Heuristics for Improving Cryptographic Key Assignment in a Hierarchy

Anne V.D.M. Kayem, Patrick Martin, and Selim G. Akl
School of Computing, Queen's University
Kingston, Ontario, CANADA, K7L 3N6
kayem@cs.queensu.ca, martin@cs.queensu.ca, akl@cs.queensu.ca

Abstract

In hierarchical distributed systems, shared data access can be controlled by assigning user groups single cryptographic keys that allow high level users derive low level keys, but not the reverse. The drawback in this approach to key management is the requirement of replacing keys throughout the entire hierarchy whenever group membership changes, to preserve security. This paper presents two algorithms, based on a precedence tree graph model, that use a distance-based heuristic to minimize the cost of key assignment and replacement, respectively. In the average case, only the keys belonging to the group affected and its sub-tree are replaced. A complexity analysis and experimental results indicating performance improvements demonstrate the feasibility of the proposed algorithms.

1. Introduction

Distributed systems inspired the proliferation of web-based applications because they allow concurrent access to shared data. In such systems, data can be classified into a number of classes C_{K_i} such that $1 \leq i \leq n$ where n is the maximum number of user groups in the hierarchy and K_i is the cryptographic key with which the data is encrypted. Possession of a correct key grants a user access to the data. Practical applications of this concept of access control include web-based distributed collaborative systems, secure group communication environments, and sensor networks where access to shared data is an issue.

Cryptographic keys for the various user groups requiring access to part of the shared data in the system are defined by classifying users into a number of disjoint security groups U_i, represented by a partially ordered set $(S, \preceq)$, where $S = \{U_0, U_1, ..., U_{n-1}\}$[1]. In the partially ordered set $U_i \preceq U_j$ implies that users in group U_j can have access to information destined for users in U_i but not the reverse.

For instance, in Figure 1, using the scheme proposed by Akl and Taylor [1], a security provider (SP) classifies users into three groups U_0, U_1, and U_2 and controls access with a series of keys generated using the formula:

$$K_i = K_0^{t_i} \bmod M \quad ...(1)$$

where $M = p \times q$ is the product of two large primes, t_i a public exponent that defines a user group's relationship vis-a-vis other groups in the hierarchy, and K_0 a secret key selected by the SP situated at U_0. Users in U_0 can access data held by U_1 and U_2 using [1, 8]:

$$K_i = K_0^{t_i} = K_0^{t_j(t_i \div t_j)} = K_j^{(t_i \div t_j)} \,(\mathrm{mod} M) \quad ...(2)$$

When the result of $t_i \div t_j$ is not an integer the computation is infeasible and access is not allowed [1]. The security of this algorithm relies on the fundamental assumption that no efficient algorithm exists for extracting roots modulo M, if M is the product of two unknown large primes [9].

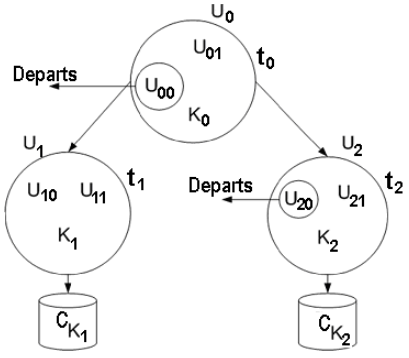

Figure 1. Example of a Key Tree

This method of key assignment has two advantages: first, users in any given group only need to store a single key rather than several which makes for better security, and second, sharing a common key alleviates the complexity of key assignment. The downside is that a change in

any of the groups' membership, requires a change of keys throughout the entire hierarchy. For example, in Figure 1, when a user U_{00} departs from U_0 both K_0 and the correlated keys K_1, and K_2 need to be changed to prevent the departed U_{00} from continuing to access C_{K_1} , and C_{K_2}. Likewise, when U_{20} departs from U_2 the keys K_0, and K_1 need to be changed in addition to K_2 so that K_0 can derive the new K_2. More importantly the change must guarantee that the new K_2 does not overlap with K_1 and unknowingly grant U_1 access to C_{K_2} or vice versa. This approach to key assignment is not scalable for environments with frequent group membership changes where meeting the goals of service level agreements is an additional constraint [14].

