object, a group of data objects, or an object class. A subject can perform only those operations on the object that are specified in the rights. A *policy* is a function that maps a subject and a object to a set of access rights. We formally denote this as follows: $P : S \times O \rightarrow \mathbb{P}(R)$ where P represents the policy function, S, represents the set of subjects, O represents the set of objects, $\mathbb{P}(R)$ represents the power set of access rights. In a database, policies are stored in the form of policy objects. A policy object P_i consists of the triple $< S_i, O_i, R_i >$ where S_i, O_i, R_i denote the subject, the object, and the access rights of the policy respectively. Subject S_i can perform only those operations on the object O_i that are specified in R_i. For example the policy object $P =< John, FileF, \{r,w,x\} >$ gives subject John the privilege to Read, Write, and Execute *FileF*.

Before proceeding further, we discuss how to represent the access rights. The motivation for this representation will be clear in Section 4. Let $\mathbf{OP}_i = \{o_1, o_2, \ldots, o_n\}$ be the set of all the possible operations that are specified on Object O_i. The set of operations in $\mathbf{OP}_i$ are ordered in the form of a sequence $< o_1, o_2, \ldots, o_n >$. We represent any access right on the object O_i as an n-element vector $[i_1 i_2 \ldots i_n]$. If $i_k = 0$ in some access right R_j, then R_j does not allow the operation o_k to be performed on the object O_i. $i_k = 1$ signifies that the access right R_j allows operation o_k to be performed on the object O_i. The total number of access rights that can be associated with object O_i equals 2^n. For example, let $< r,w,x >$ be the operations allowed on a file F. The access right $R_1 = [001]$ signifies that r, w operations are not allowed on the file F but the operation x is permitted on File F. The access right $R_2 = [101]$ allows r and x operations on the file F but does not allow the w operation.

The set of all access rights associated with a object O_i having n operations forms a *partial order* with the ordering relation $\geq_{O_i}$. The ordering relation is defined as follows: Let $R_j[i_k]$ denote the i_k-th element of access right R_j. Then $R_p \geq_{O_i} R_q$, if $R_p[i_k] = R_q[i_k]$ or $R_p[i_k] > R_q[i_k]$, for all $k = 1 \ldots n$. Given two access rights R_p and R_q associated with an object O_i having n operations, the *least upper bound* of R_p and R_q, denoted as $lub(R_p, R_q)$ is computed as follows. For $k = 1 \ldots n$, we compute the i_k-th element of the least upper bound of R_p and R_q: $lub(R_p, R_q)[i_k] = R_p[i_k] \vee R_q[i_k]$. The n-bit vector obtained from the above computation will give us the least upper bound of R_p and R_q. Given two access rights R_p and R_q associated with an object O_i having n operations, the *greatest lower bound* of R_p and R_q, denoted as $glb(R_p, R_q)$ is computed as follows. For $k = 1 \ldots n$, we compute the i_k-th element of the greatest lower bound of R_p and R_q: $glb(R_p, R_q)[i_k] = R_p[i_k] \wedge R_q[i_k]$. The n-bit vector obtained from the above computation will give the greatest lower bound of R_p and R_q. Since each pair of access rights associated with an object have a unique least upper bound and a unique greatest lower bound, the access rights of an object can be represented as a lattice. The set of all possible access rights on a object O_i can be represented as a lattice which we term the *access rights lattice of object* O_i. The notation $ARL(O_i)$ denotes the set of all nodes in the access rights lattice of object O_i. All possible access control privileges pertaining to a object can be represented as the nodes on the access rights lattice of the object. Each node in the lattice represents a specific access control privilege. The lower bound on this lattice (labeled as Node 0) denotes the absence of any access rights on this object. The upper bound denotes the presence of all the rights; any subject having these rights can perform all the operations on the object. The other points in the lattice denote the intermediate states.

