IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4917

A Policy Translation Algorithm - Enabling any Hierarchical Key Assignment Scheme to Enforce Non-Hierarchcal Access Policies

Jyh-haw Yeh
Department of Computer Science, Boise State University, 1910 University Drive, Boise, Idaho 83725, USA,
jhyeh@cs.boisestate.edu

ABSTRACT

An access control policy of an organization specifies the access rules among subjects and objects. Depending on security clearance or data sensitivity, subjects and objects are divided into classes. The organization is hierarchical if classes are a partial ordered set (POset) on the access relation. Akl and Taylor first proposed a key assignment scheme to enforce access control in a hierarchy by assigning derivable keys to different classes. In the literature, most papers in the field focused on inventing different key assignment strategies to enforce such hierarchical policies. However, in practice, more complex policies may be necessary for real systems where transitive and antisymmetric exceptions are involved. This paper presents an algorithm which is capable of translating any policy from a hierarchy-with-exception to a hierarchy so that all existing hierarchical key assignment schemes can be used to enforce this richer set of policies.

INTRODUCTION

Based on security clearances, different subjects may access objects with different sensitivity. To model such system, all subjects and objects with the same level of clearance or sensitivity are assigned to one security group or class. An access policy specifies which class may access another. The most simple policy model exhibits a hierarchical structure with three POset properties. The three properties are reflexive, transitive and antisymmetric. The reflexive property is necessary for any system because a class should always have the right to access its own data. However, the other two properties are not always required in a system, especially in a distributed environment. The transitive property usually forms an access chain which means that if *A* can access *B* and *B* can access *C*, then *A* can access *C*. This property makes the system less flexible on policy development. The antisymmetric property considers two classes to be equivalent if they can access each other. This property puts too many restrictions on class partition. Fortunately, with transitive exceptions, this restriction can be relaxed. Two classes having the right to access each other are not required to be equivalent if there are transitive exceptions which make these two classes having different access rights to or from other classes. By taking out or relaxing these two properties, the policy domain could be enlarged from a hierarchy to a hierarchy-with-exception. This new domain includes all hierarchical policies and those policies with transitive and antisymmetric exceptions.

Akl and Taylor first proposed a key assignment scheme [1] to enforce access policies in a hierarchy. One key is assigned to each class. Each class uses its assigned key to protect data against illegal accesses. The scheme assigns keys in such a way that each class *A* is capable of deriving the key for class *B* from its own key only if the policy allows *A* to access *B*. The key derivation coincides with the access policy among classes. Therefore, the access policy can be enforced by these assigned keys. After Akl and Taylor presented their scheme, many different key assignment schemes [2-14] have been proposed in the literature. Most of them tried to find different ways to assign keys to enforce the same set of policies, i.e., policies in a hierarchy. The applicable domain of policies has never been studied. This paper presents a richer set of policies, hierarchy-with-exception, and proposes an algorithm to translate any policy in the new domain to a corresponding policy in a hierarchy. Therefore, any policy in a hierarchy-with-exception can be enforced by applying any existing key assignment scheme to the corresponding policy in a hierarchy.

This paper is organized as follows. Section 2 shows how to represent an access control policy using a matrix. Section 3 presents a generic algorithm to translate any policy from a hierarchy-with-exception to a hierarchy. Section 4 argues that the keys assigned to classes in the translated hierarchical policy using any existing key assignment scheme can be used to enforce the original hierarchy-with-exception policy. Section 5 concludes this paper.

MATRIX REPRESENTATION FOR POLICY

An access policy can be represented using a matrix. Suppose that a system's subjects and objects are divided into *n* distinct classes $C = \{C1, C2 \ldots Cn\}$. Under this partition, an n × n matrix is enough for specifying any access policy among these *n* classes if both *i*-th row and *i*-th column of the matrix represent *i*-th class.

