Improving Indirect Key Management Schemes of Access Hierarchies

Brian J. Cacic and Ruizhong Wei
(Corresponding author: Ruizhong Wei)

Department of Computer Science, Lakehead University
Thunder Bay, Ontario P7B 5E1, Canada

(Received Oct. 5, 2005; revised and accepted Jan. 3, 2006)

Abstract

This paper examines possible modifications to indirect key management schemes that may improve their performance and efficiency for use within access hierarchies. A new method is proposed which uses a dedicated HMAC construction as the key transformation function, a new addressing strategy to improve accessibility verification, and a cached key update strategy which seeks to minimize key updates in large environments when immediate changes to the hierarchy are required. The proposed method can be applied to simple access hierarchies, and a modification is proposed which allows more complex access hierarchies to be addressed.

Keywords: Key management, access control, access hierarchy, and HMAC

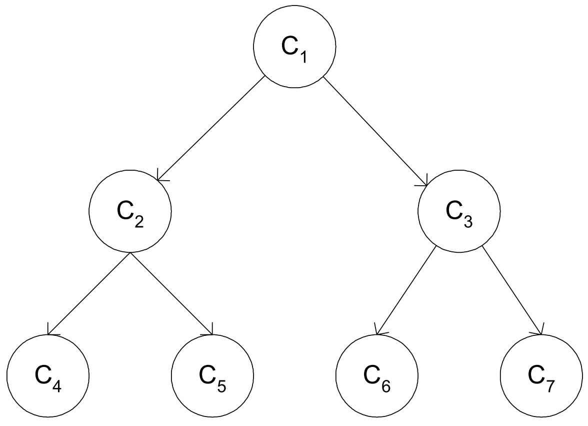

Figure 1: An example of an access hierarchy

1 Introduction

Many of our organizations in society are structured as hierarchies. These social hierarchies help to establish levels of accountability and authority within our organizations.

In most instances, we find that individuals in higher positions of the social hierarchy have greater authority and accountability. Previous research has demonstrated how to strengthen computer security by extending the idea and use of these social hierarchies to create access hierarchies for computer resources [2, 7].

In general, access hierarchies start by dividing members of a computer system into disjoint *security classes* (*class*), C_i (Figure 1). Resources stored at a particular class are only accessible by members of that class and any class that is principal and accessible. For example, resources stored at C_7 are accessible to C_7 and its principals C_1 and C_3 (Figure 1). In many instances, we may choose to model the access hierarchies after an organization's social hierarchy [7].

While access hierarchies have provided a valuable computer security service [8], they by themselves provide no support for the sharing of encrypted resources. That is, if a member belonging to a security class encrypts his resources with some enciphering key, members of security classes which are principal and accessible should be allowed to obtain the proper key and decrypt the resource. For this type of system to be supported within an access hierarchy we must seek some form of key management.

Akl and Taylor were the first to propose a method of key management within an access hierarchy [1]. In their method, each security class (C_i) is assigned a unique prime number which is used to calculate a security class's public parameter (PB_i). The enciphering key, K_i, for each security class, C_i, is calculated using the public parameter and a key transformation function based on the modular exponentiation arithmetic of RSA. To obtain the enciphering key of an accessible subordinate, a principal requires his enciphering key, his public parameter, the subordinate's public parameter, and the key transformation function. Accessibility is verified by the divisibility of the subordinate's and principal's public parameter. A symmetric cryptosystem is used to encipher all information within the hierarchy; thus, only the single enciphering key is required. The Akl-Taylor method, subsequent improvements, and subsequent derived methods

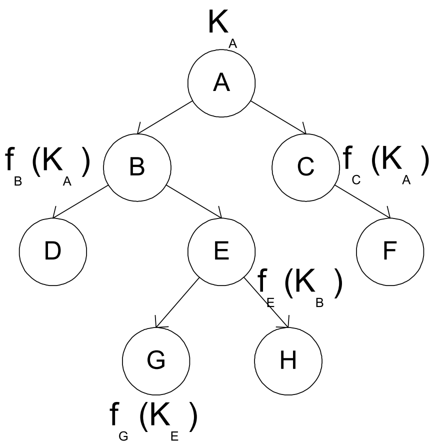

Figure 2: An example of Sandhu's key management method for an access hierarchy

are collectively known as *direct* key management schemes [1, 5, 9, 10, 11, 12].

However, there is a second method of key management, known as *indirect* key management, which has not received as much attention.

The first *indirect* key management scheme was proposed by Sandhu [15]. In it, a master key is assigned to the most principal member of a hierarchy, and subsequent keys for subordinates are calculated using some property of the subordinate, its immediate principal's key, and a one-way function, f (Figure 2). A principal generates a key for a subordinate by following the security class relationships within the access hierarchy and generating keys as relationships are traversed. A symmetric cryptosystem is used to encipher all information within the hierarchy; thus, only the single enciphering key is required. Sandhu's schemes and subsequent derivations which use a master key and recursive key operations are commonly referred to as *indirect* key management schemes [11].

