A Lattice-Based Approach for Updating Access Control Policies in Real-Time

Tai Xin Indrakshi Ray

Department of Computer Science

Colorado State University

Email: {xin,iray}@cs.colostate.edu

Note: A preliminary version of this paper under the title "Concurrent and Real-time Update of Access Control Policies" appeared in DEXA 2003.

Keywords: *security policies, concurrency control, transaction management*

Contact Author Address: Indrakshi Ray

Department of Computer Science

Colorado State University

Fort Collins, CO 80523-1873

Email: `iray@cs.colostate.edu`

Phone: (970) 491-7986, Fax: (970) 491-2466

A Lattice-Based Approach for Updating Access Control Policies in Real-Time

Abstract

Real-time update of access control policies, that is, updating policies while they are in effect and enforcing the changes immediately and automatically, is necessary for many dynamic environments. Examples of such environments include disaster relief and war zone. In such situations, system resources may need re-configuration or operational modes may change, necessitating a change of policies. For the system to continue functioning, the policies must be changed immediately and the modified policies automatically enforced. In this paper, we propose a solution to this problem – we consider real-time update of access control policies in the context of a database system.

In our model, a database consists of a set of objects that are read and updated through transactions. Access to the data objects are controlled by access control policies which are stored in the form of policy objects. We consider an environment in which different kinds of transactions execute concurrently; some of these may be transactions updating policy objects. Updating policy objects while they are deployed can lead to potential security problems. We propose algorithms that not only prevent such security problems, but also ensure serializable execution of transactions. The algorithms differ on the degree of concurrency provided and the kinds of policies each can update.

1 Introduction

Access control policies protect information resources from unauthorized access. 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 adhoc 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 adhoc operations but done through well-defined *transactions* that have been previously analyzed. Moreover, such updates should be carried out only by security administrators or other high-ranking personnel.

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 is changed while it is in effect and this change needs to be enforced immediately. Such real-time updates of access control policies are needed by dynamic 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. The updated policies should be automatically enforced.

Real-time updates of access control policies may also be needed in commercial environments. 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 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.

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 are 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. Allowing the transactions to execute in cases where the modified policy no longer gives these transactions the execution privileges results in a security breach. We propose different algorithms that allow for concurrent and real-time updates of policies and at the same time prevent such security breaches. The algorithms differ with respect to the degree of concurrency achieved and the kinds of policies that can be updated.

The first algorithm is a very simple one – any time a deployed policy is modified, transactions executing by virtue of that policy are aborted. This algorithm, although easy to implement, results in unnecessary aborts. For example, the update of policy may not be restricting privileges or may be removing privileges that does not affect the transaction that deploys this policy. Since we are dealing with critical transactions, aborting them unnecessarily is not desirable.

The second algorithm relies on using a lattice-based approach to categorize policy update transactions as policy relaxations or policy restrictions. Policy relaxations increase the access control privileges of a subject. A policy update that is not a policy relaxation is treated as a policy restriction. Policy relaxation, unlike restriction, does not require abort of transactions that are executing by virtue of the policy. The lattice-based approach allows one to syntactically determine if the policy update is a relaxation or restriction.

The last algorithm deals with the situation when multiple policies are specified over a subject and an object and priorities are specified with policies. A policy update may change the access rights associated with the policy or its priority. Here again, we use a lattice-based approach to determine if the policy update is a restriction or relaxation. The interesting thing to note is that a policy relaxation does not impact transactions executing by virtue of that policy but may require abort of transactions executing by virtue of other policies that are specified over the same subject and object.

The rest of the paper is organized as follows. Section 2 introduces our model. Section 3 describes a simple concurrency control algorithm for policy updates. Section 4 illustrates how the semantics of the policy update operation can be exploited to increase concurrency. Section 5 handles policy updates for the cases where multiple policies are specified over a subject and an object and priorities are specified on these policies. Section 6 highlights the related work. Section 7 concludes our paper with some pointers to future directions.

2 Our Model

A *database* is specified as a collection of objects together with a set of *integrity constraints* 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. Integrity constraints are predicates defined over 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[1]. An authorization policy specifies what operations an entity can perform on another entity. We focus our attention to systems that use the closed policy assumption and 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 begin by considering simple kinds of authorization policies that are specified by *subject*, *object*, and *operations*. A subject can be a user, a group of users or a process. An object, in our model, is a data object or a group of data objects. A subject can perform only those operations on the object that are specified in the operations.

Definition 1 [Policy] A *policy* is a function that maps a subject and a object to a set of operations. We formally denote this as follows: $P : S \times O \to \mathbb{P}(R)$ 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 operations.

In a database, policies are stored in the form of policy objects.

Definition 2 [Policy Object] 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 operations of the policy respectively. Subject S_i can perform only those operations on the object O_i that are specified in R_i.

Example 1 Let $P =< John, FileF, \{r, w, x\} >$ be a policy object. This policy object gives subject John the privilege to Read, Write, and Execute $FileF$.

A policy allows a subject to perform a set of operations on an object. Updating this policy will allow the subject to perform a different set of operations on the object. The algorithms that we propose in the later sections use knowledge of the kind of policy update. To better understand the effect of a policy

[1]Henceforth, we use the term policy or access control policy to mean authorization policy.

update operation, we need to represent the different sets of allowable operations on an object. We do this by representing the access rights associated with an object in the form of a lattice. Later in Section 4, we show how using the lattice gives us a formal way of classifying the policy update operations.

Definition 3 [Representing the Access Right of an Object] Let $\mathbf{Op_i}$ = $\{op_1, op_2, \ldots, op_n\}$ 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 $< op_1, op_2, \ldots, op_n >$. We represent any access right on the object O_i as an n-element vector $[i_1 i_2 \ldots i_n]$. In some access right R_j, if the k-th element of this vector is 0, (that is, $i_k = 0$) then R_j does not allow the operation op_k to be performed on the object O_i. When the k-th element equals 1 (that is, $i_k = 1$) in some access right R_m, it signifies that the access right R_m allows operation op_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.

Example 2 Let $< r, w, x >$ be the operations defined 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.

Definition 4 [Partial Order of Access Rights of an Object] The set of all access rights associated with an 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. $R_p \geq_{O_i} R_q$ holds if and only if $R_p[i_k] = R_q[i_k]$ or $R_p[i_k] > R_q[i_k]$, for all $k = 1 \ldots n$.

Definition 5 [Least Upper Bound Operation] 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.

Definition 6 [Greatest Lower Bound Operation] 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.