This paper proposes two algorithms to minimize the cost of key assignment. The first uses a distance based heuristic to guide the assignment of the exponents used to compute user group keys such that a collusion attack is avoided. Collusion occurs when two or more users at the same level in a hierarchy cooperate using their respective keys and a contrived function to compute a key that belongs to a user class higher up in the hierarchy, to which they are not entitled. The second employs an exponent replacement scheme that allows the system track of the validity of the exponents used to form the keys, to avoid re-assigning keys that have already been used or that are currently in use.

The paper is structured as follows. Section 2 gives a review of related work. In section 3 we present our algorithms and give illustrative examples to show how they work. A complexity analysis and experimental results indicating the performance improvements, effectiveness and scalability offered by the proposed algorithms are presented and discussed in section 4. Concluding remarks are offered in section 5.

2. Related Work

The cryptographic schemes for key management in a hierarchy proposed by Akl and Taylor in 1983 [1] triggered a plethora of algorithms, each aimed at alleviating the problem of key replacement. The first, though efficient with a time complexity in $O(\log_2 n)$, was shown by its authors to be vulnerable to collusion attack. The second forestalls collusion attacks but creates keys whose storage requirements grow geometrically as the number of classes in the system increases, thus making the derivation of low level keys from high level keys computationally expensive. Indeed, the time complexity required for deriving low level keys from high level keys using this scheme is $O(n \log n)$, and generating and replacing keys throughout the hierarchy requires $O(n \log n)^n$ time [1]. Since ensuring data security requires replacing every key in the hierarchy whenever any groups' membership changes, this scheme is inefficient.

Mackinnon et al. propose an algorithm to assign keys optimally, using a heuristic to restrict the growth in key size to a linear rather than a geometric progression [1, 8]. However, in the worst case key derivation still requires $O(n \log n)$ time and the time requirements for key replacements remain unchanged.

Alternative solutions to the problem of minimizing the time and space requirements of key assignment (generation) include the approach based on bottom-up key generation proposed by Harn and Lin. Although the space requirements of the public parameters of the security classes is much smaller for higher level classes, when there are many security classes in the system storage space is in $O(n^2 \log n)$ as is the case in Sandhu's scheme[3, 4, 6, 10].

Tzeng proposed using time bounded keys to avoid replacing most of the keys each time a user is integrated into (e.g. subscriptions to newsletters where new users are not allowed to view previous data), or excluded from, the system [11]. His solution supposes that each class U_j has many class keys K_j^t , where K_j is the key of class U_j during time period t. A user from class U_j for time t_1 through t_2 is given an information item $I(j, t_1, t_2)$, such that, with $I(j, t_1, t_2)$, the key K_i^t of U_i at time t can be computed if and only if $U_i \preceq U_j$ and $t_1 \leq t \leq t_2$. The scheme is efficient in key storage and computation time because the same key can be used with different time bounds for multiple sessions. However, Chien showed that the scheme is vulnerable to collusion attack and proposed a solution based on a tamper resistant device that has also been shown to be vulnerable to collusion attack [2, 13]. Moreover, time bounded schemes are not practically efficient for dynamic scenarios where user behavior is difficult to foresee making it hard to accurately predict time bounds to associate with keys.

Other schemes include batching [7] and merging [12]. Batching provides better performance by accumulating key re-assignment requests until a threshold value of requests is attained. Here, better performance comes at the added price of an increase in the size of the vulnerability window (i.e. the period from the time the key server or security provider receives a key re-assignment request to the time all the users in the group concerned receive the new key [7]) between key replacements. Merging, creates greater correlation between group keys by compressing them into single keys and characterized with time-bounds. Although this results in better efficiency for key derivation in the average case, its reliance on the more sophisticated of the Akl-Taylor schemes still leaves the time complexity for key derivation in $O(n \log n)$ and that for computing the of number of time intervals z associated with a key, in $O(z^2)$.

Thus efficient key management requires a deterministic algorithm that ensures scalable growth in key size.

3. Key Management

Our key management model is inspired by that proposed by Akl and Taylor where by a central authority (e.g. a security provider or key server) creates a precedence tree graph based on a series of correlated keys. Each user group U_i is assigned a key $K_i^{t_i}$ such that t_i is the public exponent that defines U_i's relationship vis-a-vis other groups in the hierarchy. To access data both at their level and below users in U_i must possess a valid key $K_i^{t_i}$ that belongs either to a parent node or to the node itself. Since previous solutions faced either one or both pitfalls: *vulnerability to collusion attack or inefficiency in key management*, we use dispersion between assigned keys to reduce the possibility of collusion attack and an integer factorization metric to bound the growth in key size linearly [5].