Figure 1(a) shows the possible access rights associated with a file having only two operations: Read and Write. The most significant bit denotes the Read operation and the least significant bit denotes the Write operation. The

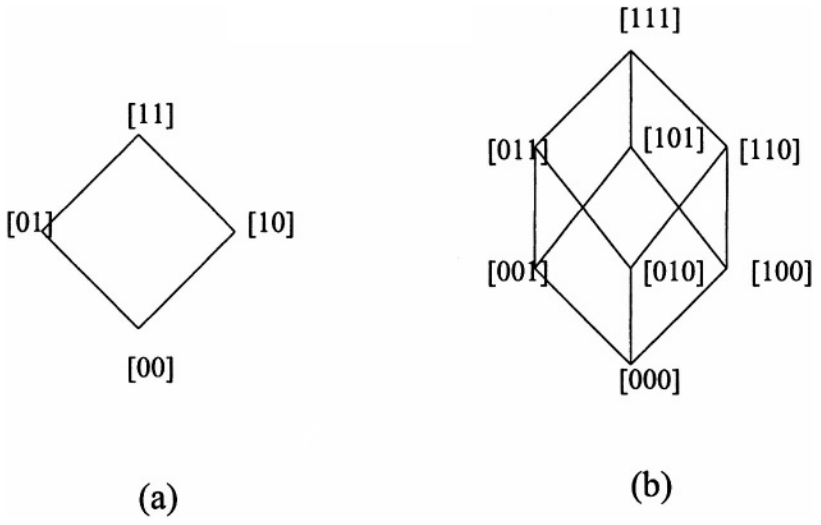

Figure 1. Representing Possible Access Control Rights of Objects

lower bound labeled as Node 00 signifies the absence of Read and Write privilege. The Node 01 signifies that the subject has Write privilege but does not have Read privileges. The Node 10 signifies that the subject has Read privilege but no Write privilege. The Node 11 indicates that the subject has both Read and Write privileges. Figure 1(b) shows the possible access rights associated with a object having three operations.

With this background, we are now ready to define a policy and policy updates in terms of the access rights lattice. A *policy* P_i maps a subject S_i's access privilege to some Node *j* in the access rights lattice of the object O_i. This is formally stated as follows: $P : S \rightarrow (ARL(O))$. A *policy update* is an operation that changes some policy object $P_i =< S_i, O_i, R_i >$ to $P_i' =< S_i, O_i, R_i' >$ where P_i' is obtained by transforming R_i to R_i'. Let R_i, R_i' be mapped to Node *j*, Node *k* of $ARL(O_i)$ respectively. The update of policy object P_i changes the mapping of the subject S_i's access privilege from Node *j* to Node *k* in the access rights lattice of object O_i.

A *policy relaxation operation* is a policy update that increases the access rights of the subject. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P_i' =< S_i, O_i, R_i' >$. Let Let R_i, R_i' be mapped to the nodes *k, j* respectively in $ARL(O_i)$. A policy update operation is a policy relaxation operation if *lub*(*k, j*) = *j*. For instance, let the operations allowed on *FileF* be < *r,w,x* >. Suppose the policy $P_i =<$ ***John***, *FileF,* [001] > is changed to $P_i' =<$ ***John***, *FileF,* [101] >. This is an example of policy relaxation because the access rights of subject John has increased. Note that *lub*([001], [101]) = [101]. Thus, this is a policy relaxation.

A *policy restriction operation* is a policy update operation that is not a policy relaxation operation. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P'_i =< S_i, O_i, R'_i >$. Let Let R_i, R'_i be mapped to the nodes *k, j* respectively in $ARL(O_i)$. A policy update operation is a policy restriction operation if $lub(k,j) \neq j$. For instance, let the operations allowed on *FileF* be $< r,w,x >$. Suppose the policy $P_i =< John, FileF,$ [001] > is changed to $P'_i =<$ $John, FileF,$ [110] >. This is an example of policy restriction because the access rights of subject John has not increased. Note that, *lub*([001],[110]) = [111]. Since $lub([001],[110]) \neq [110]$, this is an example of policy restriction. In other words, moving up the lattice along the edges indicate that policy is being relaxed. Moving down the lattice along the edges indicates that policy is being restricted. Moving from one node to another not connected by edges is also considered to be a policy restriction operation.