Sandhu's method of *indirect* key management was met with criticism for its inefficiency and limited application because some regard the recursive nature of the key generation and transformation approach to be much less efficient than what could be achieved under the *direct* key management schemes [9, 11]. Application is also limited to simple access hierarchies where each security class has no more than one immediate principal [15].

More recently, Yang proposed an *indirect* key management scheme to address the applicability of Sandhu's method [18]. In Yang's method, multiple hash functions are used to address access hierarchies where security classes have more than one immediate principal (Figure 3). While using multiple hash functions does address the more complex hierarchy, Yang's method creates a shared key situation where the immediate principals of the shared subordinate must coordinate a key sharing policy between themselves (e.g. security class E in Figure 3). This is necessary because the subordinate is assigned a key com-

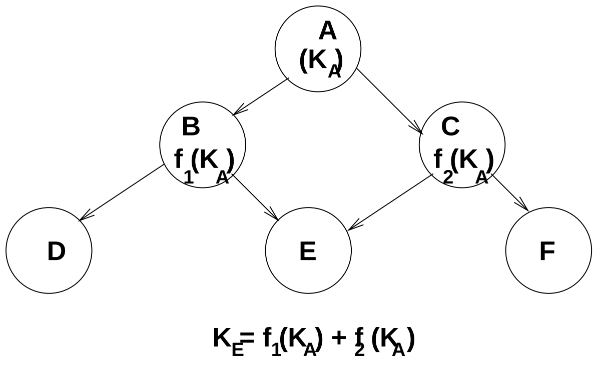

Figure 3: An example of Yang's key management scheme for an access hierarchy

posed from each immediate principal's key; thus, a principal wishing to access the shared subordinate requires the necessary keys from the immediate co-principals. Also in Yang's method, situations can arise where verification of the accessibility between principal and subordinate fails and time is lost traversing the hierarchy only to generate a false key.

In this paper, we sought to examine indirect key management schemes and look for ways of improving their performance and efficacy within access hierarchies. In the following sections we present a formal description of our proposed indirect key management scheme, and discuss our proposed modifications, their rationale, and security of the scheme.

2 Our Proposed Indirect Key Management Scheme

In this section we formally propose our indirect key management scheme and our modifications to the key transformation and key update operations, and our modifications to address more complex access hierarchies.

First, we begin with a set of assumptions:

- The access hierarchy is defined and controlled by some trusted central authority (CA). It is assumed that the CA is in a secure environment that provides no viable communication channel vulnerable to attack.
- All users within the CA's environment are divided into security classes, $SC = C_1, ..., C_n$, which are partially ordered by the binary relation $\leq$. The resulting relations, $C_j \leq C_i$, means that users belonging to security class C_i have access to information stored at the subordinate security class C_j; however, the reverse relation does not hold. That is, subordinates are not allowed access to information stored at the principal (C_i).
- Users belonging to a security class C_i only know direct relationships. That is, members in C_i know who

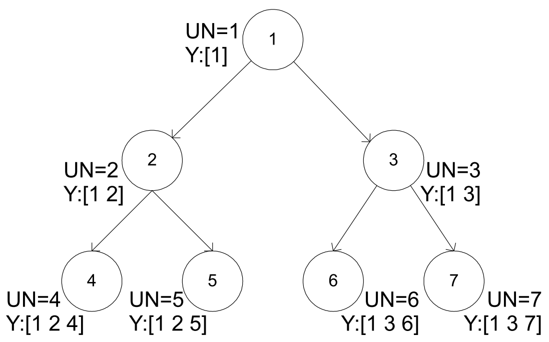

Figure 4: A tree-structured access control hierarchy

their direct principal security class (C_p) is, and who their direct subordinate classes (C_s) are.

- For now, we assume the hierarchy is represented as a simple tree. That is, no security class has more than one direct principal security class, and the most principal security class is located at the root of the tree. Later, we modify the method to handle more general hierarchies represented as directed acyclic graphs.
- All keys within the hierarchy expire after time t_r. After which, new keys are generated and assigned to the security classes. Maintaining the common principle of good key management [16], we place this restriction to discourage exhaustive key search attacks and cryptanalysis of key-encrypted information. In practice, the key assigned to a security class can be a session key while the data are stored using master keys.

We denote $H(K|M)$, to be the hashed message authentication code (HMAC, defined in [3]) that uses the dedicated secure hash function (e.g., SHA-1 (H)), with a key (K), concatenated with a security class property (M).

Hierarchy preparation and key assignment proceeds as follows:

1) Each security class (C_i) in the hierarchy contains three properties: a human readable name (M_i), a unique positive integer (N_i), and a *path address* array (Y_i). These properties are the public parameters other security classes are allowed to view.
2) The human readable name, M, is the name of the security class that allows users to discriminate one security class from another. Common names can be any length.
3) A unique positive integer (N) is assigned sequentially, starting at 1 at the root and in a left-to right top-down manner, to each security class in the hierarchy of Figure 4. Later, as we add classes to the hierarchy, regardless of their position within the hierarchy, we assign them the next integer in the sequence.