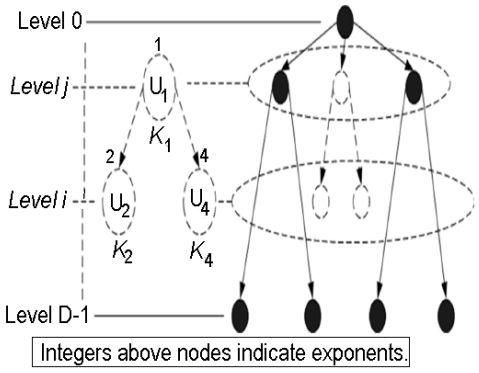

Figure 2. Collusion Avoidance

3.1. Exponent Generation Algorithm

Since the performance of any key assignment scheme, based on the Akl-Taylor model, depends on the size of t_i, our exponent generation algorithm uses the following properties to minimize the cost of key assignment and protect against collusion attack [5]:

- ***Property 1:*** Avoiding a collusion attack by verifying that the $gcd\{t_i : U_i \in G\}$ does not divide any t_j [1]. Here, G represents the graph of keys at a level in the hierarchy and t_j represents any of the exponents associated with keys higher up in the hierarchy
- ***Property 2:*** The growth function of t_i must be linear in order to minimize the cost of key assignment.

Any scheme that obeys the first property provides security against collusion attack [1, 8]. The second property bounds the growth of the exponents linearly to minimize the cost of key assignment and/or derivation. We use a regular rooted tree graph, where each node is of the same degree, to model the allocation of the exponents. In the tree graph, each node in the tree represents the exponent t_i associated with the corresponding key K_i. To guarantee access to its descendant nodes, the exponent belonging to the ancestor node must be a divisor of these exponents. This allows keys belonging to descendant nodes to be derived from those belonging to their ancestors. Thus, in addition to properties 1 and 2 cited above, we pose a third:

- ***Property 3:*** Accessibility is conditional on nodal precedence and exponent divisibility. Nodal precedence is verified by the condition that $t_i \div t_j$ and exponent divisibility by $t_i \div t_j$. Hence access to a child node is granted if and only if t_j belongs to a parent node and t_j is a divisor of the exponent t_i belonging to the child node.

Algorithm 1 : Exponent Generation (D, d)

Require: $D \geq 1;\ d \geq 1$
Ensure: $t_{0,0} \leftarrow x$ /*$x \geq 1$*: random value in* $[1..10]$**/*
1: **for** $k = 1$ to $D - 1$ **do**
2: $t_{k,0} \leftarrow t_{k-1,0} \times \left(d^k - 1\right)$
3: **if** $((t_{k,0} \leq t_{k-1,0}) \vee ((t_{k,0} \bmod t_{k-1,0}) \neq 0))$ **then**
4: $t_{k,0} \leftarrow t_{k-1,0} \times 2$ */*Get new value*/*
5: **end if**; $j \leftarrow 0$ */*Parent node position*/*
6: **for** $i = 1$ to $d^k - 1$ **do**
7: **if** $((i \bmod d) = 0)$ **then**
8: $j \leftarrow j + 1$ */*Shift to next parent node*/*
9: **end if**; $t_{k,i} \leftarrow t_{k,0} \times (i + 1)$
10: $temp \leftarrow i$ */*Bounds growth of* $t_{k,i}$ *linearly.*/*
11: **while** $((t_{k,i} = t_{k,i-1}) \vee ((t_{k,i} \bmod t_{k-1,j}) \neq 0))$ **do**
12: $t_{k,i} \leftarrow t_{k,0} \times temp$; $temp \leftarrow temp + 1$
13: **end while**
14: **end for**
15: **end for**