Having discussed about policy updates, we now focus on creation of new policies and deletion of existing policy. A *policy creation operation* is one in which a new policy $P_i =< S_i, O_i, R_i >$ is created. Note that creation of a new policy $P_i =< S_i, O_i, R_i >$ can be considered to be a policy relaxation operation. Recall that we assume that our system has positive authorization policies only and at most one policy can be specified over a given subject and object pair. Hence, before P_i was created S_i could not perform any operation on O_i. This can be represented as a dummy policy P_d that maps S_i to the minimal element in the access rights lattice of O_i (indicated by Node 0). The introduction of a new policy P_i over this subject and object can be viewed as an update of the policy P_d to the new policy P_i. Let the access rights specified by policy P_i correspond to Node *n* in $ARL(O_j)$. In this case, *lub*(0, *n*) = *n*. Hence, creation of a new policy can be treated as a policy relaxation.

A *policy deletion operation* is one in which an existing policy $P_i =<$ $S_i, O_i, R_i >$ is deleted. Note that deletion of an existing policy P_i specified over the subject S_i and object O_i can be thought of as modifying the existing policy P_i to the dummy policy P_d which maps S_i to the minimal element (Node 0) in the access rights lattice of object O_i. In this case, $lub(e, 0) \neq 0$; hence, deletion of an existing policy is a policy restriction.

3. TRANSACTION PROCESSING MODEL FOR POLICY UPDATES

Having given some background on policy updates, we now discuss our transaction processing model that is based on the standard transaction processing model [3]. Our database consists of data objects and policy objects. The operations allowed on the data objects are read and write. Policy objects, like data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed.* A policy object P_j is said to be *deployed* if there exists a subject that is currently accessing a object by virtue of the privileges given by policy object P_j. Suppose the policy object P_i allows subject S_j to read object O_k. Subject S_j initiates a transaction T_l that reads O_k. While the transaction T_l reads O_k, we say that the policy object P_i is deployed.

The operations specified on data objects are Read and Write. A policy object is associated with four operations: Read, Deploy, WriteRelax, WriteRestrict. The Write operations on policy object are classified as WriteRelax or WriteRestrict. A WriteRelax operation is one in which the policy gets relaxed. All other write operations on the policy object are treated as WriteRestrict. Two operations are said to *conflict* if both operate on the same object and one of them is a Write operation. The Write operation conflicts with a Read or a Deploy operation on the same object.

We define transaction in the following manner. A *transaction* T_i is a partial order with ordering relation $<_i$ where (1) $T_i \subseteq \{r_i[x], w_i[x] \mid x$ is a data object $\} \cup \{d_i[x], r_i[x], ws_i[x], wx_i[x] \mid x$ is a policy object $\} \cup \{a_i, c_i\}$; (2) $a_i \in T_i$ iff $c_i \notin T_i$; (3) if t is c_i or a_i, for any other operation $p \in T_i, p_i <_i t$; (4) if $r_i[x]$, $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$; (5) if $d_i[x]$, $ws_i[x] \in T_i$, then either $d_i[x] <_i ws_i[x]$ or $ws_i[x] <_i d_i[x]$; and (6) if $d_i[x]$, $wx_i[x] \in T_i$, then either $d_i[x] <_i wx_i[x]$ or $wx_i[x] <_i d_i[x]$. Condition 1 states that the operations allowed on data objects are Read and Write and the operations allowed on policy objects are Read, Deploy, WriteRelax (denoted by *wx*), and WriteRestrict (denoted by *ws*). Condition 2 states that this set contains an Abort or a Commit operation but not both. Condition 3 states that Abort or Commit operation must follow every other operation of the transaction. Condition 4 requires that the partial order $<_i$ specify the order of execution of Read and Write operations on a common data or policy object. Condition 5 specifies that if there is a Deploy operation on a policy object and a WriteRestrict operation on the same object, then the ordering relation $<_i$ must specify the order of the operations. Condition 6 specifies a similar condition for Deploy and WriteRelax operation.

The algorithm that we propose is an extension of the two phase locking protocol [3]. Each data object O_i in our model is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). A policy object P_j is associated with four locks: read lock (denoted by $RL(P_j)$), write relax lock (denoted by $WXL(P_j)$), write restrict lock (denoted by $WSL(P_j)$) and deploy lock (denoted by $DL(P_j)$). The locking rules for data and policy objects are the similar to those in the standard two-phase locking protocol [3]: the same object cannot be locked by different transactions in conflicting modes.