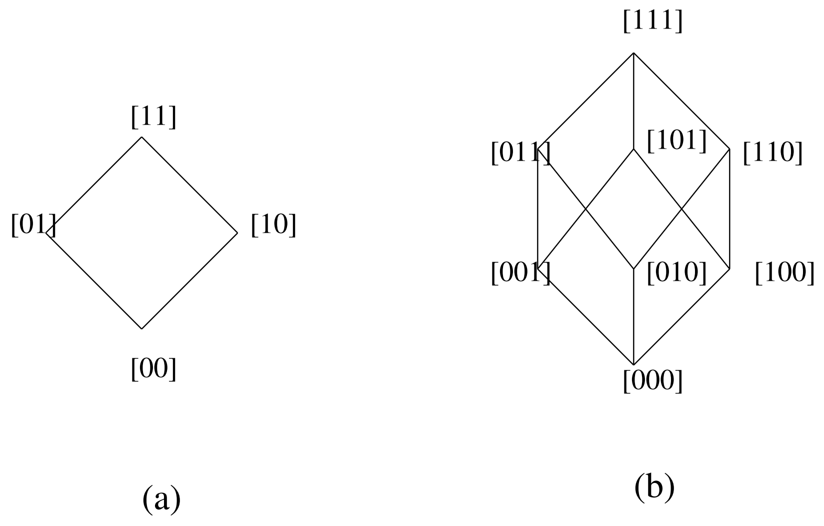

Figure 1: Representing Possible Access Control Rights of Objects

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.

Definition 7 [Access Rights Lattice of an Object] The set of all possible access rights on an 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 an 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 privileges.

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 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 an object having three operations.

A policy now can be defined 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: $P : S \rightarrow (ARL(O))$.

Definition 8 [Policy Update] 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.

In this paper, we only focus on policy updates that change access privileges. We do not consider the operations that change the subject or the object in a policy. Having given some background on the policies, we are now in a position to discuss policy objects. Recall from Definition 2 that policies are stored in the database in the form of policy objects. Next we describe the operations associated with the policy objects. Policy objects, like data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed*.

Definition 9 [Deploy] A policy object P_j is said to be *deployed* if there exists a subject that is currently accessing an object by virtue of the privileges given by the policy object P_j.

Example 3 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.

Note that, not all entities can execute a deploy operation on a policy object. Only the subject specified in a policy can execute a deploy operation on the policy. A policy object, during its lifetime, can be in one of two states: *inactive* and *deployed*. The policy object enters the inactive state after it has been created. When some transaction T_i executes a deploy operation on a policy object, it enters the deployed state. During this state, other transactions may also deploy this policy object. When all the transactions deploying this policy object have committed or aborted, the policy object returns to the inactive state.

The environment we are considering is one in which multiple users will be accessing and modifying data and policy objects, while the policy objects are deployed. To deal with this scenario, we need some concurrency control mechanism. The objectives of our concurrency control mechanism are the following.

- Allow concurrent access to data objects and policy objects.
- Prevent security violations arising due to policy updates.

We now present a simple concurrency control algorithm for updating policies.

3 A Simple Algorithm for Policy Updates

In this section, we present a simple algorithm for policy updates. We describe the algorithm briefly to motivate our work. We refer the interested reader to our earlier works [33, 34] for more details.

We assume that each data object is associated with two operations: Read and Write. A policy object is associated with three operations: Read, Write and Deploy. We begin by giving some definitions.

Definition 10 [Conflicting Operations] Two operations are said to *conflict* if both operate on the same data object and one of them is a Write operation.

The Write operation conflicts with a Read or a Deploy operation on the same object.

Definition 11 [Transaction] 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 or policy object $\} \cup \{d_i[x] \mid x$ 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 $p \in T_i$, $p_i <_i t$; and
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 $d_i[x]$, $w_i[x] \in T_i$, then either $d_i[x] <_i w_i[x]$ or $w_i[x] <_i d_i[x]$.

Condition 1 defines the different kinds of operations in the transactions ($r_i[x]$, $w_i[x]$, $d_i[x]$, a_i, c_i denote Read operation on object x, Write operation on x, Deploy operation on x, Abort or Commit operation respectively). 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 requires that the partial order $<_i$ specify the order of execution of Deploy and Write operations on a common policy object.

The algorithm that we propose is an extension of the two phase locking protocol [6]. 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 $WL(O_i)$). The locking rules for data objects are the same as the standard two-phase locking protocol [6]. A policy object P_j is associated with three locks: read lock (denoted by $RL(P_j)$), write lock (denoted by $WL(P_j)$) and deploy lock (denoted by $DL(P_j)$).

The locking rules for data objects are the same as the standard two-phase locking protocol [6]. A policy object P_j is associated with three locks: read lock (denoted by $RL(P_j)$), write lock (denoted by $WL(P_j)$) and deploy lock (denoted by $DL(P_j)$).

	Wants		
Has	RL	WL	DL
RL	Yes	No	Yes
WL	No	No	No
DL	Yes	Signal	Yes

Table 1: Locking Rules for Policy Objects

The locking rules for the policy objects are given in Table 1. *Yes* entry in the lock table indicates that the lock request is granted. *No* entry indicates that the lock request is denied. *Signal* entry in the lock table indicates that the lock request is granted, but only after the transaction currently holding the lock is aborted and the lock is released. The first row corresponds to the case where some transaction has a read lock on a policy object. The first column in the first row is a *Yes* – another transaction requesting a read lock on the same object is given the lock. The second column in the first row is a *No* – another transaction requesting a write lock on the same object is not given the lock. The entry in the first row third column is a *Yes* – this means, that if a transaction has a read lock on a policy object, and another transaction requests a deploy lock on the same object, then this lock request is granted. The second row corresponds to the case where a transaction has a write lock on a policy object. This row has all No entries; this means that no other transaction will be given any other locks to the object. Now consider the third row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and third columns of the third row is a Yes signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the third row is $Signal$. Signal means that the lock request is granted after the transaction currently holding the lock is aborted. For example, suppose some transaction T_i holds a deploy lock DL on a policy object, and another transaction T_j wishes to get the write lock WL and update the policy object. In such a case a signal is generated to abort T_i, after which T_i releases the DL lock and T_j is granted the WL lock. Note that, a transaction updating a policy (T_j) has a higher priority than a transaction (T_i) that deploys this policy; hence, we generate a signal to abort the transaction deploying the policy (T_i).

Next we define what it means for a transaction in our model to be well-formed.

Definition 12 [Well-formed Transaction] A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data or policy object must deploy the policy object that authorizes the transaction to perform the operation.
2. A transaction before deploying, reading, or writing a policy object must acquire the appropriate lock.
3. A transaction before reading or writing a data object must acquire the appropriate lock.
4. A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode.
5. All locks acquired by the transaction are released when the transaction completes.

