2004－CSEC－25　(3)
2004／5／21

Locking Protocol for Information Flow Control

Ryung Chon, Tomoya Enokido, and Makoto Takizawa
Dept. of Computers and Systems Engineering
Tokyo Denki University
E-mail {den, eno, taki}@takilab.k.dendai.ac.jp

Abstract

This paper discusses a novel locking protocol to prevent illegal information flow among objects in a role-based access control (RBAC) model. In this paper, we newly define a conflicting relation "a role R_1 conflicts with another role R_2" to show that illegal info rmation flow may occur if a transaction with R_1 is performed befor e another transaction with R_2. Here, we newly introduce a role lock on an object to abort a transaction with R_1 if another transaction with R_2 had been already performed on the object. Role locks are not released even if transactions issuing the ro le locks commit. A role lock on an object can be released if information brought into the object got obsolete. We discuss how to release obsolete role locks.

オブジェクト間で起こり得る不正な情報流を防止する役割ロック

田 隆　榎戸 智也　滝沢 誠
東京電機大学理工学部情報システム工学科
E-mail {den, eno, taki}@takilab.k.dendai.ac.jp

役割アクセス制御 (RBAC) モデルでは、オブジェクト間で不正な情報流が生じ得る。本論文で不正な情報流を防止するために、新たなロックプロトコルを論ずる。役割 R_1 のトランザクション T_1 の次に役割 R_2 の他のトランザクション T_2 が実行されると、不正な情報流が生じるとき、R_2 は R_1 に競合すると定義する。各トランザクションは、この役割をモードとするロックをかける。役割競合するロックがかけられているときは、トランザクションはアボートする。この役割ロックは、トランザクションがコミットされたとしても、解放されない。不要になった役割ロックを解放する方法についても論じる。

1 Introduction

In access control models, $confinement\ problem$ [6] occurs. That is, illegal information flow might occur even if only authorized requests are performed on each object. In order to resolve the confinement problem, mandatory access control (MAC) [1] and lattice-based access control (LBAC) [3, 9] models are discussed for traditional $read$ and $write$ methods on simple objects like files and tables. Information flow control in object-based systems is discussed in the paper [8]. However, information flow control for only $read$ and $write$ methods is discussed. An object-based system [7] is composed of objects which are distributed in a network. Objects support more abstract methods than $read$ and $write$ ones. In the paper we discuss access control in object-based systems.

In a role-based access control model (RBAC) [10, 11], a $role$ shows a job function in an enterprise. A role is specified in a set of access rights [4, 10]. Each access right is a pair $<o, t>$ which means that an object o is allowed to be manipulated by a method t. If a subject is granted a role including an access right $<o, t>$, the subject can issue a request t to an object o. A subject can be granted multiple roles. A subject initiates a transaction with role granted to manipulate objects. A transaction is modeled to be an atomic sequence of methods issued to objects.

Suppose a role includes a pair of access rights $<f, read>$ and $<g, write>$ and a transaction first issues a method $read$ to a file object f and then $write$ to a file object g. Here, data in a file object f is derived through the method $read$ and then the data is brought into the file object g through the method $write$. Suppose a role R_2 includes another access right $<g, read>$ where data in the object g is derived through the method $read$. Here, a transaction T_2 with R_2 can obtain data in the object f from the object g even if T_2 is not allowed to manipulate f. If the role R_2 includes an access right $<f, read>$ where data in f can be derived through a method $read$, information flow from f to the transaction T_2 is not illegal. Otherwise, the information flow is illegal. Thus, we discuss how information flow to occur by performing transactions with roles. In addition, information flow depends on in what order methods are performed in a transaction.

In order to check if illegal information flow might occur on performing a method of a transaction, we newly introduce an asymmetric $conflicting$ relation among roles. A role R_1 $conflicts\ with$ another role R_2 iff illegal information flow might occur by performing a transaction with a role R_1 before another transaction with a role R_2. We newly introduce $role\ locks$ on objects. A transaction T with a role R locks an object o with a role lock of mode R before manip-

ulating the object o. If no role lock conflicting with role R is held on the object o, the transaction T can manipulate the object o. Otherwise, the transaction T is aborted because illegal information flow with the object o might occur if T manipulates the object o. Here, all role locks held by the transaction T are released. However, even if a transaction commits, role locks held by the transaction are not released differently from traditional locking protocols [2]. Thus, role locks are monotonically accumulated on each object each time a transaction commits. Suppose a transaction with a role R derives data from an object o_1. In the meantime, another transaction updates the object o_1. Here, the role lock R on another object o_2 is $obsolete$ since the value brought into the object o_2 is obsolete, i.e. different from the current value in the object o_1. We discuss how to release obsolete role locks on objects.