Table 1: Summary of public parameters assigned to security classes belonging to Figure 4

Unique Number (N)	Path Array (Y)
1	[1]
2	[1 2]
3	[1 3]
4	[1 2 4]
5	[1 2 5]
6	[1 3 6]
7	[1 3 7]

4) The path array (Y) acts as an address for a security class in Table 1. The address assigned to a security class records the N traversal path starting from the root to the security class. The last entry in a security class's path array should correspond to its N value.
5) The CA assigns the root of the tree (the most principal security class) a randomly generated 1024-bit master key, K_1, which is kept secret (see Section 3.2).
6) A security class C_k is assigned a key dependent on its direct principal security class C_i as follows:

$$K_k = H(K_i|N_k). \tag{1}$$

When a user belonging to a security class C_i wishes to derive the key for security class C_k, and C_k is the direct subordinate of C_i, then K_k is obtained using

$$K_k = H(K_i|N_k).$$

Otherwise, if C_k is not a direct subordinate of C_i, C_i proceeds as follows:

1) C_i retrieves the path array for C_k, Y_k.
2) Using a sequential search on the array, C_i checks for his N_i within Y_k.
3) If the search returns FALSE, then C_i knows that it lacks the sufficient permission to access security class C_k and does not proceed to generate the key. If the search returns TRUE, C_i stops and records the index x at which its N_i was located and proceeds to the next step.
4) Starting from $x+1$ to the end of Y_k, C_i generates the key using the HMAC. For example, if the portion of the array is [N_i,N_j,N_k] then the key derivation step is:

$$\begin{aligned} K_j &= H(K_i|N_j) \\ K_k &= H(K_j|N_k) \end{aligned} \tag{2}$$

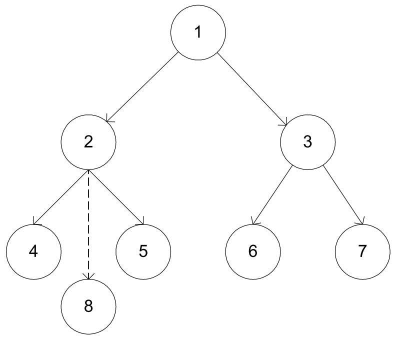

Figure 5: Adding a new security class to a leaf position in a simple tree hierarchy

2.1 Adding and Removing Security Classes from the Hierarchy

Ideally, it would simplify all key management solutions if the hierarchy remains static. Unfortunately, this is not always the case. As users come and go, or as an organization changes, the need to add and remove security classes from the hierarchy will arise. As such, our key management method should handle these changes.

We identified four cases for adding and removing security classes from the hierarchy. They are as follows:

1) Adding a security class to a leaf position,

2) Removing a security class from a leaf position,

3) Adding a security class to an interior position, and

4) Removing a security class from an interior position.

Adding and removing security classes to or from a leaf position is trivial. In Figure 5, 8 is added to a leaf position, becoming the new subordinate to 2. It is assigned a common name M_8 and a unique number $N_8 = 8$. The path array for 8 (Y_8) is created by inheriting the path array from 2 (Y_2 = [1,2]) and appending $N = 8$ to the end of the path (Y_8 = [1,2,8]). Removing a security class from a leaf position in the hierarchy can be done without any affect to any principal classes.

Adding and removing security classes to and from interior (non-leaf) positions presents some challenges. In studying the key management problem, we saw that all previous *direct* and *indirect* methods dealt with the problem in a similar manner. Their designers chose to immediately re-calculate and update the keys for the affected classes [9, 14, 11, 15, 18]. This may or may not be advantageous in all situations. For example, the necessity to immediately add or remove a security class cannot be delayed or overlooked, but the disturbance caused to users within the affected security classes, or the time and resources required to update the keys would be costly or inconvenient. In these instances, it would be better to have a method that could delay a key update until a more convenient time arises, or as in our method, a pre-specified key freshness time (t_r) expires.

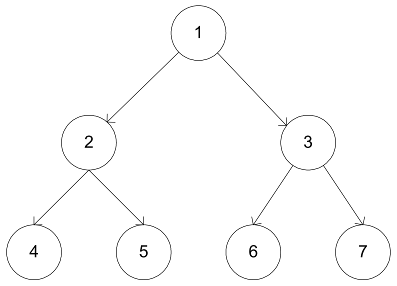

Figure 6: A simple tree hierarchy

To address this problem of key updates, we created an update strategy called the *cached key* update strategy. The cost associated with the method is that it requires a newly added security class to have additional storage allocated for one extra key (a key cache), and a modification to the key derivation process.

2.1.1 Cached Key Update Strategy

The *cached key* update strategy is best understood with illustrations. In Figure 6, we show our simple tree access hierarchy. For brevity, we refer to security classes by their Ns (e.g. (8)).