Definition 13 [Well-formed Two-phase Transaction] A well-formed transaction T_i is *two-phase* if all its lock operations precede any of its unlock operations.

Example 4 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.

Definition 14 [Policy-Compliant Transaction] 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.

We borrow the definitions of *history*, *conflict equivalence*, *committed projection of a history*, *serial history*, *serialization graph*, and *conflict serializable history* from Bernstein et al. [6].

Theorem 1 A history is serializable iff its serialization graph is acyclic.

Proof 1 This proof is identical to the proof of the same theorem that is given in Bernstein et al. [6].

Next we define what we mean by a *policy-compliant* history.

Definition 15 [Policy-compliant] A history is *policy-compliant* if all the transactions in the history are policy-compliant transactions.

Theorem 2 A history H in which all the transactions in the committed projection of the history H are well-formed and two-phase is conflict serializable.

Proof 2 We prove this by contradiction. Assume that the history H produced by transactions $\{T_1, T_2, \ldots, T_n\}$ is not serializable. The graph produced from this history contains a cycle. Without loss of generality, assume that this cycle is $T_1 \rightarrow T_2 \rightarrow T_3 \ldots T_n \rightarrow T_1$. The presence of the arrow $T_1 \rightarrow T_2$, signifies that there is an operation in T_1 that conflicts with and precedes another operation in T_2. The unlock operation in T_1 must precede the lock operation in T_2 (this is because the data object involved in a conflicting operation can be locked by only one transaction at any time). That is, $U_1(O_a) < L_2(O_a)$.

Using similar arguments, we can argue that for the edge $T_2 \rightarrow T_3$, there is an unlock operation in T_2 that precedes a lock operation in T_3. That is, $U_2(O_b) < L_3(O_b)$. Since transaction T_2 is two-phase, $L_2(O_a) < U_2(O_b)$. Therefore, we can conclude that $U_1(O_a) < L_3(O_b)$. This argument can be extended and we can arrive at the conclusion that $U_1(O_a) < L_1(O_k)$. This is not possible because T_1 is two-phase. Thus, we arrive at a contradiction. Hence, our initial assumption that the history is not serializable is wrong. Therefore, the history H is serializable.

Theorem 3 A history H in which all the transactions in the committed projection are well-formed and two-phase is policy-compliant.

Proof 3 Assume that the history H is not policy-compliant. This means that one or more transactions in the history H are not policy-compliant. Suppose T_i is one such transaction. Without loss of generality, assume that the transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i is well-formed, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for the policy object. In other words, before T_i can access O_r, it has to obtain the deployment lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy P_m got deleted or modified. But according to well-formed rules this is not possible. Any transaction T_j modifying the policy P_m has to obtain a write lock (WL) on P_m. Before the write lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (Table 1). Thus, the above scenario of T_i updating O_r without any policy authorizing T_i to do so, does not arise in our case. Therefore, T_i is policy-compliant. Our assumption, that the history H is not policy-compliant, is wrong.

Although the lock based concurrency control approach provides policy-compliant and serializable schedule, it is overly restrictive. A change of policy may not change the specific subject's access privileges or may result in increased access privileges; in such cases terminating valid access will result in poor performance. This motivates us to propose the following approach.

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. Before going into the details, we first give a definition of policy relaxation and policy restriction.

Definition 16 [Policy Relaxation Operation] 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 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 $lub(k,j) = j$.

Example 5 Let the operations defined on *FileF* be $< r, w, x >$. Suppose 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.

Definition 17 [Policy Restriction Operation] 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 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 $lub(k,j) \neq j$.

Example 6 Let the operations defined on *FileF* be $< r, w, x >$. Suppose 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.

4.1 Concurrency Control Based on Knowledge of Policy Change

We now give a concurrency control algorithm that uses the knowledge of the kind of policy change. The operations specified on data objects are Read and Write. A policy object is now associated with four operations: Read, Deploy, WriteRelax, WriteRestrict. The Read and Deploy operations are similar to those specified in Section 3. 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. Since the operations are different than those discussed in Section 3, we modify the definition of transaction given in Definition 11 to the following:

Definition 18 [Transaction] A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data object }\} \cup \{d_i[x], r_i[x], ws_i[x], wx_i[x] \mid x \text{ is a policy object }\} \cup \{a_i, c_i\}$;
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 $p \in T_i$, $p_i <_i t$; and
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 $ws_i[x] <_i d_i[x]$.
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 is changed from that in Definition 11 to reflect 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). Conditions 2,3, and 4 are the same as given in Definition 11. Condition 5 given in Definition 11 is no longer applicable as there is no simple Write operation on policy objects; this condition is replaced by two conditions (Conditions 5 and 6 in Definition 18). Conditions 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 definition of well-formed transaction is changed as follows:

Definition 19 [Well-formed Transaction] A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data object must deploy the policy object that authorizes the transaction to perform the operation.
2. 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.
3. A transaction before reading or writing a data object must acquire the appropriate lock.
4. A transaction before deploying, reading, write relaxing, or write restricting a policy object must acquire the appropriate lock.
5. A transaction cannot acquire a lock on a policy object or data object if another transaction has some kind of a write lock on the object.
6. All locks acquired by the transaction are released when the transaction completes.

To ensure serializable and policy-compliant histories, we require each transaction to be well-formed (Def. 19) and two-phase (Def. 13).

We next present our locking mechanism. 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 [6]. 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 locking rules are given in the table 2. Let us consider the fourth row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and fourth columns of the fourth row is a Yes signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the fourth row is a Yes – if a transaction has a deploy lock on a policy object and another transaction wants a relax lock on the same object, then the lock request is granted. Thus, if a transaction T_m holds a deploy lock DL on a policy object P_j, and another transaction T_n wants to perform a policy relaxation on P_j, then T_n is granted the WXL lock. Note that T_m does not have to be aborted in this case because the policy P_j that is deployed by T_m is being relaxed. 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.

	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

Table 2: Locking Rules for Policy Objects

Note that our algorithm does not generate well-formed transactions. Specifically, it violates Condition 5 given in Definition 19. Thus, it does not generate conflict-serializable histories. We define an alternate correctness criterion, namely, serializability, and show that our histories do satisfy this new criterion.

Definition 20 [Equivalence of Histories] Two histories H and H' are equivalent if they satisfy the following conditions:

1. they are defined over the same set of transactions,

2. 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.

Definition 21 [Serializable History] A history H that is equivalent to a serial history H_s is a serializable history.

Theorem 4 Histories generated by the locking rules of Table 2 are serializable.