Next we define what it means for a transaction in our model to be well-formed. A transaction is *well-formed* if it satisfies the following conditions: (i) A transaction before reading or writing a data object must deploy the policy object that authorizes the transaction to perform the operation, (ii) A transaction before reading, write relaxing or write restricting a policy object must deploy the policy object that authorizes the transaction to perform the operation. (iii) A transaction before reading or writing a data object must acquire the appropriate lock. (iv) A transaction before deploying, reading, write relaxing, or write restricting a policy object must acquire the appropriate lock. (v) A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode. (vi) All locks acquired by the transaction are eventually released. A well-formed transaction T_i is *two-phase* if all its lock operations precede any of its unlock operations. Consider a transaction T_i that reads object O_j (denoted by $r_i(O_j)$) and then writes object O_k (denoted by $w_i(O_k)$). Policies P_m and P_n authorize the subject initiating transaction T_i, the privilege to read object O_j and the privilege to write object O_k respectively. An example of a well-formed and two-phase execution of T_i consists of the following sequence of operations: $< DL_i(P_m), RL_i(O_j), d_i(P_m), r_i(O_j), DL_i(P_n), WL_i(O_k), d_i(P_n), w_i(O_k), UL_i(P_m), UL_i(P_n), UL_i(O_j), UL_i(O_k) >$, where DL_i, RL_i, WL_i, d_i, r_i, w_i, UL_i denote the operations of acquiring deploy lock, acquiring read lock, acquiring write lock, deploy, read, write, lock release, respectively, performed by transaction T_i.

A transaction is *policy compliant* if for every operation that a transaction performs, there exists a policy that authorizes the transaction to perform the operation for the entire duration of the operation. Note that, all transactions may not be policy compliant. For instance, suppose entity A can execute a long-duration transaction T_i by virtue of policy P_x. While A is executing T_i, P_x changes and no longer allows A to execute T_i. In such a case, if transaction T_i is allowed to continue after P_x has changed, then T_i will not be a policy compliant transaction. Next we define what we mean by a *policy compliant* history. A history is *policy compliant* if all the transactions in the history are policy compliant transactions. A history H in which all the transactions in the committed projection are well-formed and two-phase is conflict-serializable and policy-compliant.

4. CONCURRENCY CONTROL USING SEMANTICS OF POLICY UPDATE

In this section we show how we can use semantics of the policy update operation to increase concurrency. The basic idea is to classify a policy update operation either as a *policy relaxation* or as a *policy restriction* operation. Policy relaxation causes increase in subject's access rights; transactions executing by virtue of a policy need not be aborted when the policy is being relaxed. On the other hand, a policy restriction does not increase the access rights of the subject. To ensure policy-compliant transactions, we must abort the transactions that are executing by virtue of the policy that is being restricted.

The mechanism that we propose is an extension of the two-phase locking protocol. Each data object O_i is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). The locking rules for data objects are the same as in the two-phase locking protocol [3]. Corresponding to the four operations on the policy object, we have four kinds of locks associated with policy objects: read locks (RL), deploy locks (DL), relax locks (WXL) and restrict locks (WSL). The entry in the third column of the fourth row is *Signal*. This is the case of some transaction T_i holding a deploy lock *DL* on a policy object, and another transaction T_j wanting to perform an update causing policy restriction. In this scenario, a signal is generated to abort T_i, after which T_i releases the *DL* lock and T_j is granted the *WSL* lock.

Table 1. **Locking Rules for Policy Objects**

	Wants			
Has	RL	WXL	WSL	DL
RL	Yes	No	No	Yes
WXL	No	No	No	No
WSL	No	No	No	No
DL	Yes	Yes	Signal	Yes

Since the histories generated by the locking rules given in Table 1 may not be conflict serializable, we define another correctness criterion, namely, serializability, and show that our histories do satisfy this new criterion. But first, we need the notion of equivalence. Two histories *H* and H' are *equivalent* if they satisfy the following conditions: (i) they are defined over the same set of transactions, (ii) the execution of *H* on some initial state *S* results in the same final state S' as the execution of H' on the same initial state *S*. A history *H* is serializable if it's committed projection is equivalent to a serial history. Histories generated by the locking rules of Table 1 are serializable and policy-compliant.

5. IMPLEMENTING THE POLICY UPDATE ALGORITHM

In this section, we will discuss how to implement the algorithms proposed in previous section for real-time access control policy update.