For internal additions to the hierarchy the rule-set is as follows:

- (Figure 7) If a new security class (28) is added between two classes whose key caches are empty, (1,2), the CA assigns the new class (28) a path address from 1 (Y_{28} =[1,28]) and a key from its direct principal class ($K_{28} = H(K_1|28)$). The CA also provides the key for the direct subordinate (2) to the new class (28), which the new class (28) will store in its key cache. If additional classes (25) are added to the new class (28) as direct subordinates, they are assigned a path and key relative to the new class (28) – (25: K_{25}=H(K_{28}|25), (Y_{25} = [1,28,25])).
- (Figure 8) If a new security class (38) is added between two classes where the subordinate key cache is empty (2) and the principal full (28), the new class (38) is given a path from 1 (Y_{38} =[1,28,38]), a key derived from its parent's key (K_{38}=H(K_{28}|38)), and a key from the direct subordinate (2) which the class (38) will store in its key cache. If additional classes (25) are added to the new class (38) they will receive a path containing the new class (Y_{25} =[1,28,38,25]) and a key generated from the new class (K_{25} =H(K_{38}|25)).
- (Figure 9) If a new security class (48) is added between two classes where each key cache is full

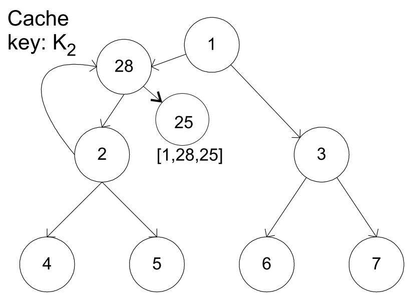

Figure 7: Adding a new security class between two classes with empty key caches

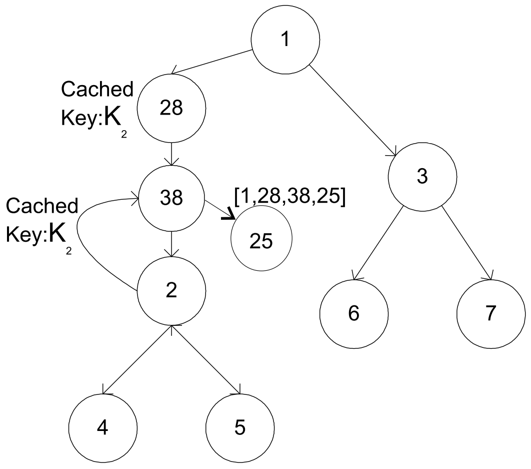

Figure 8: Adding a new security class between two classes were the subordinate's key cache is empty

(28,38), we initiate an update. Key caches are cleared (28,38). The new class (48) is given a path from 1 (Y_{48} =[1,28,48]) and a key derived from its principal's key (K_{48} =H($K_2$8|48)). The classes subordinate to the new class (38,2,4,5) have their keys regenerated and paths updated.

- (Figure 10) If a new security class (58) is added as a principal to a class that has a full key cache (28), we initiate an update. Key caches are cleared. The new class (58) is given a path from 1 [1, 58]. The classes subordinate to the new class (28,38,2,4,5,...) have their keys regenerated and paths updated.

The process to remove a key follows a similar method to addition. The rule set is as follows:

- (Figure 11) If the security class being removed (8) has subordinates that are leaf-classes (9, 10), the leaf-classes (9, 10) are assigned to the principal (2). Because the paths from the principal (2) to new subordinates (9,10) is short, there are two options. First, if the update of keys and paths to the subordinates (9,10) would cause no inconvenience, then the keys and paths may be updated immediately. Otherwise,

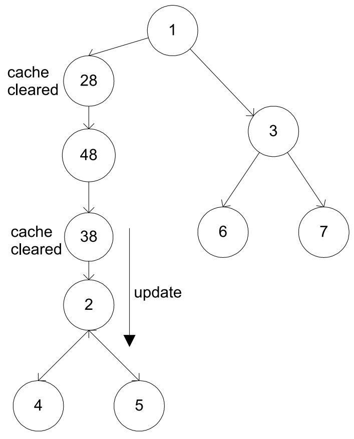

Figure 9: Adding a new security class between two classes with full key caches

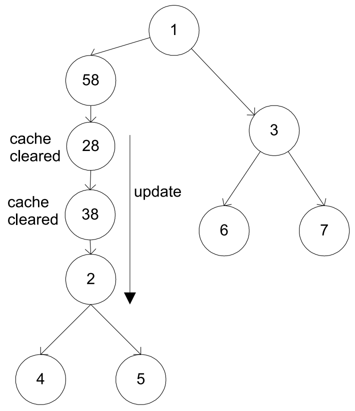

Figure 10: Adding a new security class as the principal to a class that has a full key cache

the principal (2) caches the key of the outgoing class (8).