Proof 4 Our mechanism generates histories that may not be conflict-serializable. Non conflict-serializable execution occurs because the locking rules allow two transactions to hold conflicting locks on a policy object at the same time. Specifically, the rules allow some transaction T_i to hold a deploy lock on a policy object P_q, while another transaction, say, T_j acquires a relax lock on P_q to increase the privileges. This situation is not problematic and the effect of the execution of these two transactions is equivalent to the serial execution of T_i followed by T_j. Any history H containing such transactions T_i and T_j can therefore be converted to an equivalent serial history.

Theorem 5 Any history H generated by this concurrency control mechanism is policy-compliant.

Proof 5 Assume that the history H is not policy-compliant. This means that one or more transaction in the history H is not policy-compliant. Suppose transaction T_i is not policy-compliant. Without loss of generality, assume that the transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i satisfies all but Condition 4 of the Definition 12 , it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for the policy object. In other words, before T_i can access O_r, it has to obtain the deploy lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy P_m got modified such that T_i no longer has the privilege to update O_r. But this scenario cannot occur according to our algorithm. Any transaction T_j modifying the policy P_m has to obtain a write lock (WL) on P_m. Before the write lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (because T_j restricts the policy). Thus, the above scenario of T_i updating O_r without any policy authorizing T_i to do so, does not arise in our case. Therefore, T_i is policy-compliant. Our assumption, that the history H is not policy-compliant, is wrong.

4.2 Creation and Deletion of Policies

We consider a system having positive authorization policies only. The absence of any policy on a specified subject S_i and object O_j indicates that S_i cannot perform any operation on O_j.

Theorem 6 Creating a new policy $P_n =< S_i, O_j, R_i >$ corresponds to a policy relaxation operation.

Proof 6 The absence of any policy on subject S_i and object O_j can be represented as the existence of a dummy policy P_d that maps the subject S_i to the minimal element in the access rights lattice of the object O_j (indicated by Node 0). The introduction of a new policy P_n over this subject and object can be viewed as an update of the policy P_d to the new policy P_n. Let the access rights specified by policy object P_n correspond to Node n in $ARL(O_j)$. In this case, $lub(0,n) = n$. Hence, this is an example of policy relaxation.

Thus, when a transaction creates a policy object, it acquires a write relax lock on the policy object. The transaction after completing relaxes this lock. Since no other operations can be performed on the policy object before it is created, no other transactions will have any lock on this policy object when the policy creation transaction executes. Thus, when a new policy is being added, existing transactions are not affected.

Theorem 7 Deleting an existing policy $P_e =< S_i, O_j, R_i >$ can be thought of as a policy restriction operation.

Proof 7 Deletion of an existing policy P_e specified over the subject S_i and object O_j can be thought of as modifying the existing policy P_e to the dummy policy P_d which maps S_i to Node 0 in $ARL(O_i)$. In this case, $lub(e,0) \neq 0$; hence, this is policy restriction.

The transaction, say T_i, that deletes the policy object P_e must acquire a write restrict lock on this policy object. Other transactions may be executing by virtue of this policy object P_e. Such transactions are aborted when T_i acquires the write restrict lock on P_e. In short, when a policy is deleted, transactions executing by virtue of this policy is aborted by the concurrency control mechanism.

5 Updating Policies Specified with Priorities

When different policies are specified over a given subject and an object, we may not want all the policies to be enforced at all times. In such cases, policies may be associated with priorities. The context or the environment will determine the priority of the policies. The understanding is that the policy with the highest priority is the one that must be deployed; this policy determines the rights a subject has on the object. In other words, priority is used to determine whether a policy is deployable or not.

Priorities can be specified in two ways: relative or absolute priorities. In relative specification, the policies are ordered according to their priorities. In absolute specification, the set of policies is fixed and any policy can have a priority value from this set. We assume the latter and each policy has a priority value taken from the set of priorities.

Definition 22 [Policy Object with Priority] A policy object with priority is defined as a quadruple $P_i =< S_i, O_i, R_i, pr_i >$ where S_i, O_i, R_i, pr_i denote the subject, object, operations, and priority of policy P_i respectively. pr_i can take any value from the set of priorities defined below.

Definition 23 [Priority Set] Each application has a priority set PR that defines the domain of priorities. That is, it contains all the possible values that can be taken by the priority of any policy in the application. The elements in the set are totally ordered and the ordering relation is denoted by $<$. Consider two policies P_i and P_j having priorities pr_i and pr_j respectively. $pr_i < pr_j$ signifies that policy P_i has a lesser priority than policy P_j or P_j has a higher priority than P_i. Since the elements of this set can be totally ordered, this set can be represented as a lattice which we term *priority lattice*.

Example 7 Suppose in an organization the set of priorities PR is defined as follows $PR = \{Low, High\}$. Thus, the priority for a policy can take on either of the two values Low or $High$.

Let us once again consider the access rights associated with each object. For an object O_i with n possible operations, there are 2^n possible access rights where each access right is represented as per Definition 3. If there are l elements in the priority set PR, then there are $l * 2^n$ possible priority and access rights pairings. These $l * 2^n$ pairs form a partially ordered set as defined below.

Definition 24 [Partial Order of (Access Right, Priority) Pairings of an Object] The set of all ordered pairs of access rights and priorities associated with an object O_i forms a partial order with the ordering relation $\preceq$. Each element e_i in this set is represented as an ordered pair (l_i, r_i) where l_i denotes the priority level and r_i denotes the access right represented according to Definition 3. For any two elements, (l_i, r_i) and (l_j, r_j), if $(l_i, r_i) \preceq (l_j, r_j)$, then the following condition holds: $l_i \leq l_j$ and $r_i \wedge r_j = r_i$ where $l_i \leq l_j$ signifies that either $l_i < l_j$ or $l_i = l_j$.

We next define least upper bound and greatest lower bound operations on this set.

Definition 25 [Least Upper Bound Operation] Given two elements (l_i, r_i) and (l_j, r_j) where each element describes the priority, access right of some object O, the *least upper bound* of (l_i, r_i) and (l_j, r_j) is computed as follows:
$lub((l_i, r_i), (l_j, r_j)) = (l_k, r_i \vee r_j)$ where $l_k = max(l_i, l_j)$ where $max(l_i, l_j)$ indicates the priority that is higher in the set $\{l_i, l_j\}$.