In section 2, we discuss a conflicting relation among roles. In section 3, we discuss how to lock and release objects in roles. In section 4, we discuss implementation of role locks.

2 Conflicting Relation on Roles

2.1 Roles on objects

An object-based system is composed of objects. Each object is an encapsulation of data and methods for manipulating the data [5]. Methods are more abstract than primitive methods like $read$ and $write$ on a simple object like table since methods are procedures realized by primitive and other methods. For example, a $counter$ object a supports methods $initialize$($init$), $increment$(inc), and $decrement$(dec). The $counter$ object a is manipulated only through these methods. Objects are distributed in servers which are interconnected with high-speed networks.

A method is characterized in terms of two types $Input$ and $Output$ with respect to whether data is derived from an object or brought into an object. A method t is an $Input$ type if data in an object o is derived through the method t. A method t is an $output$ type if data is brought into an object o through the method t. Here, $Input$(t) and $Output$(t) indicate propositions that a method t is an $Input$ and $Output$ type, respectively. For example, $Input$(inc), $Input$(dec), and $Output$($check$) hold in a $counter$ object.

An access right (or permission) is specified in a pair $<o, t>$ of an object o and a method t supported by the object o. Only a subject s who is granted an access right $<o, t>$ is allowed to manipulate an object o only through a method t. A $role$ R is a collection of access rights. Each role R shows a job function in an enterprise. A subject performing a job function is granted a role R showing the job function in the enterprise. If a subject performs multiple job functions, the subject is granted multiple roles. A subject initiates a transactions to objects to do some work. A $transaction$ is a unit of work which is a sequence of access requests. A $transaction$ is associated with one of roles which the subject is granted. A $transaction$ T associated with a role R is allowed to issue a method t on an object o if $<o, t> \in R$. Otherwise, the access request $<o, t>$ is rejected, i.e. the transaction T is aborted. An access request t on an object o is also written as a pair $<o, t>$.

Information in an object o is $derived$ in a role R ($o \Rightarrow R$) if and only if (iff) $<o, t> \in R$ and $Input$(t). Information in a role R is $brought$ into an object o ($R \Rightarrow o$) iff $<o, t> \in R$ and $Output$(t). Suppose a role R includes access rights $<a, inc>$ and $<a, check>$ on a $counter$ object a. Here, $a \Rightarrow R$ since some transaction with role R may issue a method inc which brings data into an object a, i.e. $Input$(inc). $R \Rightarrow a$ since data in the object a is derived by $check$, i.e. $Output$($check$) [Figure 1].

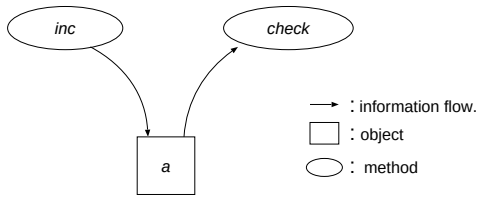

Figure 1. Information flow.

Information in an object o_1 $flow\ into$ an object o_2 in a role R ($o_1 \Rightarrow_R o_2$) iff $o_1 \Rightarrow R$ and $R \Rightarrow o_2$. Let a and b be a pair of $counter$ objects. Suppose there are a pair of roles R_1 = {$<a, check>$, $<b, inc>$, ...} and R_2 = {$<a, dec>$, $<b, check>$, ...}. If a transaction T with R_1 issues a method $check$ to the object a ($a \Rightarrow R_1$) and then inc to b ($R_1 \Rightarrow b$), $counter$ information in the object a might be brought into the other object b, i.e. $a \Rightarrow_{R_1} b$. $b \Rightarrow_{R_2} a$ since $b \Rightarrow R_2$ in $check$ and $R_2 \Rightarrow a$ in dec. Information in an object o_1 $flow\ into$ an object o_2 ($o_1 \Rightarrow o_2$) iff $o_1 \Rightarrow_R o_2$ for some role R.