- (Figure 12) If the security class being removed (5) has subordinates that are not leaf-classes (8,9,10), the principal (2) receives the outgoing class's (5) key to store in the key cache, and the subordinates belonging to the outgoing class (9,10) are added as subordinates to the principal (2).
- (Figure 13) If the security class being removed (8) is not a leaf-class and is subordinate to a principal whose key cache is full (2), the principal receives the subordinate classes (9,10) of the outgoing class (8), clears its (2) cache, and updates paths and keys to all its subordinate classes (9,10).

Each security class holding a cached key must modify its search strategy when searching a path array. For example, in Figure 7 if 28 requests the path array for 4

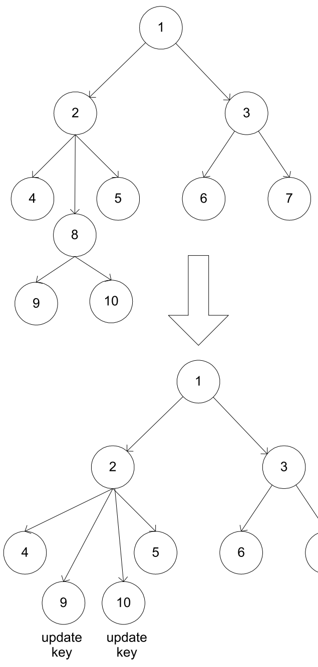

Figure 11: Security class (with leaf-classes) being removed

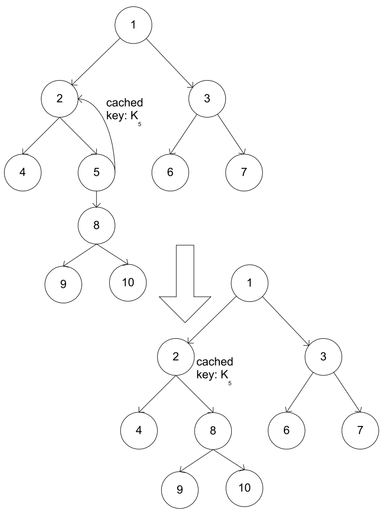

Figure 12: Security class (with non-leaf subordinates) being removed

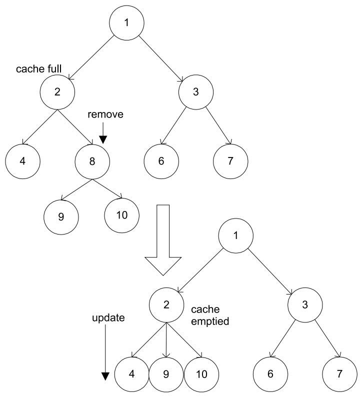

Figure 13: Removing a subordinate from a principal with a full key cache

(Y_4), then 28 searches Y_4 for its N and the N belonging to the cached key (K_2). If it locates the N belonging to its cached key (K_2), then 28 uses the cached key (2) to derive the key for 4 by following the path and using the cached key. Otherwise if 28 finds its N within Y_4, it uses its key to follow the path and recursively generate the key for 4 (see Equation 2).

A side benefit of the cached key strategy is that we may be able to accommodate additions where previous deletions occurred. For example, in Figure 11 if a class was added into the position previously held by 8, we could modify our addition strategy to have the CA simply reassign the new class a value of $N = 8$ and the key held in cache by 2 (K_8). This would suggest that rather than removing a class completely from the hierarchy, the better strategy might be to have the CA maintain a deletion list to keep track of classes that are removed. This way, if a new class is re-introduced to a deletion point, adding it can be accommodated much more easily than undergoing a completely new addition and key update.

2.2 Addressing Complex Hierarchies

In access hierarchies represented as directed acyclic graphs, a subordinate security class can have more than one direct principal class in Figure 14. Consequently, the subordinate requires a key that can allow both direct principal classes access to the subordinate.

With *indirect* approaches, directed acyclic graphs present issues regarding traversals of paths. Referring to Figure 14, when either 2 or 3 wishes to access information stored at 5, they must have some knowledge about the composition of 5's key. Similar to Yang's approach, we chose to generate the subordinate's key by composing the direct principals' keys. For example, in Figure 14 the key assigned to 5 would be

$$K_5 = H(H(K_2|5)|H(K_3|5)).$$

Thus, in order for either 2 or 3 to derive the key for 5, either 2 would require the knowledge of the sub-key $H(K_3|5)$, or 3 would require the knowledge of the sub-key $H(K_2|5)$. We chose to have the CA cache the sub-keys. A consequence of this approach is that principal classes which share a direct subordinate will rely upon the CA to provide the cached key.

However, the more immediate problem was how to represent this security class as having a key composed from two or more direct subordinates. Yang's solution was to use multiple hash functions, but we preferred the flexibility of having a single fast dedicated hash function within

Table 2: Modified path addresses for security classes in a DAG access hierarchy

Security Class	Path Address
1	(1)
2	(1, 2)
3	(1, 3)
4	(1, 2, 4)
5	((2 3), 5)
6	(1, 3, 6)
7	((4 5 6), 7)

the HMAC-method. The best solution we could devise modified the *path address* array (Y).