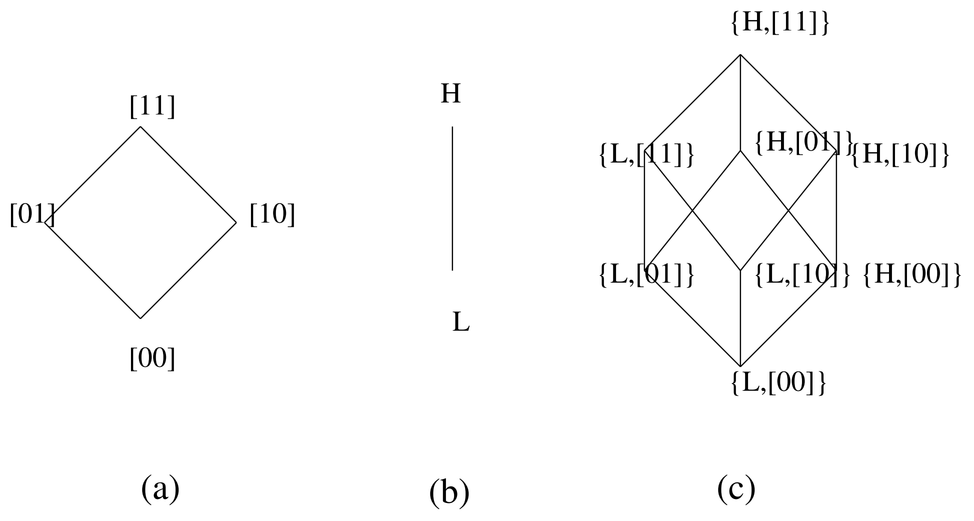

Figure 2: (a) Access Control Rights Lattice of O_i, (b) Priority Lattice PR, and (c) $PARL(O_i, PR)$

Definition 26 [Greatest Lower Bound Operation] Given two elements (l_i, r_i) and (l_j, r_j) where each element describes the priority, access right of some object O, the *greatest lower bound* of (l_i, r_i) and (l_j, r_j) is computed as follows:
$glb((l_i, r_i), (l_j, r_j)) = (l_k, r_i \wedge r_j)$ where $l_k = min(l_i, l_j)$ where $min(l_i, l_j)$ indicates the priority that is lower in the set $\{l_i, l_j\}$.

Since each pair of elements (where each element is an ordered pair of priority and access right) associated with an object has a unique least upper bound and a unique greatest lower bound, the priority-access right pairs of an object can be represented as a lattice.

Definition 27 [Priority Access Right Lattice of an Object] The set of all possible priority-access right pairs of an object O_i and priority set PR can be represented as a lattice which we term *priority access right lattice of object* O_i. The set of all elements in the priority access rights lattice of an object O_i is denoted as $PARL(O_i, PR)$.

Example 8 Consider the example shown in Figure 2. Figure 2(a) shows the access rights lattice of an object O_i having two operations. Figure 2(b) shows the priority lattice PR for the case where there are only two priorities L and H. Figure 2(c) shows the priority access right lattice $PARL(O_i, PR)$ obtained from combining these two lattices.

A policy $P_i =< S_i, O_i, R_i, pr_i >$ maps a subject S_i to a node in the priority access right lattice of object O_i. This is formally stated as follows: $P : S \mapsto (PARL(O, PR))$ where P is the policy function, S is the set of subjects, $PARL(O, PR)$ denotes the set of all priority-access right pairs for all objects.

When priorities are specified on policies, not all policies can be deployed at a given instant of time. We next define the notion of deployable policies. Stated informally, the set of deployable policies are those having the highest priority.

Definition 28 [Deployable Policies] Let $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of policies defined over the same subject S and object O where $P_i =< S, O, R_i, pr_i >$ for $1 \leq i \leq n$. Let $pr_{max} = max(pr_1, pr_2, \ldots, pr_n)$ where max equals maximum of the set of priorities of policies in **P**. The set of deployable policies **D** satisfies the following conditions:

1. $\mathbf{D} \subseteq \mathbf{P}$
2. $\forall P_i \in \mathbf{D}$, $pr_i = pr_{max}$

Recall from Condition 1 of Definition 19, that each transaction before reading or writing a data object must deploy the policy object before performing the authorization. When priorities are specified on policies, a transaction can only deploy deployable policies. That is, a transaction can obtain a deploy lock on a policy object only if it is deployable.

When multiple policies are specified on a subject and an object, the maximum privilege that a subject can have depends on the *resultant policy*. The resultant policy is obtained by combining all the privileges of deployable policies specified over the same subject and object. This is formalized below.

Definition 29 [Resultant Policy] Let $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of policies defined over the same subject S and object O where $P_i =< S, O, R_i, pr_i >$ for $1 \leq i \leq n$. Let **D** be the set of policies in **P** which can be deployed where $\mathbf{D} = \{P_{d1}, P_{d2}, \ldots, P_{dm}\}$. Let $P_r =< S, O, R_r, pr_r >$ be the resultant policy. The priority of the resultant policy pr_r equals pr_{d1}, that is, $pr_r = pr_{d1}$, where pr_{d1} is the priority of some deployable policy. The operations specified by the resultant policy, R_r, is given by $R_r = lub(R_{d1}, R_{d2}, \ldots, R_{dm})$.

Example 9 Let $P_i =< S, O, 01, H >$ and $P_j =< S, O, 11, L >$. The resultant policy $P_r =< S, O, 01, H >$. In this case only P_i is the deployable policy.

Definition 30 [Policy Update] A policy update is an operation that changes some policy $P_i =< S_i, O_i, R_i, pr_i >$ to P'_i where P'_i is obtained by changing R_i to R'_i and/or by changing pr_i to pr'_i. Let (pr_i, R_i), and (pr'_i, R'_i) be mapped to Node i, Node i' of $PARL(O_i, PR)$ respectively. The update of policy object P_i changes the mapping of the subject S_i's access privilege from Node i to Node i' in the priority access rights lattice of object O_i.

Definition 31 [Policy Relaxation Operation] A *policy relaxation operation* is a policy update that increases the access rights of the subject and/or increases the priority of the policy. Let the policy object $P_i =< S_i, O_i, R_i, pr_i >$ be changed to $P'_i =< S_i, O_i, R'_i, pr'_i >$. Let (pr_i, R_i), (pr'_i, R'_i) be mapped to the nodes k, j respectively in $PARL(O_i, PR)$. A policy update operation is a policy relaxation operation if $lub(k, j) = j$.

Since only policies with the highest priorities can be deployed, increasing the priority of the policy may cause the policy to become deployable which results in increased access privilege of some subject.