The central authority (CA) *e.g. key server*, selects a nodal degree, d, and depth, D and creates an exponent tree graph in which each node is labeled by the level at which it is situated and its position at that level. In Figure 2, for instance, $t_{j,0}$ indicates the exponent assigned to the node situated at level j and position 0. Integer values are allocated to each node such that the greatest common divisor (GCD) of the exponents at any level, say i (see Figure 2), does not divide any of the exponents belonging to their ancestors. A distance based heuristic is used to achieve this

by selecting an exponent for the leftmost node at any level such that it does not divide any of the exponents belonging to nodes at higher levels. Subsequent values assigned to the nodes at the same level are multiples of the value of the leftmost node. Thus the GCD of the exponents at any level, say i, is equal to the value belonging to the leftmost node and *Properties 1,2, and 3* are verified. The procedure is repeated until every node in the hierarchy has been assigned an exponent. The exponents allocated are then used to compute the keys required for each of the user groups. Algorithm 1 formally outlines the procedure.

Algorithm 2 : Exponent Replacement (rk, ri)

Require: $rk, ri \geq 0$ */*Position of t_i being replaced*/*
Ensure: $R \neq \emptyset$ */*Registry of exponents*/*

1: **for** $k = rk$ to $D - 1$ **do**
2: $j \leftarrow 0;\ i \leftarrow 1$
3: **while** $(i \leq ri)$ **do**
4: **if** $((i \bmod d) = 0)$ **then**
5: $j \leftarrow j + 1$ */*Parent node position*/*
6: **end if** $i \leftarrow i + 1$
7: **end while** $t_{k,ri} \leftarrow t_{k,0} \times (ri + 1)$
8: $temp \leftarrow ri$ */*Bounds growth of $t_{k,ri}$ linearly.*/*
9: **while** $((t_{k,ri} \in R) \vee ((t_{k,ri} \bmod t_{k-1,j}) \neq 0))$ **do**
10: $t_{k,ri} \leftarrow t_{k,0} \times temp$ */*Multiplicity rule*/*
11: $temp \leftarrow temp + 2$ */*Invalid Exponent,next?*/*
12: **end while**
13: **if** $\left(ri \neq d^{k-1} - 1\right)$ **then**
14: **for** $i = ri + 1$ to $d^k - 1$ **do**
15: **if** $((i \bmod d) = 0)$ **then**
16: $j \leftarrow j + 1$ */*Shift to next parent node*/*
17: **end if** $t_{k,i} \leftarrow t_{k,0} \times (i + 1)$
18: $temp \leftarrow i$
19: **while** $((t_{k,i} \in R) \vee ((t_{k,i} \bmod t_{k-1,j}) \neq 0))$ **do**
20: $t_{k,i} \leftarrow t_{k,0} \times temp;\ temp \leftarrow temp + 1$
21: **end while**
22: **end for**
23: **end if**
24: **end for**

3.2. Exponent Replacement

On reception of a message indicating a user's wish to depart from the system, the CA computes a new exponent for the node concerned and checks to ensure that it is not in the registry to prevent re-using exponents with the same key K_0 (see Section 1, equation 1). Next, the exponent is checked to ensure it is a multiple of the exponent belonging to the leftmost node and is a factor of the exponents belonging to its descendant nodes. If this is the case, the new exponent is recorded in the registry and assigned to the node. When no valid exponent can be found that satisfies the properties above, the CA will resort to selecting a new set of exponents for the whole hierarchy or changing the key K_0 and re-assigning keys to the whole hierarchy (this is the case in previous schemes). Algorithm 2 outlines the exponent replacement procedure.

3.3. Illustration

As illustrated in Figure 3, to generate the exponent tree graph, $t_{0,0}$ is randomly assigned a value of 1. At level 1, $t_{1,0} \leftarrow 1$ initially, but $t_{1,0} = t_{0,0}$ (see Algorithm 1, line 3) so $t_{1,0} \leftarrow 2$ (see Algorithm 1, line 4). Next $t_{1,1} \leftarrow t_{1,0} \times 2 = 4$ (see Algorithm 1, line 9). Since this value is valid and $t_{1,1}$ is the last node at level 1, we move on to level 2 where $t_{2,0} \leftarrow t_{1,0} \times \left(2^{2-1}\right)$ (see Algorithm 1, line 2:) and $t_{2,1} \leftarrow 12$. Initially, $t_{2,2} = 18$ (see Algorithm 1, line 12) but $t_{1,1} = 4$ is not a divisor of 18, so the next multiple of $t_{2,0}$, 24 (see Algorithm 1, line 16) is selected. Likewise, $t_{2,3} = 30$ is discarded in favor of 36.