2.2 Conflicting relation on roles

Suppose counter object a supports a method $check$ and another counter object b supports a pair of methods $check$ and inc where $Output$($check$), $Output$($check$), and $Input$(inc). Let T_1 and T_2 be a pair of transactions with roles R_1 = {$<a, check>$, $<b, inc>$, ...} and R_2 = {$<b, check>$, ...}, respectively. Suppose that the transaction T_1 issues a pair

of access requests <a, $check$> and <b, inc>. Here, information in a $counter$ object a flow into another $counter$ object b ($a \Rightarrow b$). Then, the transaction T_2 issues an access request <b, $check$>. If the role R_2 includes an access right <a, $check$>, the transaction T_2 is allowed to derive data from the object o_1. Otherwise, T_1 cannot issue the access request <b, $check$> because T_1 might obtain data in the counter object a from another object b. That is, illegal information flow might occur. This is shown in a directed $object$-$role$ graph [Figure 2]. Here, there are two types of nodes, $role$ and $object$ nodes. Directed edges "$o \rightarrow R$" from an object node o to a role node R and "$R \rightarrow o$" mean $o \Rightarrow R$ and $R \Rightarrow o$, respectively. $o \rightarrow^* R$, and $R \rightarrow^* o$, $\alpha \rightarrow^* \beta$ and $R \rightarrow^* R'$ iff $o \Rightarrow^* R$, $R \Rightarrow^* o$, respectively. $\alpha \rightarrow^* \beta$ shows a path from a node α to β in an object-role graph. A role R is referred to as $Output$ and $Input$ types if $R \rightarrow o$ and $o \rightarrow R$ for some object o, respectively. In Figure 2, the roles R_1 and R_2 are $Output$ types and R_2 and R_3 are $Input$ types.

[Definition] Information in a role R_1 $flow\ into$ a role R_2 ($R_1 \Rightarrow R_2$) iff $R_1 \Rightarrow o \Rightarrow R_2$ for some object o.□

Information in a role R_1 $transitively\ flow$ into another role R_2 ($R_1 \Rightarrow^* R_2$) iff $R_1 \Rightarrow R_2$ or $R_1 \Rightarrow R_3 \Rightarrow^* R_2$ for some role R_3. In Figure 2 a role R_3 is {<b, $check$>. Here, $R_1 \Rightarrow^* R_3$ since $R_1 \Rightarrow R_2$ and $R_2 \Rightarrow R_3$. Data written to the counter object a in the method inc by a transaction with a role R_1 might be derived in $check$ by another transaction with a role R_3.

A following property holds for the subsumption relation:

[Property] $R_3 \Rightarrow R_2$ if $R_1 \Rightarrow R_2$ and $R_1 \subseteq R_3$.□

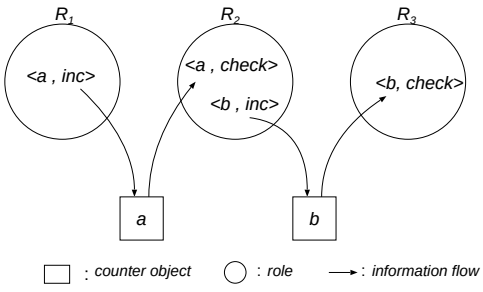

Figure 2. Object-role graph ($R_1 \Rightarrow R_2 \Rightarrow R_3$).

Let **R** be a set of roles in a system. A role R_1 is referred to as $connected$ to another role R_2 in **R** ($R_1 \mapsto R_2$) iff $R_1 \Rightarrow R_2$ or $R_1 \mapsto R_3 \mapsto R_2$ for some role R_3 in **R**. $R_1 \mapsto R_2$ means that some data written into objects by a transaction with a role R_1 might be derived by another transaction with a role R_2.