5.1 Data Structures for Implementing Policy Updates

A diagram showing the data structures used in implementing the policy update algorithm is shown in Figure 2.

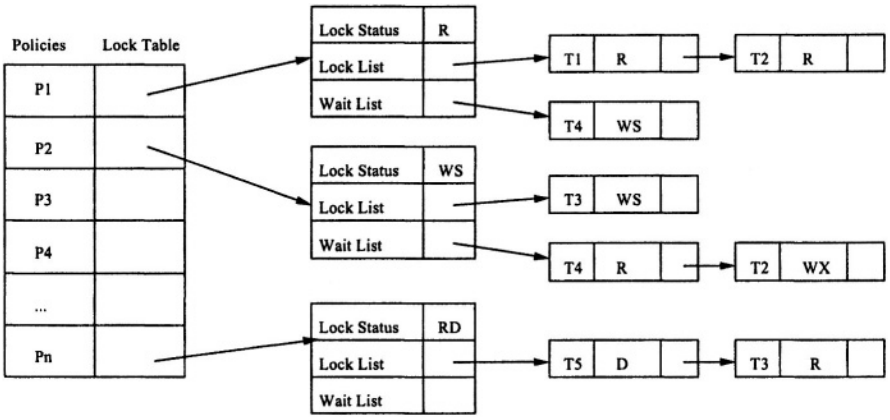

Figure 2. Storing Lock Information for Policy Objects

Policy Object Table This is a hash table containing entries corresponding to each locked policy object and a pointer to the address where the lock information of this policy object is stored. In other words, each entry in this hash table consists of two fields: the first one stores the policy object id and the second one points to the Lock Information Table of the corresponding policy object. The size of this hash table corresponds to the number of policy objects that are locked.

Lock Information Table This table contains all the lock information pertaining to a policy object. It contains an entry for lock status, a pointer to lock list, and a pointer to a wait list. The lock status specifies what modes of locks are currently held on the policy object. The lock list specifies the list of transactions that are currently holding locks on the policy object, and the modes of lock they have on the policy object. The lock list can be implemented as a linked list or hash table. Note that, the lock list also contains the lock status information. The lock status information is used frequently. Traversing the lock list and summarizing the lock status information is time-consuming; hence, we keep this information in a separate field. The wait list specifies the list of transactions that are waiting to lock the policy object and the type of lock they want on the policy object. The wait list can be implemented as a priority queue.

In our algorithm, a policy object is associated with four kinds of locks, namely, read, deploy, relax and restrict, that correspond to Read, Deploy, WriteRelax and WriteRestrict operation performed on policy objects. Some of these locks may be held concurrently and others must be held exclusively. The lock status field in the lock information table summarizes what kind of locks are currently held on a policy object. The lock status field can take on any of the following values: **D**: all the locks held on that policy object are deploy locks; **R**: all the locks held on that policy object are read locks; **RD**: all the locks held on that policy object are read and deploy locks; **WX**: the lock held on that policy object is a relax lock; **WS**: the lock held on that policy object is a restrict lock; and, **DWX**: the locks held on that policy object are relax and deploy locks.

When multiple transactions are waiting to lock a policy object, these transactions are inserted into a wait list. Note that, different transactions have different kinds of priority in our system. For critical applications such as the military, policy updates may have a higher priority than transactions deploying the policy. For this reason, when transactions are inserted into a waiting list we arrange them in the order of their priorities. The next transaction that is given the lock from this wait list is the one with the highest priority. In short, we implement this wait list using a priority queue.