Using Figure 14 as an example, Table 2 shows how each security class's path address would appear under the modified addressing scheme. For brevity, we refer to security classes by their unique numbers (UN).

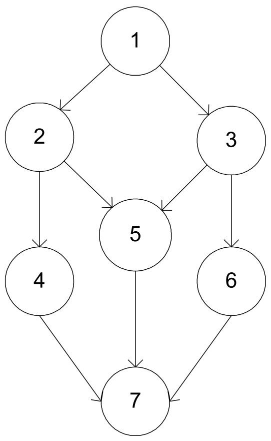

Figure 14: Directed acyclic graph structured hierarchy

Our modification was to change the structure of the path address from being an array to being a list that could contain nested lists. A nested list within the list (e.g. ((2 3) , 5)), indicates that the security class holds a key which is composed from the sub-key belonging to the security classes in the nested list. For example, from Table 2 the address list for security class 5, ((2 3) , 5), indicates that 5 is composed from the sub-keys of security class 2 and security class 3. Security classes that are not composed of sub-keys are simply represented as a list of numbers (e.g. 4, (1 , 2 , 4)). Next, we modified the key derivation process to reflect the modified *path list* addressing scheme.

If the first element in a *path list* is not a nested list (e.g. (1 , 2 , 4)), then we know that a direct path exists to the desired class and we operate on the list as if we were using the *path address* array from a security class in a simple tree hierarchy.

If the first element in a *path list* is a nested list (e.g. ((2 3) , 5)), we implement an *expand-and-search* strategy. For example, in Figure 14 if security class 1 wanted to access security class 5, it requests the *path list* for security class 5, (((2 3) , 5)), notices the nested list as the first element, and proceeds to follow the expand-and-search strategy:

1) The security class searches the nested list looking for its UN. In our example, 1 searches the list ((2 3) , 5) and does not find itself in the nested list.

2) If the UN is not located within the nested list, the address for the first element in the nested list is expanded. In our example, the list ((2 3) 5) is expanded and becomes (((1 2) 3) , 5).

3) The principal security class repeats steps 1-2 until it finds its UN within an expanded list. In our example, 1 will find itself in the expansion of 2's address list: (((1 2) 3) , 5). If 1 did not find itself in the expanded list of 2, it would move onto the next element, 3, and perform the expand-and-search again.

4) Once the UN is found within a list, the key represented for that list is generated. In our example 1 will create the key for 2 by following the path address ((((1 2) 3) , 5) $\rightarrow$ ((((K_2) 3) , 5).

5) At this point, search-and-expand stops and the security class will request the CA to produce the sub-keys for the other members of the sub-list. In our example, having generated the key for 2, 1 will stop expand-and-search and request the CA to produce 3's sub-key for 5: ((K_2, K_{3_5}), 5).

6) Having received the remaining sub-keys from the CA, the principal can combine them in order to produce the key for the desired subordinate. In our example, 1 will create 5's key using $H(H(K_2|5)|K_{3_5})$.

With this new derivation method, the best case scenario is that the first element in the nested list produces a valid key and the search-and-expand is aborted so that the remaining sub-keys can be requested. The worst case scenario is that all members of the nested list undergo search-and-expand and no keys are produced. This situation could occur frequently in weakly connected directed acyclic graph access hierarchies. For example, Figure 15 shows just such a hierarchy. If 1 were to request access to 4, it would spend time performing search-and-expand only to find that it lacked the proper access. Having spent time with the problem we leave it as open and state that although our newly devised *path list* addressing scheme could accommodate DAG hierarchies, it is not an optimal method for all DAG hierarchies.

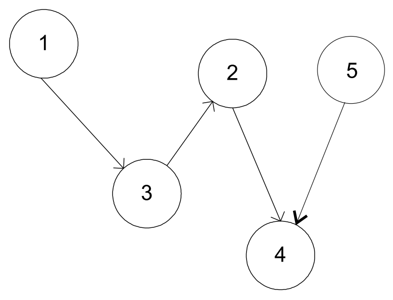

Figure 15: A weakly connected DAG hierarchy

3 Discussion

3.1 Proposed Modifications and Rationale

In our attempts to improve indirect key derivation schemes, we first sought an improved method of verifying accessibility relationships between principals and subordinates before undertaking a key derivation traversal. We noted the problems encountered in the schemes proposed by Sandhu and Yang, so our goal was to eliminate exhaustive searches of the hierarchy and false key generations. That will save time from path searching and key derivations. Previous research had suggested using long character strings delimiting the traversal path [15]. Our improvement over using a character string naming convention was to assign a unique integer identification number to each security class and provide each security class with an address array. The benefit is that searching an integer array and comparing integer values can occur much more quickly then string manipulations and string comparisons. The address array allows us to quickly verify accessibility relationships; thus, preventing unnecessary traversals and false key derivations. We still use character strings as names for security classes, but these are used to help individuals discriminate different security classes.