[Definition] A role R_1 $conflicts$ with a role R_2 ($R_1 \triangleright R_2$) iff $o \Rightarrow R_1 \Rightarrow R_2$ and $o \not\Rightarrow R_2$ for some object o.□

A role R_1 $transitively\ conflicts$ with a role R_2 ($R_1 \triangleright^* R_2$) iff $o \Rightarrow R_1 \Rightarrow^* R_2$ and $o \not\Rightarrow R_2$ for some object o. A role R_1 is $compatible$ with a role R_2 ($R_1 \square R_2$) iff $R_1 \triangleright R_2$ does not hold. Neither the conflicting relation $\triangleright$ nor compatible relation $\square$ is symmetric. That is, $R_2 \triangleright R_1$ and $R_2 \square R_1$ may not hold even if $R_1 \triangleright R_2$ and $R_1 \square R_2$, respectively. In addition, the conflicting relation $\triangleright$ and compatible relation $\square$ are neither transitive nor reflexive. Suppose a role R_1 conflicts with another role R_2 ($R_1 \triangleright R_2$). Let T_1 and T_2 be transactions with roles R_1 and R_2, respectively. Suppose the transaction T_1 derives some data in an object o through a method t_1, i.e. $Input(t_1)$ and then writes the data into another object o_1 through a method t_2, i.e. $Output(t_2)$ ($o \Rightarrow R_1 \Rightarrow o_1$). The other transaction T_2 derives data from the object o_1 ($o_1 \Rightarrow R_2$), but is not allowed to derive data from the object o ($o \not\Rightarrow R_2$) [Figure 3]. Thus, if T_2 is performed after T_1, illegal information flow might occur. The transaction T_2 might get data in the object o by manipulating the object o_1 even if T_2 is not allowed to derive data from o. On the other hand, if T_2 is performed before T_1, no illegal information flow occur. Thus, R_2 is compatible with R_1 ($R_2 \square R_1$) even if $R_1 \triangleright R_2$. This example shows it depends on the computation order of transactions whether or not illegal information flow might occur.

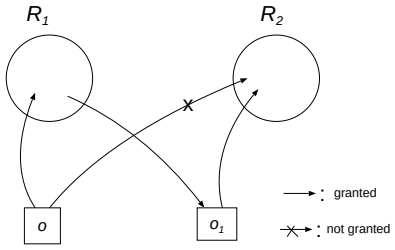

Figure 3. R_1 conflicts with R_2 ($R_1 \triangleright R_2$).

[Properties] If a role R_1 conflicts with another role R_2 ($R_1 \triangleright R_2$),

1. $R_3 \triangleright R_2$ if $R_1 \subseteq R_3$.
2. $R_1 \triangleright R_3$ if $R_3 \subseteq R_2$ and $R_1 \Rightarrow R_3$.□

Let us consider a pair counter objects a and b. Let R_1 and R_2 be a pair of roles {<a, $check$>, <b, inc>} and {<b, $check$>, <b, dec>}, respectively. R_1 conflicts with R_2 ($R_1 \triangleright R_2$) as discussed here. Let R_3 be another role {<a, $check$>, <b, inc>, <b, $check$>}. Here, $R_3 \triangleright R_2$ since $R_1 \subseteq R_3$. Let R_4 be a role {<b, $check$>}. $R_1 \triangleright R_4$ since $R_1 \Rightarrow R_4$.

The conflicting relation $\triangleright \subseteq \mathbf{R}^2$ is obtained by searching an object-role graph for R:

[Conflicting relation $\triangleright$]

- For every pair of roles R_1 and R_2 in **R**, first assume that $R_1 \square R_2$ and $R_2 \square R_1$ held.
- For every $Output$ role R in **R**,

for every object o such that $o \rightarrow R$,
for every $Input$ role R' such that $R \rightarrow^* R'$ and $R \square R'$, unless $o \rightarrow R'$, $R \square R'$ is changed with $R \triangleright R'$.□

[Definition] A role R is $safe$ iff $R \square R'$ for every role R' in **R**.□

A transaction is $safe$ iff the transaction is associated with a $safe$ role. No illegal information flow occur if only safe transactions are performed. A pair of $safe$ transactions can be performed in any order. A system is $safe$ iff every role is $safe$. Safe systems are restricted and most systems are unsafe. In this paper, we discuss how to prevent illegal information flow even in unsafe systems.