This section defines three different forms of matrices for an access policy. A matrix *L1* of the first form only specifies which other classes each class may access. Its definition is as follows:

$$L_1[i,j] = \begin{cases} 1 & \text{if } C_i \text{ can access } C_j \\ 0 & \text{otherwise} \end{cases}$$

A matrix *L2* of the second form is derived from *L1* to explicitly indicate the existence of transitive exceptions. Its definition is as follows:

$$L_2[i,j] = \begin{cases} 1 & \text{if } L_1[i,j] = 1 \\ -1 & \text{if } \exists\, k_1, k_2, \ldots, k_m, \text{ where } 1 \le k_i \le n,\ i = 1,2,\ldots,n \text{ and } 1 \le m \le n-2, \\ & \text{such that } L_1[i,k_1] = L_1[k_1,k_2] = \\ & \ldots. = L_1[k_{m-1},k_m] = L_1[k_m,k_j] \\ & = 1 \textit{ and } L_1[i,j] = 0 \\ & \text{(transitive exception from class } C_i \text{ to class } C_j\text{)} \\ 0 & \text{otherwise} \end{cases}$$

A matrix *L3* of the third form can be generated from *L2* by further including more information of transitive exceptions. The third form *L3* will be used as an input to the translation algorithm presented in the next section. Its definition is as follows:

$$L_3[i,j] = \begin{cases} 2 & \text{if } L_2[i,j]=1 \text{ and } \exists k, \text{where} \\ & k \neq i,j, \text{such that } L_2[j,k]=1, \\ & \text{but } L_2[i,k]=\text{-}1 \\ & (\text{class } C_j \text{ is an intermediate} \\ & \text{class for a transitive} \\ & \text{exception from } C_i \text{ to } C_j) \\ L_2[i,j] & \text{otherwise} \end{cases}$$

The transformations from *L1* to *L2* and then from *L2* to *L3* can be accomplished by the following two algorithms. Algorithm 2.1 computes the transitive closure *L1** of *L1* and compares *L1** and *L1* to discover the existence of transitive exceptions.

Algorithm 2.1 *Let L1 be a policy matrix of the first form. Then the following algorithm translates L1 to a policy matrix L2 of the second form.*

L2 = L1 - (L1* - L1) = 2L1 - L1*

Given a matrix *L2* of the second form, Algorithm 2.2 generates a policy matrix *L3* of the third form by finding out those intermediate classes for transitive exceptions.

Algorithm 2.2 *Let L2 be a policy matrix of the second form. Then the following algorithm translates L2 to a policy matrix L3 of the third form.*

```
for i ← 1 to n
  for j ← 1 to n
    {
     L3[i,j] = L2[i,j]
     if L2[i,j] = 1
       for k ← 1 to n
         if L2[j,k] = 1 & L2[i,k] = -1
           {
            L3[i,j] = 2
            break
           }
    }
```

An example of a matrix *L1* of the first form with 6 classes is given in Table 1. Table 2 and 3 show the resulting *L2* of the second form and *L3* of the third form by applying algorithm 2.1 and 2.2 respectively.

To test whether a policy is hierarchical, either one of the above two algorithms can be used. If a policy has the same first, second and third forms, then there is no transitive exceptions, and hence, it is hierarchical. Any policy in a hierarchy is also in a hierarchy-with-exception. For each of those policies not in a hierarchy but in a hierarchy-with-exception, its first form is different from its second form since the second form will have some -1 entries representing the existence of transitive exceptions. An algorithm provided in the next section tries to translate any hierarchy-with-exception policy to a hierarchical policy with possibly more classes, and then any existing hierarchical key assignment scheme can be applied to enforce it.