5.2 Algorithms for Implementing Policy Updates

Algorithm 1 Request a lock on policy object
Input: *(i) t: the transaction requesting a lock, (ii) tp: the priority of the transaction requesting the lock, (iii) p: the policy object for which lock is requested, (iv) lm: the lock mode that requested by the transaction, and (v) ht: hash table storing information about policy object.*
Output: TRUE/FALSE: *indicates whether the request was granted or denied.*

```
Procedure RequestPolicyLock(t, tp, p, lm, ht)
begin
    if there is no entry for p in ht
      lt = create(p, t, lm);            /* create a lock table for p */
      insert(ht, p, &lt);              /* add entry for p in hash table */
      return TRUE;
    lt = getLockTable(ht, p);
    if (lt.lockStatus == 'R' or lt.lockStatus == 'RD')
      if (lm == 'D') and (lt.lockStatus ≠ lm)
        lt.lockStatus = 'RD';
      if (lm == 'R' or lm == 'D')
        insert(lt.LockList, t);        /* insert t in the LockList */
        return TRUE;
      if (lm == 'WX' or lm == 'WS')
        insert(lt.WaitList, t, tp);     /* insert t in the WaitList */
        return FALSE;
    else  (lt.lockStatus == 'D' )       /* p has deploy locks */
      if (lm == 'R')
        lt.lockStatus == 'RD'
      if (lm == 'WX')
```

```
            lt.lockStatus == 'DWX'
          if (lm == 'WS' )
            for each transaction $T_i$ in lt.LockList
                abort $T_i$;      /* release all deploy locks */
            delete(lt.lockList);
            lt.lockStatus = 'WS';
          insert(lt.LockList, t);      /* insert t in LockList */
          return TRUE;
        else                /* policy p has a restrict or relax lock */
            insert(lt.Waitlist, t, tp);      /* insert t in WaitList */
            return FALSE;
end
```

Algorithm 2 Release a lock on policy object
Input: *(i) t - the transaction that want to release a lock and (ii) p - the policy object for which lock is being released.*
Output: TRUE/FALSE - *indicates whether the release was successful or not.*

```
Procedure ReleasePolicyLock(t, p)
begin
      if (p is not in ht)           /* indicates p is not locked */
        return FALSE;
      lt = getLockTable(ht, p);
      remove(lt.LockList, t);              /* remove t from LockList */
      if (lt.LockList == NULL) /* if no other transactions have a lock on p*/
        if (lt.WaitList == NULL) /* if no transactions are waiting to lock p*/
          remove(ht, p);
        else
            request lock for next transaction in wait list
      else                   /* lock list is not empty */
                       /* update lock status for the policy object */
          read = false; deploy = false; relax = false
          for each transaction $T_i$ in lt.LockList do
              if $T_i$.mode == 'R'
                read = true;
              else if $T_i$.mode == 'D'
                    deploy = true;
              else if $T_i$.mode == 'WX'
                    relax = true;
          if (read == true) and (deploy == false) and (relax == false)
            lt.LockStatus = 'R';
        else if (read == true) and (deploy == true) and (relax == false)
            lt.LockStatus = 'RD';
        else if (read == false) and (deploy == true) and (relax == false)
            lt.LockStatus = 'D';
        else if (read == false) and (deploy == true) and (relax == true)
            lt.LockStatus = 'DWX';
        else if (read == false) and (deploy == false) and (relax == true)
            lt.LockStatus = 'WX';
    return TRUE;
end
```

6. RELATED WORK

Although a lot of work appears in the area of security policies (please refer to Damianou's thesis [6] for a survey), policy updates have received relatively little attention. Some work has been done in identifying interesting adaptive policies and formalization of these policies [7, 16]. A separate work [15] illustrates the feasibility of implementing adaptive security policies. The above works pertain to multilevel security policies encountered in military environments; the focus is in protecting confidentiality of data and preventing covert channels. We consider a more general problem and our results will be useful to both the commercial and military sector.

In an earlier work, Ray and Xin [14] have proposed algorithms for real-time update of access control policies. The current work enhances one of these algorithms by providing the implementation details and also discusses how to handle addition and deletion of policies. In a separate work Ray [13] shows how the semantics of transactions can provide more concurrency for real-time update of access control policies.

Automated management of security policies for large scale enterprise has been proposed by Damianou [5]. This work uses the PONDER specification language to specify policies. The simplest kinds of access control policies in PONDER are specified using a *subject-domain, object-domain* and *access-list.* The subject-domain specifies the set of subjects that can perform the operations specified in the access-list on the objects in the object-domain. This work describes the implementation of a basic toolkit. The toolkit has a high-level language editor for specifying policies, a compiler for translating policies into enforcement components objected to different platforms, a browser to view and manipulate the domains of subjects and objects to which policies apply. Thus, new subjects can be added to the subject-domain or subjects can be removed from the subject-domain. The object-domain can also be changed in a similar manner. But this work does not allow the policy specification itself to change. An example will help illustrate this point. Suppose we have a policy in PONDER that is implementing Role-Based Access Control: *subject-domain = Manager, object-domain = /usr/local, access-list = read, write.* This policy allows all *Managers* to *read/write* all the files stored in the directory */usr/local.* Now the toolkit will allow adding/removing users from the domain *Manager,* adding/deleting files in the domain */usr/local.* However, it will not allow the policy specification to be changed. For example, the subject-domain cannot be changed to *Supervisors.* Our work, focuses on the problem of updating the policy specification itself and complements the above mentioned work.