3 Role Locking Protocol

3.1 Role locks

Transactions have to manipulate objects so that no illegal information flow occurs in a role-based access control (RBAC) model. Suppose a transaction T is associated with a role R. The transaction T manipulates a $role\ set$ variable $T.role$. Each object o also has a $role\ set$ variable $o.role$. Initially, $T.role := \phi$ and $o.role := \phi$. Suppose a transaction T issues an access request $<o, t>$ for an object o and a method t. The $role\ set$ variables $T.role$ and $o.role$ are manipulated for the access request $<o, t>$ as follows :

[Manipulation of access request $<o, t>$]

1. If $Input(t)$, i.e. a transaction T derives data from an object o through a method t, $T.role := T.role \cup o.role$.
2. If $Output(t)$, i.e. T brings data into an object o through a method t, $o.role := o.role \cup T.role$ if $R_1 \square R$ (R_1 is compatible with R) for every role R_1 in $o.role$. Otherwise, T is aborted.
3. The method t is performed on the object o.□

Suppose that a transaction T with a role R issues an access request $<o, t>$ where $Output(t)$, i.e. T would like to derive data from an object o. If every role R_1 in $o.role$ is compatible with the role R ($R_1 \square R$), the access request t is performed on the object o. Then, the role lock mode R is added to the role set variable $o.role$ of the object o, i.e. $o.role := o.role \cup \{R\}$. If some role R_1 in $o.role$ conflicts with the role R ($R_1 \triangleright R_2$) in the condition 2, illegal information flow to the object o might occur by performing the method t. Hence, the transaction T is aborted.

[Example] Suppose there are a pair of $counter$ objects a and b. Let R_1 and R_2 be a pair of roles $\{<a, check>, <b, inc>\}$ and $\{<b, check>\}$, respectively. Here, R_1 conflicts with R_2 ($R_1 \triangleright R_2$) as shown in Figure 4. Suppose a pair of transactions T_1 and T_2 are associated with roles R_1 and R_2, respectively. First, the transaction T_1 first issues an access request $<a, check>$ and then an access request $<b, inc>$. Here, $b.role = \{R_1\}$. Next, suppose the other transaction T_2 is performed while issuing an access request $<b, check>$. Since $a.role = \phi$ and $Output(check)$, i.e. data is derived by $check$, roles in $b.role$ are compared with the role R_2 of T_2. Since $R_1 \in b.role$ and R_1 conflicts with R_2 ($R_1 \triangleright R_2$), the access request $<b, check>$ is rejected, i.e. T_2 is aborted.

On the other hand, suppose the transaction T_2 is first performed on the object b before T_1. Here, $a.role = \phi$ and $b.role = \phi$. Then, the transaction T_1 is performed. $b.role = \phi$ when T_2 issues an access request $<b, deposit>$. Since there is no role lock in $b.role$ which conflicts with R_2, T_2 can issue inc on the $counter$ object b.□

This example shows that illegal information flow might not occur even if transactions with conflicting roles are performed. Hence, each time a transaction T issues a method t on an object o, it is checked by using role locks if illegal information flow might occur on performing the method t in our approach.

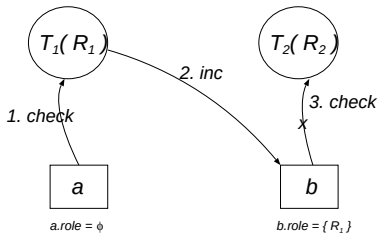

Figure 4. Information flow.

3.2 Obsolete roles locks

A role set $o.role$ of an object o is monotonically growing since roles are just added to the role set $o.role$ each time an $Output$ method is performed on the object o. Here, suppose a transaction T_1 with a role R_1 derives data from an object o_1 through a method t_1 and then writes the data into another object o_2. Here, $o_1 \Rightarrow R \Rightarrow o_2$. Next, suppose a transaction T_2 with a role R_2 derives data from the object o_2 and then writes the data into an object o_3. Hence, a pair of role locks R_1 and R_2 are held on the object o_3, i.e. $o_3.role = [R_1, R_2]$. Suppose another transaction T_3 updates the object o_1 [Figure 5]. The data derived from the object o_1 by the transaction T_1 is obsolete now. Hence, the role lock R_1 can be released on the object o_3.