As we developed our method, we discovered some pragmatic problems with regards to key updates. We found that there could be situations where structural changes to the access hierarchy must be made immediately, but the resulting key updates that would occur may be too costly, in time and/or money, to undertake. The cached-key update strategy was designed to address this issue.

In our *cached key* update strategy, the size of the key cache determines the delay between key updates. Increasing the number of cached keys will increase the delay between key updates. However, increasing the size of the key cache will require some modifications to the rule-set of adding and removing classes. With one key cache, we are able to cache keys belonging to immediate subordinates. Consequently, in situations where more than one key from a subordinate would need to be cached, we currently initiate an update. With larger key caches the rule-sets will need to be modified to reflect the fact that keys from lower subordinates must be cached. The need for larger or smaller key caches will be dependent upon the nature of the organization the hierarchy represents and/or the key expiry and key update schedule an organization wishes to implement.

The *cached key* update strategy we proposed is to address concerns with the overhead and costs incurred if we update the hierarchy in response to every addition and deletion of a security class. As the name suggests, the *cached key* update strategy sacrifices a small amount of storage per affected security class. Updates to portions of the hierarchy are delayed if such updates would be costly or inconvenient. Fortunately for additions, the cached key is assigned to the newly added security class, thus it may be easier to assign a cached key to a new class than to update keys in the affected subordinates. A removed class provides its key to its principal so that the principal may access the inherited subordinates. Using a deletion list and cached keys, we may also be able to accommodate unique situations where classes are deleted, yet new classes are re-introduced into the same position sometime later. We should reiterate that the *cached key* update strategy may not be suitable for all situations. The nature of the keys and organization may warrant the simple *immediate* update strategy taken by previous *indirect* and *direct* methods.

In evaluating the key transformation function we decided to use a message authentication code (MACs) built from a single dedicated hash function; specifically SHA-1. The decision to use a single dedicated function and a single key transformation function was out of simplicity and security. Dedicated hash functions provide good throughput, and being able to use an unaltered "off-the-shelf" implementation of a single hash function helps to simplify the development and management of the system. However, the structure of our scheme is independent from the hash function we used. should the hash function being used become compromised, it could be substituted by another hash function without significantly altering the key transformation procedure. Another reason for choosing HMAC is that only one key should be stored for one security class in the scheme (for example, NMAC is not suitable for our scheme).

Finally, while the use of a single hash function with the key transformation function proved adequate for simple hierarchies, modifications were required to address complex hierarchies where a subordinate has more than one immediate principal. The procedure we designed to address this issue suffers from a similar key sharing problem found in the Yang method, but we are hopeful that a better method, either pragmatic or mathematical, will be found and we are re-addressing this issue with further research.

3.2 Security

The security of our proposed method lies in the underlying security of the master key and the SHA-1 dedicated hash function.

We chose the master key be at least 1024-bits in size, because the master key can derive all keys within the hierarchy. Such a large key size helps thwart random guessing of the master key. The probability of an attacker guessing the key would be 2^{-1024}. An exhaustive key search of the key would require the attacker to generate and test all 2^{1024} possible key combinations. We believe this key size can provide adequate protection of the master key.

We chose to specifically use SHA-1 as an example, because it produces a digest length of 160-bits, resulting in 2^{160} possible message digests. Under the birthday attack, to force a collision under SHA-1 would require an exhaustive search and comparison of at least 2^{80} message digests. However, without having access to the message digests that are being used as keys for security classes, the attacker would most likely have to generate all 2^{160} message digests and test each one he creates against each security class in the hierarchy. This should increase the complexity of the attack. If the attacker tries to simply guess a key, his probability of success is 2^{-160}.

Although there are collision attacks on SHA-1 recently, our scheme should still be safe. The main reason is that our scheme used a hash function as a one-way function with a secret key. The collision attacks do not help finding K from the knowledge of $H(K|M)$. In fact, even if a lot of collision values $x_i, i = 1, \ldots, n$, are found such that $H(x_i) = H(K|M)$, it is most likely that $x_i \neq K|M$ for $i = 1, \ldots, n$. So it is infeasible for a class to find the key of its direct principal class. However, to fully prove the security of our scheme, we need to use a random oracle assumption (see [4]). So we can use a random oracle R instead of using a hash function in the scheme. Here a random oracle is a map from $\{0,1\}^*$ to $\{0,1\}^\infty$ chosen by selecting each bit of $R(x)$ uniformly and independently for every x. Then we can use steps similar to the proofs of the scheme in [17] to our scheme. The main difference is that the scheme in [17] depends on Diffie-Hellman assumption (DDH), but our scheme depends on the random oracle assumption.

4 Conclusions

We have proposed an indirect key management scheme for access hierarchies which is modified from the schemes of [15] and [18].