Definition 32 [Policy Restriction Operation] A *policy restriction operation* is a policy update operation that is not a policy relaxation operation. Let the policy object $P_i =< S_i, O_i, R_i, pr_i >$ be changed to $P'_i =< S_i, O_i, R'_i, pr'_i >$. Let (pr_i, R_i), (pr'_i, R'_i) be mapped to the nodes k, j respectively in $PARL(O_i, PR)$. A policy update operation is a policy restriction operation if $lub(k, j) \neq j$.

Note that, when a policy's priority is decreased, it may no longer be enforced if this is not the highest priority policy in effect. Thus, the privilege that a subject enjoys by virtue of some policy may be diminished when the policy's priority is decreased. Hence, decreasing the priority of a policy corresponds to policy restriction.

5.1 Concurrency Control Mechanism

When multiple policies are specified over a subject and an object, updating one policy has certain side effects. The modification of a policy not only impacts transactions executing due to that policy but also transactions that are executing by the privileges given by other policies specified over the same subject and object. An example will help illustrate this point. Suppose a transaction T_p is executing by virtue of policy $P_i =< S, O, 010, Low >$. Consider another policy $P_j =< S, O, 001, Low >$ where the priority of P_j is changed from Low to $High$. In such a situation since P_i is no longer a deployable policy, T_p must be aborted.

In short, a policy restriction operation causes transactions executing by virtue of that policy to be aborted. A policy relaxation does not require transactions executing by virtue of it to be aborted. However, if the policy relaxation operation changes the priority of the policy to be greater than the priorities of the deployed policies defined over the same subject and object, then the transactions executing by virtue of these other policies must be aborted.

The above situation is complex. Whenever the priority of a deployed policy is increased, the transactions executing by virtue of the unchanged policies on the same subject and object get affected. Our previous locking mechanisms will no longer work. In our previous mechanisms, when a policy was relaxed no transactions executing by virtue of this policy was affected. Now a policy relaxation also affects transactions executing by virtue of other policies.

For our mechanism, we use the same kinds of locks as specified in Section 4.1. The locking rules are given in Table 2. The algorithm which we use to update policy is formalized below.

Algorithm 1 Update policy

Input: (i) Policy $P_i =< S_i, O_i, R_i, pr_i >$ that is to be updated, (ii) R_i' – the new rights of P_i, (iii) pr_i' – the new priority of P_i, (iv) **P** – the set of policies defined over S_i and O_i, and (v) **D** – the set of deployable policies defined over S_i and O_i

Output: (i) Updated policy P_i', (ii) **P′** – the new set of policies defined over S_i and O_i, and (iii) **D′** – the new set of deployable policies

```
Procedure UpdatePolicy($P_i$, $R_i'$, $pr_i'$, P, D)
begin
    D′ = D
    if $lub\{(pr_i, R_i), (pr_i', R_i')\} = (pr_i', R_i')$    /* policy relaxation */
    begin
        acquire $WXL(P_i)$
        $P_i' =< S_i, O_i, R_i', pr_i' >$    /* modify $P_i$ */
        release $WXL(P_i)$
        P′ = P ∪ $\{P_i'\} - \{P_i\}$
        if $P_d \in \mathbf{D} \wedge P_d \neq P_i$
            if $pr'_i > pr_d$      /* new priority greater than that of deployed policies */
                for each $P_j \in \mathbf{D} \wedge P_j \neq P_i$
                    acquire $WSL(P_j)$      /* will abort transactions executing by virtue of $P_j$*/
                    release $WSL(P_j)$
                $\mathbf{D}' = \{P'_i\}$
            else if $pr'_i = pr_d \wedge pr_i < pr_d$      /* new priority equal to that of deployed policies */
                $\mathbf{D}' = \mathbf{D} \cup \{P'_i\}$
    end
    else      /* policy restriction */
    begin
        acquire $WSL(P_i)$
        $P'_i =< S_i, O_i, R'_i, pr'_i >$      /* modify $P_i$ */
        release $WSL(P_i)$
        $\mathbf{P}' = \mathbf{P} \cup \{P'_i\} - \{P_i\}$
        if $P_d \in \mathbf{D} \wedge P_d \neq P_i \wedge pr'_i < pr_d \wedge pr_i = pr_d$      /* new priority lesser than that of deployed policies
            $\mathbf{D}' = \mathbf{D} - \{P_i\}$
        if $\mathbf{D} = \{P_i\}$      /* if $P_i$ was the only deployable policy */
            /* Calculate the new set of deployable policies */
            $\mathbf{D}' = \{\}$
            $pr_{max}$ = max_priority of policies in $\mathbf{P}'$
            for each $P_m \in \mathbf{P}$
                if $pr_m = pr_{max}$
                    $\mathbf{D}' = \mathbf{D}' \cup P_m$
    end
end
```

The algorithm begins with initializing the new set of deployable policies $\mathbf{D}'$ with the existing set of deployable policies $\mathbf{D}$. This is the default case. Some operations may require a change in the set of deployable policies. We will explain these as and when the cases arise. We then check whether the update of policy P_i is a relaxation or not. Based on this test, we divide the main body of the algorithm into two parts: one for policy relaxation and the other for policy restriction.

For policy relaxation, we obtain the write relax lock on the policy P_i and update it. The lock is released when the policy update transaction terminates. The new set of policies defined over the subject S_i and O_i, denoted by **P′**, is computed by removing the old policy P_i and including the new one P'_i. Policy relaxation may increase the priority of P_i. In such cases, we need to decide the fate of existing deployed policies. The actions to be taken depend on the priority pr_i of the original policy P_i as well as the priority pr'_i of the modified policy P'_i. The first case is when the priority pr_i is increased to a value that is greater than that of the existing deployable policies. In this case, the new set of deployable policies **D′** consists of P_i only. The original set of deployable policies **D** is no longer deployable. Transactions executing on behalf of the policies in **D** need to be aborted. To achieve this, we obtain a write restrict lock on all the policies in **D**. This will cause transactions holding deploy locks on these policies to be aborted. The second case is when the old priority pr_i of P_i is less than that of the deployable policies and the new priority pr'_i is equal to that of the deployable policies. In this case P'_i needs to be included in the new set of deployable policies **D′**.

Next, we consider the case when the operation is a policy restriction. For policy restriction, a write restrict lock is acquired on P_i. This causes transactions executing by virtue of P_i to abort. After the transaction terminates, this lock is released. The new set of policies **P′** defined over subject S_i and object O_i is computed by removing P_i from the set **P** and including P'_i. Here again, the change of priority may cause a change in the set of deployable policies. The first case is when the original priority pr_i is the same as that of the existing deployable policies and the new one pr'_i is less than that. In this case P_i is removed from the new set of deployable policies **D′**. There may also be a case when P_i is the only deployable policy. In such a case reducing the priority of P_i requires re-evaluating the new set of deployable policies **D′**. To find this set **D′**, we first compute pr_{max} which is the maximum priority of the policies in the set **P′** and then insert in the set **D′** the policies in **P′** that have the same priority as pr_{max}.