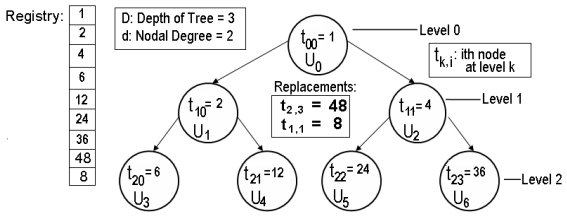

Figure 3. Exponent Generation and Allocation

Now, suppose two users decide to depart from U_6 and U_2 at times T, and $(T + 2)$ respectively. Assuming U_6's message arrives before U_2's, the CA selects a new exponent for $t_{2,3}$. As shown in Figure 3, since the first selection 42, is not evenly divisible by 4 the exponent belonging to $t_{1,1}$ (see Algorithm 2, line 8) the next available multiple of $t_{2,0} = 6$ which is 48 (see Algorithm 2, line 11) is selected. The new $t_{2,3} = 48$ is not in the registry and is divisible by its parent $t_{1,1} = 4$ so $t_{2,3} \leftarrow 48$, and is recorded in the registry. Similarly, 8 is selected to replace 4 and $t_{1,1} = 8$. When the exponent being replaced falls on a leftmost node, say $t_{2,0}$, we replace all the nodes at that level to avoid collusion. For example, replacing $t_{1,0}$ with 3 results in $t_{1,1}$ being replaced with 9, $t_{2,2}$ with 54 and $t_{2,3}$ with 54. Likewise, replacing the root node $t_{0,0}$ could result in a complete change of the hierarchy. We assume however, that in practical scenarios, replacements at the root node occur rarely and leftmost nodes can be used for user groups where membership changes occur rarely.

4. Analysis

This section presents an analysis of the algorithms proposed, in comparison to the Akl-Taylor scheme that is invulnerable to collusion. Since the proposed algorithms operate along the Akl-Taylor principle, we could also have used the Mackinnon scheme but choose not to because the worst case complexity bounds in the Mackinnon scheme tend, towards the Akl-Taylor scheme. The other schemes mentioned in section 2, either have complexity bounds for key generation and management that are similar to the Mackinnon scheme, are not adapted to highly dynamic scenarios (e.g. time-bounded schemes), and/or sacrifice security for performance [7, 10, 11, 12].

4.1. Complexity Analysis

Every user group stores a key K_i, that requires $\lceil log_d K_i \rceil$ space, where K_i is the largest key and d, the nodal degree of the tree. In an n node tree, the size of the largest t_i obtained by $t_i = \prod_{U_i \preceq U_j} p_j$, where p_j is a distinct prime assigned to a node U_j, is $O(n \log n)^n$ when the size of the n^{th} prime is in $O(n \log n)$[1]. Whereas, with the proposed exponent generation algorithm, the size of the largest t_i is $O(n \log n)$. Thus, in our scheme the size

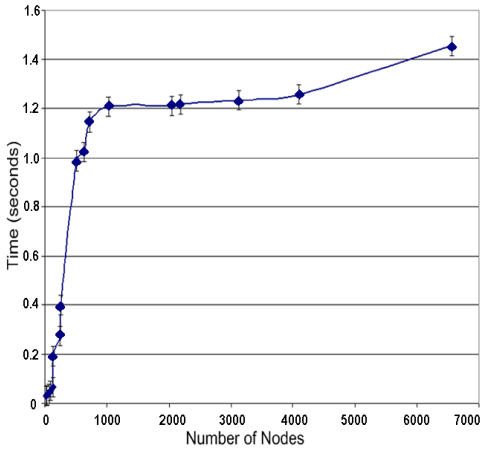

Figure 4. Computation Time vs. Number of Nodes Generated

of the exponents t_i, will grow linearly with the number of user groups n, whereas in the Akl-Taylor scheme it grows geometrically. To derive K_i from K_j the user in U_j needs $O(n \log n)$ time using the the Akl-Taylor scheme whereas our scheme requires $O(\log n)$ time. Finally, key replacement in the Akl-Taylor scheme requires generating keys for the whole hierarchy in the best, average, and worst cases. Although our replacement scheme still requires $O(n \log n)$ time in the worst case, we achieve better performance in the best and average cases, by replacing keys only in the portions of the hierarchy directly connected to the node that needs to be replaced.