In our scheme, an HMAC is used as a one-way function which is more efficient than the scheme in [15] which used DES. The scheme in [18] also used hash functions, but it requires many secure hash functions. Using a keying hash function is a practice solution.

Comparing to direct key management scheme, one disadvantage of indirect key management schemes is that it needs to find a path to a lower level node from a higher level node. Our scheme uses path arrays to take care of this problem. The scheme in [18] didn't consider the path search problem. The scheme in [15] used long character strings which is not as efficient as our methods. Moreover, the method in [15] only works for tree structured hierarchies.

One advantage of indirect key management scheme is that it can be used for dynamic access control problems, such as adding, deleting or modifying relationships between nodes (see [18]). So when one node changes, just all the related nodes need to refresh keys. In our scheme, we proposed a cached key update strategy which further enhanced this advantage of indirect key managements. Using this strategy, adding or removing a node will not cause key refreshing for other nodes in many situations. This strategy can be used in any indirect key management for access hierarchies.

Acknowledgement

Research supported by NSERC grant 239135-01.

References

[1] S. G. Akl and P. D. Taylor, "Cryptographic solution to a problem of access control in a hierarchy," *ACM Transaction on Computer System*, vol. 1, no. 3, pp. 239-248, 1983.

[2] D. Bell and L. LaPadula, *Secure Computer System Unified Exposition and Multics Interpretation*, Technical Report MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976.

[3] M. Bellare, R. Canetti and H. Krawczyk, "Keying hash functions for message authentication," *CRYPTO'96*, LNCS 1109, pp. 1-15, Springer-Verlag, 1996.

[4] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *First ACM Conference on Computer and Communication Security*, pp. 62-73, ACM Press, 1993.

[5] G. C. Chick and S. E. Tavares, "Flexible access control with master keys," in *Proceedings on Advances in cryptology*, LNCS 435, pp. 316-322, Springer-Verlag, New York, Inc., 1989.

[6] H. Dobbertin, A. Bosselaers, and B. Preneel, "Ripemd-160: A strengthened version of ripemd," in *Proceedings of the Third International Workshop on Fast Software Encryption*, pp. 71-82, Springer-Verlag, 1996.

[7] D. Ferraiolo and R. Kuhn, "Role-based access controls," in *15th NIST-NCSC National Computer Security Conference*, pp. 554-563, 1992.

[8] M. P. Gallahger, A. C. O'Connor, and B. Kropp, *The Economic Impact of Role Based Access Control*, NIST Planning Report 02–01, National Institute of Standards and Technology, 2002.

[9] L. Harn and H. Y. Lin, "A cryptographic key generation scheme for multilevel data security," *Computer Security*, vol. 9, no. 6, pp. 539-546, 1990.

[10] M. S. Hwang, "A new dynamic key generation scheme for access control in a hierarchy," *Nordic Journal of Computing*, vol. 6, no. 4, pp. 363-371, 1999.

[11] M. S. Hwang and W. P. Yang, "Controlling access in large partially ordered hierarchies using cryptographic keys," *Journal System Software*, vol. 67, no. 2, pp. 99-107, 2003.

[12] S. J. MacKinnon, P. D. Taylor, H. Meijer, and S. G. Akl, "An optimal algorithm for assigning cryptographic keys to control access in a hierarchy," *IEEE Transaction on Computers*, vol. 34, no.9, pp. 797-802, 1985.

[13] B. Prenel and P. C. V. Oorschot, "Mdx-mac and building fast macs from hash functions," in *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pp. 1-14, Springer-Verlag, 1995.

[14] I. Ray, I. Ray, and N. Narasimhamurthi, "A cryptographic solution to implement access control in a hierarchy and more," in *Proceedings of the seventh ACM symposium on Access control models and technologies*, pp. 65-73, ACM Press, 2002.

[15] R. S. Sandhu, "Cryptographic implementation of a tree hierarchy for access control," *Information Processing Letters*, vol. 27, no.2, pp. 95-98, 1988.

[16] B. Schneier, *Applied Cryptography (2nd ed.): Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., 1995.

[17] J. Wu and R. Wei, "An access control scheme for partially ordered set hierarchy with provable security," in *Proceedings of SAC'05*, pp. 223-245.

[18] C. Yang, *A Secure Object-Oriented Role-Based Access Control Model for Distributed Systems*, PhD thesis, University of Regina, Regina, Saskatchewan, Aug. 2003.

Brian Cacic received his M.Sc. in Mathematical Science from Lakehead University in 2004, and is currently studying law at the University of Western Ontario.

Ruizhong Wei received the B.Sc degree from the Suzhou University, China, in 1982, and the Ph.D degree from the University of Nebraska-Lincoln in 1998. Before joining Lakehead University, Canada in 2000, he worked at the Suzhou University, the University of Nebraska-Lincoln and the University of Waterloo. He is currently a Professor in Computer Science. His research interests include cryptography, network security and combinatorics. He is a Fellow of Institute of Combinatorics and its Application.