We now prove some desirable properties of our mechanism.

Theorem 8 Any history H generated by this mechanism is serializable.

Proof 8 This can be proved in a manner that is similar to the proof of Theorem 4.

Theorem 9 Any history H generated by this mechanism is policy-compliant.

Proof 9 Assume that the history H is not policy-compliant. This means that one or more transactions in the history H is not policy-compliant. Suppose transaction T_i is one such transaction that is not policy-compliant. Without loss of generality, assume that transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i satisfies condition 1 of Definition 19, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. One possibility is that while T_i was updating O_r, the policy P_m got deleted or modified such that T_i no longer has the privilege to update O_r. But this scenario cannot occur according to our algorithm. Any transaction T_j restricting the policy P_m has to obtain a write restrict lock (WSL) on P_m. Before the write restrict lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (because T_j restricts the policy). Thus, the above scenario of T_i updating O_r without any policy authorizing T_i to do so, does not arise in our case. The other possibility is that while T_i was executing, the policy P_m was not changed, but it became an undeployable policy because some other deployable policy P_n increased its priority. This is also not possible by our algorithm because whenever P_n's priority is increased to be higher than P_m's, the policy update algorithm acquires a write-restrict lock on P_m which causes transactions deploying P_m to be aborted. Thus, the situation of T_i updating O_r without the authorization of any policy object, does not arise in our case. In other words, T_i is policy-compliant. We can make a similar case for all other transactions. Thus, H is policy-compliant.

6 Related Work

A large body of work appears in the area of access control. Researchers have proposed access control models [5, 7, 37, 38], policy specification languages [4, 9, 13, 17, 19, 20, 30, 35], and techniques for identifying conflicts in policies [3, 8, 16, 23, 28, 42, 43]. Policy updates have received relatively little attention. Ray and Xin [33] proposed algorithms for concurrent and real-time update of access control policies. The algorithms proposed consider only simple kinds of authorization policies where the only one access control policy is specified for any given subject and an object and only the rights associated with the policy is subject to change. In a subsequent work [34] the authors show how such algorithms can be implemented. The use of semantic knowledge for real-time update of access control policies was proposed by Ray [31]. In this work the author considered only simple kinds of authorization policies $P_i =< SS_i, TS_i, RS_i >$ where SS_i, TS_i, RS_i represents the set of subjects, the set of objects, and the set of access rights respectively. A policy can be changed by performing a series of set union or set difference operations. Depending on what kinds of operations were performed, the policy update is classified as policy restriction or relaxation. The author proposed algorithms that uses this knowledge to generate conflict serializable histories. The author also introduced the idea of how the application context can be used to check whether a policy update transaction interferes with a transaction that deploys the policy. However, a formal treatment of this idea is missing from the work. In a subsequent work [32], a more formal treatment of the semantic-based approach is proposed. This work also shows how transactions can be decomposed to minimize the effects of useless aborts. Semantic-based transaction processing comes at a cost – the cost is in terms of analyses needed to ensure correct behavior. Consequently, the current work uses a syntactic approach for policy update. None of the earlier works consider the case where there are multiple policies specified on a given subject and an object, priorities are specified on policies and the priorities themselves are subject to change.

Some work has been done in identifying interesting adaptive policies and formalization of these policies [14, 40]. A separate work [39] 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.

Automated management of security policies for large scale enterprise has been proposed by Damianou [12]. 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.

Lymberopoulus et al. [24] proposed an adaptive policy-based framework for specifying policies for management of network services. The framework is expected to support automated policy deployment and flexible event triggers to permit dynamic policy configuration. The authors focused on the dynamic policy adaptation with changes within the managed environment. The policy adaptation includes dynamically changing policy parameters and reconfiguring the policy objects. The authors then described how the policy-based management framework could be used to provide dynamic management of services in Differentiated Services networks. The system is claimed to allow flexible and adaptive management where the policies define the adaptation choices without recoding or even shutting down the system. The proposed adaptive policy-based framework is based on PONDER which is an object-oriented, declarative language for specifying management and security policies. Their approach is focused on the use of obligation policies, instead of the authorization policies. The obligation policies specify the actions that must be performed when certain events occur, and provide the ability to respond to changing circumstance. Obligation policies are event-triggered condition action rules, composing of subjects, target objects, events, and actions. The proposed policy adaptation is capable to modify the network behavior in 3 ways: (1) Adaptation by dynamically changing the parameter of a QoS policy to specify new attribute values for the run-time configuration of managed objects; (2) Adaptation by selecting and enabling/disabling a policy from a set of pre-defined QoS policy at run-time, and the parameters of the selected network QoS policy are set at run-time; (3) Adaptation by learning the most suitable policy configuration strategies from the system behavior. However, the authors only focused on the first two categories of policy adaptation, while the third mechanism is not addressed. The authors also described the enforcement architecture for the adaptive policy system, composing of Service PMA, PolicyRequest, Policy Services, Event Services, and Network level PMAs. Interaction and message communications among these components are also addressed. Then, the authors present a usage scenario to show how the adaptive policy framework can apply in a Differentiated Services network environment. There are obligation policies defined to handle service performance degradation events and policies to support changes in routing or link failures, as well as policies to reflect changes in application or user requirements. In these papers, the authors focused on the policy adaptation with obligation policies. In many applications, authorization policies are used, and the issues with real-time update of authorization policies are not resolved as in previous work. Our policy-updating paper will focus on the issues arise due to updates of authorization policies in real-time.