Table 1: A matrix L1 *of the first form with 6 classes*

Ci	1	2	3	4	5	6
1	1	1	1	1	0	1
2	0	1	0	1	1	1
3	0	0	1	0	1	1
4	0	0	0	1	0	1
5	0	0	0	0	1	1
6	0	0	0	0	0	1

Table 2: A matrix L2 *of the second form*

Ci	1	2	3	4	5	6
1	1	1	1	1	**-1**	1
2	0	1	0	1	1	1
3	0	0	1	0	1	1
4	0	0	0	1	0	1
5	0	0	0	0	1	1
6	0	0	0	0	0	1

Table 3: A matrix L3 *of the third form*

Ci	1	2	3	4	5	6
1	1	**2**	**2**	1	**-1**	1
2	0	1	0	1	1	1
3	0	0	1	0	1	1
4	0	0	0	1	0	1
5	0	0	0	0	1	1
6	0	0	0	0	0	1

TRANSLATION ALGORITHM

Since a matrix represents an access policy among classes, these two words "matrix" and "policy" will be used interchangeably hereafter. Given a hierarchy-with-exception policy *L1* of the first form, Algorithm 3.1 presented in this section translates it to a corresponding hierarchical policy *M1* of the first form, in which the dimension of *L1* is less than or equal to the dimension of *M1*. The growing dimension from *L1* to *M1* is because a class may spawn another class during the translation if it is an intermediate class for any transitive exception. The intermediate class information of a policy is explicitly indicated in its third form. Therefore, the translation Algorithm 3.1 is accomplished by first applying Algorithm 2.1 and 2.2 to transfer *L1* to *L3* and then applying some other rules to map *L3* to *M1*.

Algorithm 3.1 *The following five steps translate an n × n policy matrix L1 to its corresponding m × m matrix M1, where n d•m and L1 is a matrix representation for a policy in a hierarchy-with-exception and M1 is its corresponding matrix representation in a hierarchy.*

1. Transfer L1 of the first form to L2 of the second form and then to L3 of the third form by performing algorithms 2.1 and 2.2 respectively.
2. Determine the set of indexes for intermediate classes I = {i | □ 1 ≤ i ≤ n, if □ k, where 1 ≤ k ≤ n, such that L3[k,i] = 2}.
3. For each class Ci, spawn another class Ci' if i □ I.
4. Create an m × m matrix M1, in which m = n + |I|. Both rows and columns of M1 are indexed as L1 except a new row i' and a new column i' are inserted right after all i □ I.
5. Fill each entry of M1 as follows.

$$M_1[i,j] = \begin{cases} 1 & \text{if } i = j \text{ or} \\ & \text{if } i' \text{ does not exist} \\ & \text{and } L_3[i,j] = 1 \text{ or } 2 \\ 0 & \text{otherwise} \end{cases}$$

$$M_1[i',j] = \begin{cases} 1 & \text{if } L_3[i,j] = 1 \text{ or } 2 \\ 0 & \text{if } L_3[i,j] = 0 \text{ or } -1 \end{cases}$$

$$M_1[i, j'] = \{ 0 \quad \text{for all } i \text{ and } j$$

$$M_1[i', j'] = \begin{cases} 1 & \text{if } i' \neq j' \\ 0 & \text{otherwise} \end{cases}$$

Table 4 gives the resulting matrix *M1* after applying Algorithm 3.1 to the matrix *L1* in Table 1, where *M1* is simply a hierarchical policy. Algorithm 2.1 could be used to test whether *M1* is hierarchical. With the transferred matrix *M1*, any existing key assignment scheme in literature can be used to assign keys to enforce *M1* in the expanded system. However, it is the policy *L1* in the original system, not the policy *M1* in the expanded system, which needs to be enforced. Next section will introduce a two-key assignment concept and it will become clear that enforcing *M1* is actually equivalent to enforcing *L1*.

SEMANTICS BEHIND THE TRANSLATION

Algorithm 3.1 translates a policy *L1* to another policy *M1* with possibly more classes. It seems that these two policies are different. However, with proper interpretation, these two policies are actually the same.