[Reduction rule 1] If a role lock R on an object o is obsolete, R is released, i.e. removed from $o.role$.□

[Definition] A role lock R held on an object is $obsolete$ iff an object o_1 is updated where $o_1 \Rightarrow R \Rightarrow^* o$.□

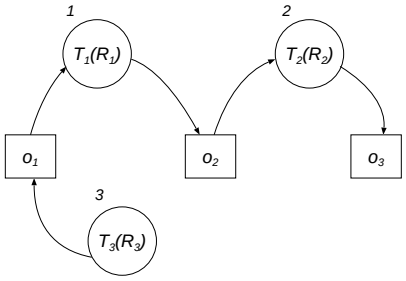

Figure 5. Information flow.

From properties of the conflicting relation, following roles in the role set $o.role$ can be removed :
[Reduction rule 2] For every pair of roles R_1 and R_2 in $o.role$, R_1 is released, i.e. removed from $o.role$ if $R_1 \subseteq R_2$.□

4 Implementation

4.1 One-server model

We discuss how to lock objects in role modes and release obsolete role locks. First, suppose that no lock is held on every object o. A transaction T with a role R issues an access request $<o, t>$ at time τ. An object o is locked in a mode R. Here, the role lock R on the object o is uniquely identified in a tuple $<o, R, \tau>$. Even if another transaction T' with a role R issues a same access request $<o, t>$ at time τ, the lock is identified in a tuple $<o, R, \tau'>$ where $\tau \neq \tau'$. In addition, each role lock $<o, R, \tau>$ is assigned with a unique identifier id. A tuple $<id, o, R, \tau>$ is stored in a role set. **new_id( )** is a function which creates a new identifier in the system. **OaddRlock** and **TaddRlock** are functions by which a role lock is stored in the role sets $o.role$ and $T.role$, respectively. The method t issued by the transaction T with a role R is performed on an object o as follows:
perform (*t*, *o*, T)

1. $Input(t)$: a method t is an input type.
   if some role lock R' in $o.role$ conflicts with the role R ($R' \rhd R$), **then** T is aborted;
   else {**OaddRlock**(o, R, τ, _);
   for every $<id', R', \tau', id''>$ **in** $T.role$,
   OaddRlock (o, R', τ', id');}
2. $Output(t)$: a method t is an output type.
   for every $<id', R', \tau', id''>$ **in** $o.role$, {
   TaddRlock (T, R, τ', id');
   OaddRlock (o, R, τ, id);
   for every $<id', R', \tau', id''>$ **in** $o.role$,
   if R' = R and $\tau > \tau'$, {
   $<id', R', \tau', id'>$ is removed from $o.role$;
   delRlock (id');}}
3. The method t is performed on the object o.□

The functions **TaddRlock**, **OaddRlock**, and **delRlock** are realized as follows:
TaddRlock(o, R, τ, id){ a tuple <**new_id**( ), R, τ, $id>$ is added to $o.role$;}
OaddRlock(T, R, τ, id){ a tuple <**new_id**( ), R, τ, $id>$ is added to $T.role$;}
delRlock(id){ **for** every object o, {every tuple $<id'$, R', τ', $id>$ is removed from $o.role$;
delRlock(id');}}

In the one-server model, all objects are stored in one computer. Objects are locked in role locks by using a $role\ lock$ table $RL(rlid, object, role, time, p\_rlid, tid)$ as shown in Figure 6. The attribute $rlid$ shows an identifier of a role lock. The attribute tid shows an identifier of a transaction which issues a role lock request. Suppose an object o is locked in a mode R by a transaction T at time τ. A tuple $<id, o, R, \tau,$ _, $T>$ is stored in the RL table where id is an identifier of the $role\ lock$ $<o, R, \tau>$. In the meantime, another transaction T' with a role R' derives data from the object o and then writes the data into another object o'. Hence, the object o' is locked in a role lock R', i.e. a tuple $<id', o', R', \tau',$ _, $T'>$ is added to the RL table. Here, "_" shows null value. At the same time, the role lock R on the object o is also held on the object o' [Figure 7]. That is, a tuple $<id'', o', R, \tau, id, T'>$ is added to the RL table. Here, id'' is the identifier of the tuple. The attribute p_rlid shows the identifier id of the role lock R held on the object o.