Concurrency control in database systems is a well researched topic. Some of the important pioneering works have been described by Bernstein et al. [3]. Thomasian [18] provides a more recent survey of concurrency control methods and their performance. The use of semantics for increasing concurrency has also been proposed by various researchers [1, 2, 8–11, 17].

7. CONCLUSION AND FUTURE WORK

Real-time updates of policy is an important problem for both the commercial and the military sector. In this paper we focus on real-time update of access control policies in a database system. A database consists of data objects that are accessed and modified through transactions. We consider an environment in which transactions execute concurrently some of which are policy update transactions. In an earlier work we proposed an algorithm that allows real-time update of policies. In this work we extend our approach to handle addition and deletion of policies and describe properties of histories generated by our algorithm. We also give the implementation details of our algorithm.

In this work we assume there exists exactly one policy by virtue of which any subject has access privilege to some object. In a real-world scenario multiple policies may be specified over the same subject and object. The net effect of these multiple policies depend on the semantics of the application. Changing the policies in such situations is non-trivial. Moreover, precedence relationships may be specified over the different policies. Update of policies may change these precedence relationship. In future, we plan to propose algorithms that address these issues.

In future we plan to extend our approach to handle more complex kinds of authorization policies, such as, support for negative authorization policies, incorporating conditions in authorization policies, support for specifying priorities in policies. Specifically, we plan to investigate how policies specified in the PONDER specification language [4] can be updated.

References

[1] P. Ammann, S. Jajodia, and I. Ray. Applying Formal Methods to Semantic-Based Decomposition of Transactions. *ACM Transactions on Database Systems,* 22(2):215–254, June 1997.

[2] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems,* 17(1):163–199, March 1992.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop,* Bristol, U.K., January 2001.

[5] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman. Tools for Domain-based Policy Management of Distributed Systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium,* Florence, Italy, April 2002.

[6] N. C. Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, London, U.K., 2002.

[7] J. Thomas Haigh et al. Assured Service Concepts and Models: Security in Distributed Systems. Technical Report RL-TR-92-9, Rome Laboratory, Air Force Material Command, Rome, NY, January 1992.

[8] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems,* 8(2): 186–213, June 1983.

[9] M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences,* 43(1):25–61, August 1991.

[10] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-ouration transaction systems using the NT/PV model. *ACM Transactions on Database Systems,* 19(3):492–535, September 1994.

[11] Nancy A. Lynch. Multilevel atomicity—A new correctness criterion for database concurrency control. *ACM Transactions on Database Systems,* 8(4):484–502, December 1983.

[12] J. Park and R. Sandhu. Towards Usage Control Models: Beyond Traditional Access Controls. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies,* pages 57–64, Monterey, California, June 2002.

[13] I. Ray. Real-Time Update of Access Control Policies. *Information and Software Technology,* 2004. To appear.

[14] I. Ray and T. Xin. Concurrent and Real-Time Update of Access Control Policies. In *Proceedings of the 14th International Conference on Database and Expert Systems,* volume 2736 of *Lecture Notes in Computer Science,* pages 330–339, Prague, Czech Republic, September 2003. Springer-Verlag.

[15] E. A. Schneider, W. Kalsow, L. TeWinkel, and M. Carney. Experimentation with Adaptive Security Policies. Technical Report RL-TR-96-82, Rome Laboratory, Air Force Material Command, Rome, NY, June 1996.

[16] E. A. Schneider, D. G. Weber, and T. de Groot. Temporal Properties of Distributed Systems. Technical Report RADC-TR-89-376, Rome Air Development Center, Rome, NY, September 1989.

[17] L. Sha, J. P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers,* 37(2): 146–159, February 1988.

[18] A. Thomasian. Concurrency Control: Methods, Performance and Analysis. *ACM Computing Surveys,* 30(1):70–119, 1998.