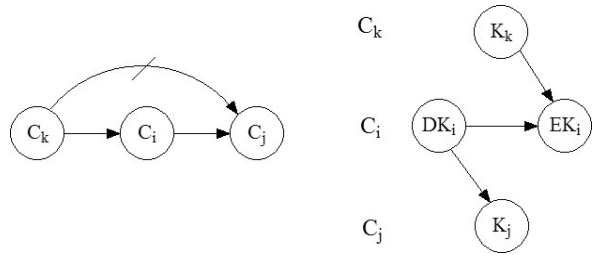
All existing key assignment schemes only assign one key per class. Under this scenario, it is impossible to enforce transitive exceptions. That is because if there is a transitive exception from *Ck* to *Cj*, in which another class *Ci* is an intermediate class for this exception, then *Ck* can always derive *Cj*'s key by first deriving *Ci*'s key. This one-key assignment dilemma for transitive exceptions can be solved by assigning two keys to all intermediate classes, where one key is for encryption and the other key is for derivation. The way to enforce transitive exceptions by two-key assignments is depicted in Figure 1.

In Figure 1, *Ck* and *Cj* have only one key *Kk* and *Kj* respectively. They can be used for both encryption and derivation. The intermediate class *Ci* has *DKi* as its derivation key and has *EKi* as its encryption key. *Ck* can access *Ci* and *Ci* can access *Cj* since *Kk* can derive *EKi* and *DKi* can derive *Kj*. However, *Ck* can not access *Cj* because there is no key derivation path from *Kk* to *Kj*.

Table 4: A translated hierarchical policy M1 *from a hierarchy-with-exception policy* L1 *in Table 1*

Ci	1	2	2′	3	3′	4	5	6
1	1	1	0	1	1	1	0	1
2	0	1	0	0	0	0	0	0
2′	0	1	1	0	0	1	1	1
3	0	0	0	1	0	0	0	0
3′	0	0	0	1	1	0	1	1
4	0	0	0	0	0	1	0	1
5	0	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0	1

Figure 1: The way to enforce a transitive exception by assigning two keys to an intermediate class, where DKi *is the derivation key and* EKi *is the encryption key.*



There are two access semantics for a class, which are "who can access the class" and "who can be accessed by the class". The first semantic is for encryption because a class can use an encryption key to encrypt data to control who can access the class. The second semantic is for derivation because a class could use a derivation key to derive another class's encryption key to access its data. The reason for a class to spawn another class in Algorithm 3.1 is to separate these two access semantics. Suppose that any intermediate class *Ci* in *L1* has two assigned keys *DKi* and *EKi* as in the Figure 1. After the translation from *L1* to *M1*, if *Ci* spawns *Ci'*, then *Ci* in *M1* inherits the encryption semantic and get the encryption key *EKi*, whereas *Ci'* in *M1* inherits the derivation semantic and get the derivation key *DKi* from the original *Ci* in *L1*. This separation for two access semantics is the core to enforce transitive exceptions.

The question left is that how we assign two keys to all intermediate class *Ci* in *L1*. This paper does not try to propose any new key assignment scheme to assign two keys; instead it applies any existing one-key assignment scheme to assign one key to all classes in the corresponding hierarchical policy *M1*, then combines the keys assigned to *Ci* and its spawned class *Ci'* in *M1* together for the class *Ci* in the original policy *L1*. The following list gives the itemized summary.

1. Translate a policy *L1* in a hierarchy-with-exception to a corresponding policy *M1* in a hierarchy by applying Algorithm 3.1. Any intermediate class *Ci* will spawn another class *Ci'*. Step 5 in Algorithm 3.1 makes sure that *Ci* inherits the encryption semantic and *Ci'* inherits the derivation semantic. That is, any other class can access *Ci* in *L1* should still be capable of accessing C*i* but not *Ci'* and any other class can be accessed by *Ci* in *L1* should be capable of being accessed by *Ci'* rather than *Ci*.
2. Apply any existing one-key assignment scheme to assign keys to classes in *M1*. Let *Ki* be the key for *Ci* and *Ki'* be the key for *Ci'* if *Ci'* exists.
3. For each intermediate class *Ci* in *L1*, let *Ki* be its encryption key and *Ki'* be its derivation key. For each non-intermediate class *Ci* in *L1*, let *Ki* be both its encryption and derivation key.