RL	rlid	object	role	time	p_rlid	tid
	id	o	R	τ	-	T
	id'	o'	R'	τ'	-	T'
	id''	o'	R	τ	id	T'

Figure 6. RL table.

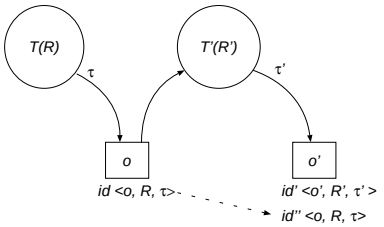

Figure 7. Role locks.

Suppose an object o is updated. Every role lock $<o, R, \tau>$ in $o.role$ is now obsolete. For every obsolete role lock $<o, R, \tau>$, the tuple $<id, o, R, \tau,$ _, $T>$ showing the role lock $<o, R, \tau>$ held by a transaction T is deleted from the RL table. At the same time,

every tuple whose p_rlid is equal to id, i.e. $<id', o', R, \tau, id, T'>$ is removed from the RL table. Then, every tuple whose p_rlid is id' is also deleted from the RL table. Thus, the cascading removal of tuples is realized by performing the function **delRlock**(id).

4.2 Multi-server model

In a distributed system, an RL table is maintained in each computer to store role locks on local objects. If a computer notifies the other computers of deletions of role locks each time one object is updated, the larger communication overheads are implied. Hence, we take a following strategy if a transaction T with a role R updates an object o in a computer c.

[Manipulation of access request $<o, t>$]

1. Each computer maintains a role lock (RL) table for locking local objects.
2. The RL table in the computer c is updated if some role lock R' on an object o is obsolete. Here, a tuple $<id, o, R, \ldots>$ is removed from the RL table. The value id of the attribute $rlid$ of the tuple removed is stored in a file ID.
3. The computer c periodically sends the file ID to all the other computers. $ID := \phi$ after sending ID.
4. On receipt of ID, every tuple whose p_rlid is in ID is removed from the RL table by **delRlock**(id) for every id in ID. If a tuple $<id'$, $\ldots, id>$ is removed here, id' is added to the file ID in the same way as step 2.□

Even if a role lock R gets obsolete in a computer, the role lock R is not soon released on every object in another computer. The obsoleteness of the role lock R is gradually propagated in networks. The shorter the period, the more consistent the RL table are but the more communication is required.

4.3 Commitment

A transaction takes the strict two-phase locking protocol [2] for concurrency control. That is, every object is locked before manipulated and all the locks held by a transaction are released on commitment or abort of the transaction.

Each time a transaction manipulates an object o, role locks are stored in the temporary role set variable $o.temprole$ in stead of $o.role$. If the transaction commits, role locks in $o.temprole$ are merged into $o.role$ at the same time the lock on the object o is released. If the transaction aborts, the role locks are just erased in $o.temprole$.

5 Concluding Remarks

We discussed how to prevent illegal information flow among objects in the role-based access control (RBAC) model. We newly introduced $role\ locks$ on objects. An object is locked in a role lock before a transaction manipulates the object. If the object cannot be locked, the transaction is aborted since illegal information flow might occur by performing the transaction. Role locks are not released even if the transaction commits. We discussed what role locks held on objects are obsolete and how to release the obsolete role locks. We also discussed how to implement the role locks.

References

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. In *Mitre Corp. Report*, pages 74–244, 1975.

[2] P. A. Bernstain, V. Hadzilaces, and G. N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[4] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. of 15th NIST-NCSC Nat'l Computer Security Conf.*, pages 554–563, 1992.

[5] A. Goldberg. Smalltalk-80: The interactive programming environment. *Reading. Addison-Wesley*, 5(2):169–172, 1984.

[6] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, 1973.

[7] Object Management Group Inc. The Common Object Request Broker : Architecture an d Specification. *Rev. 2.1*, 1997.

[8] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *IEEE Trans. on Knowledge and Data Engineering*, 9(4):524–538, 1997.

[9] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[10] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[11] Z. Tari and S. W. Chan. A role-based access control for intranet security. *IEEE Internet Computing*, 1(5):24–34, 1997.