Some work has been done in the area of dynamic and adaptive workflow systems which is related to this work. Researchers [10, 21, 26, 27, 29, 36, 45] have motivated the need for workflow systems that have the ability to adapt to the changing environments. Kammer et al. [21] have identified different kinds of exception handling capabilities for adaptive workflows. They also describe how some of these capabilities have been incorporated in the Endeavors workflow support system. Van der Aalst et al.[45] have mentioned that when a workflow is changed there are three ways to manage workflow instances that were active during the change: (a) restart, (b) proceed, and (c) transfer. Restart causes abort of all the tasks that were being executed under the original model. Proceed requires continuation of the workflow instance under the original model. Transfer requires the instance to be changed dynamically such that it complies with the new model. Sadiq [36] recognizes that changes to workflow model or workflow instances can lead to serious inconsistencies and errors. He points out that the problem becomes more critical when active instances are involved while the workflow model is being changed. The authors approach this problem by first formalizing how workflow can be modified, and then checking whether the instance initiated under the original workflow complies with the updated workflow. The instances generated from the old workflow may continue to follow the old model or they may be provided with a revised schedule to comply with the new model. The generation of revised schedule may need manual intervention. Mathews [27] has also addressed the problem of changing workflow policies while active workflow instances are in progress. He agrees that aborting active workflow instances prior to changing a workflow policies is inefficient and ineffective. To deal with this situation, he proposes SWAP, an agent-based architecture, that deals with changes in B2B workflow policies. In this architecture, the different agents keep track of the active workflow instances and their state of execution and are responsible for transitioning the workflow instance so that it complies with the new policy. Muller and Rahm[29] suggested that dynamic workflow modification is needed for many applications, such as, medical domain. In such domains the large number of exceptions cannot be handled by the domain except. They have developed a rule-based approach for detecting semantic exceptions that occur during instance execution. The control-flow of other instances affected by this exception needs to be modified dynamically. For this purpose, the authors have proposed two algorithms that carry out the dynamic change of control-flow. Although the problems have a similar flavor, the solutions for the problem of policy updates is very different. We are interested in preventing security breaches for dynamic policy updates without requiring any manual intervention. For making a workflow compliant with the new policy some amount manual task is involved.

Concurrency control in database systems is a well researched topic. Some of the important pioneering works have been described by Bernstein et al. [6]. Thomasian [44] 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, 15, 18, 22, 25, 41].

7 Conclusion and Future Work

Dynamic environments pose new challenge to access control. In such situations, the entities change, the configurations change and the operational modes change – all of these require real-time update of access control policies. In this paper, we address the problem of how policies can be updated in real-time and how the modified policies can be immediately enforced. We discuss our work in the context of a database system. We start with the simple case where a single policy is defined over a subject and an object and show how updates to such policy can be performed in real-time. In real-world, multiple policies may be specified over a given subject and an object. Priorities may be specified on these policies. We propose a mechanism that addresses the update of such policies where the priorities are also subject to change. Our mechanisms generate serializable and policy-compliant histories.

In future we plan to extend our approach to handle more complex kinds of authorization policies, such as, support for negative authorization policies and obligation policies, and incorporating conditions in authorization policies. Specifically, we plan to investigate how policies specified in the PONDER specification language [11] can be updated. Another important work that needs to be done is analyzing the policy update transactions. If the policy update transaction erroneously gives too much or too little privilege to any subject, then the confidentiality, integrity, and availability of the application may be affected. Checking whether the policy update transaction gives adequate privilege or not can be done by statically analyzing the application together with all the policy update transactions. In future we plan to investigate how such analysis can be done.

Acknowledgement

This work was partially supported by AFOSR under award no. FA9550-04-1-0102 and by NSF under grant no. IIS 0242258.

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] A. Bandara, E. Lupu, and A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. In *Proceedings 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, Lake Como, Italy, June 2003.

[4] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[5] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

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

[7] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[8] M. Charalambides, P. Flegkas, G. Pavlou, A. Bandara, E. Lupu, A. Russo, N. Dulay, M. Sloman, and J. Rubio-Loyola. Policy Conflict Analysis for Quality of Service Management. In *Proceedings 6th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2005)*, Stockholm, Sweden, June 2005.

[9] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[10] E. C. Cheng. OMM: An Organization Modeling and Management System. In *Proceedings of Towards Adaptive Workflow Systems Workshop co-located with CSCW 98*, Seattle, WA, November 1998.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[16] N. Griffeth and H. Velthuijsen. Reasoning About Goals to Resolve Conflicts. In *Proceedings of the International Conference on Intelligent Cooperative Information Systems*, pages 197–204, Los Alamitos, California, 1993.

[17] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.

[18] 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.

[19] M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[20] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[21] P. Kammer, G. Bolcer, R. Taylor, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work*, 9(3-4):269–292, 2000.

[22] H. F. Korth and G. Speegle. Formal Aspects of Concurrency Control in Long-duration Transaction Systems Using the NT/PV Model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.

[23] E. Lupu and M. Sloman. Conflict Analysis for Management Policies. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management*, pages 430–443, San Diego, California, May 1997. Chapman & Hall.

[24] L. Lymberopoulos, E. Lupu, and M. Sloman. An Adaptive Policy-Based Framework for Network Services Management. *Journal of Network and System Management*, 11(3):277–303, September 2003.

[25] Nancy A. Lynch. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[26] D. Manolescu and R. Johnson. Dynamic Object Model and Adaptive Workflow. In *Proceedings of Metadata and Active Object-Model Pattern Mining Workshop co-located with OOPSLA 99*, Denver, CO, November 1999.

[27] M. G. Mathews. Supporting Dynamic Change in B2B Coordination. Technical report, The MITRE Corporation, June 2001.

[28] N. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 89–97, Annapolis, MD, May 1996.

[29] R. Muller and E. Rahm. Rule-Based Dynamic Modification of Workflows in a Medical Domain. In A.P. Buchmann, editor, *Proceedings of BTW*, pages 429–448, Freiburg in Breisgau, Germany, March 1999. Springer.

[30] R. Ortalo. A Flexible Method for Information Systems Security Policy Specification. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[31] I. Ray. Real Time Updation of Security Policies. *Data and Knowledge Engineering*, 49(3):287–309, June 2004.

[32] I. Ray. Applying Semantic Knowledge to Real-Time Update of Access Control Policies. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):844–858, June 2005.

[33] 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*, pages 330–339, Prague, Czech Republic, September 2003.

[34] I. Ray and T. Xin. Implementing Real-Time Update of Access Control Policies. In *Proceedings of the 18th IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 65–80, Sitges, Spain, July 2004.

[35] C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[36] S. Sadiq. Workflows in Dynamic Environment – Can they be managed? In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, Wollongong, Australia, March 1999.

[37] P. Samarati and S. Vimercati. Access Control: Policies, Models and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196, September 2000.

[38] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[39] 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.

[40] 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.

[41] 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.

[42] E. Sibley. Experiments in Organizational Policy Representation: Results to Date. In *Proceedings of the IEEE International Conference on Systems Man and Cybernetics*, pages 337–342, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[43] E. Sibley, J. Michael, and R. Wexelblat. Use of an Experimental Policy Workbench: Description and Preliminary Results. In C. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 47–76. Elsevier Science Publishers, 1992.

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

[45] W.M.P. van der Aalst, T. Basten, H.M.W. Verbeek, P.A.C. Verkoulen, and M. Voorhoeve. Adaptive Workflow. In J.B.L. Filipe, editor, *Enterprise Information Systems*. Kluwer Academic Publishers, 1999.