With the assigned key(s) for each class *Ci* in *L1*, *Ci* is capable of encrypting its own data using the encryption key to protect against illegal accesses from other classes and is capable of accessing other classes' data by first deriving their encryption key using its own derivation key.

CONCLUSION

A simple and generic algorithm is proposed in this paper to extend the domain of enforceable policies, from a hierarchy to a hierarchy-with-exception, for any existing key assignment scheme in literature. The algorithm accomplishes the extension by first performing a sequence of translations from a policy in a hierarchy-with-exception to a corresponding policy in a hierarchy, and then any existing key assignment scheme can be applied to assign keys to classes in the translated policy. Section 4 describes how to use these assigned keys of the translated policy in a hierarchy to enforce the original policy in a hierarchy-with-exception. The semantics behind the translations are also addressed in Section 4.

REFERENCES

1. S.G. Akl, P.D. Taylor, Cryptographic Solution to a Problem of Access Control in a Hierarchy. *ACM Transactions on Computer Systems*, V1, N3, pp 239-248, 1983.

2. S.J. Mackinnon, P.D. Taylor, H. Meijer, S.G. Akl, An Optimal Algorithm for Assigning Cryptographic Keys to Control Access in a Hierarchy. *IEEE Transactions on Computers*, V(c-34), N9, pp 797-802, 1985.

3. R. S. Sandhu, Cryptographic Implementation of a Tree Hierarchy for Access Control. *Information Processing Letters*, V27, pp 95-98, 1988.

4. L. Harn, H.Y. Lin, A Cryptographic Key Generation Scheme for Multilevel Data Security. *Computers & Security*, V9, No6, pp 539-546, 1990.

5. L. Harn, Y.R. Chien, T. Kiesler, An Extended Cryptographic Key Generation Scheme for Multilevel Data Security. *Fifth Annual Computer Security Application Conference*, pp 254-262, 1989.

6. B.M. Shao, J.J. Hwang, P. Wang, Distributed Assignment of Cryptographic Keys for Access Control in a Hierarchy. *Computers and Security*, V13, N1, pp 79-84, 1994.

7. H.T. Liaw, A Dynamic Cryptographic Key Generation and Information Broadcasting Scheme in Information Systems. *Computers and Security*, V13, N7, pp 601-610, 1994.

8. M.S. Hwang, W.P. Yang, A New Dynamic Cryptographic Key Generation Scheme for a Hierarchy. *1994 IEEE Region 10's Ninth Annual International Conference*, pp 465-468.

9. T.C. Wu, T.S. Wu, W.H. He, Dynamic Access Control Scheme Based on the Chinese Remainder Theorem. *Computer Systems Science and Engineering*, V10, N2, pp 92-99, Apr 1995.

10. H.M. Tsai, C.C. Chang, A Cryptographic Implementation for Dynamic Access Control in a User Hierarchy. *Computers and Security*, V14, N2, pp 159-166, 1995.

11. S.J. Wang, J.F. Chang, A Hierarchical and Dynamic Group-Oriented Cryptographic Scheme. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, V(E79-A), N1, pp 76-85, 1996.

12. F.K. Tu, C.S. Laih, W.C. Kuo, Cryptanalysis of a New Cryptographic Solution for Dynamic Access Control in a Hierarchy. *1997 Information Security Conference*, pp 104-108.

13. Y.M. Tseng, J.K. Jan, A Scheme for Authorization Inheritance in a User Hierarchy. *1997 Information Security Conference*, pp 109-115.

14. C.H. Lin, Hierarchical Key Assignment without Public-Key Cryptography. *Computers and Security*, V20, N7, pp 612-619, 2001.