4.2. Experimental Results

We evaluated the performance and scalability of the exponent generation and replacement algorithms, with experiments conducted on an IBM Pentium 4 computer with an Intel 2.66Ghz processor and 504MB of RAM. In the first experiment we studied the effect of varying tree sizes on exponent computation time, by using a series of different randomly generated trees comprised of 16 to 6561 nodes respectively. The exponent generation algorithm was executed 1000 times on each tree instance, and the computation times averaged and reported in Figure 4. The error bound for each point plotted is ± 0.04 seconds. We observed that the computation times grow logarithmically with the number of nodes in the hierarchy, which confirms the result obtained in the complexity analysis and noted that increased tree size (>1000 nodes) does not significantly affect system overhead.

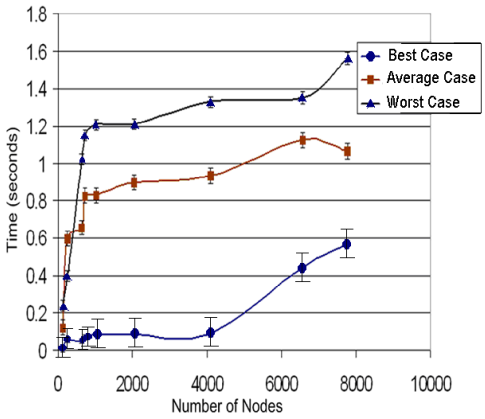

Figure 5. Cost of Key Replacement in Relation to Position and Percentage Replaced

The second experiment evaluates the cost of replacing exponents with respect to the tree size, number of nodes replaced, and nodal position. We randomly selected trees comprised of 128 to 7776 nodes respectively. In each tree 25%, 50%, and 75% of the nodes were replaced, in the best, average and worst cases respectively. In the best case, the nodes replaced were situated at the leaves or one level above the leaves. In the average case nodes were selected at random points in the tree, but not the root or leaf nodes.

The worst case consisted of selecting nodes situated to the left hand side of the tree including the root node. The replacement algorithm was executed 1000 times for each case on each tree and the results were averaged to obtain the plots in Figure 5. The error bounds for each of the points in the respective graphs plotted are, $\pm 0.013, \pm 0.04$ and ± 0.03 seconds in the best,average, and worst cases respectively.

The worst case of replacements is more computationally intensive than the best and average cases, but again the growth curve is logarithmic. In fact, for replacements of between 1000 and 6500 nodes the replacement times stay relatively constant. Thus, our exponent replacement algorithm is scalable to increased tree size and does not create significant overhead. The best case records a lower computation time than all the other cases because there are fewer comparisons and verifications required in replacing leaf nodes than there are in the average and worst cases.

5. Conclusions

In this paper we have presented two algorithms for cryptographic key management in a hierarchy based on a tree graph model. The first uses a distance based heuristic to control the growth of the exponents used to generate keys for access control in the tree hierarchy. While the second proposes a key replacement scheme that minimizes the number of keys that need to be changed whenever there is a change in user group membership by re-assigning keys only to portions of the graph rather than to the whole graph as is the case in previous schemes.

A complexity analysis and experimental results indicate that the algorithms are scalable and perform effectively. We note that verifying exponent validity can be computationally intensive, potentially creating overhead, and propose that this process be performed off line or during periods when the system is idle. Questions that may come to mind in relation to the proposed scheme include the possibility of key reuse. We answer this question in the negative, arguing that the registry of exponents and the exponent tree graph allow the system to keep track of the exponents that have been used or that are currently in use. In the extreme case when no valid exponent exists, the CA can select a new secret key K_0 and re-assign new keys to the whole hierarchy.

In closing, we mention a couple of open problems. Completely eliminating the window of vulnerability created between key replacements and determining an optimal complexity bound for handling out-of-sync data continue to remain an issue. The out-of-sync problem occurs when a valid user attempts to (between key replacements) decrypt and update a file encrypted with a key that is no longer valid or vice versa [7].

References

[1] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer Systems*, 1(3):239–248, August 1983.

[2] H.-Y. Chien. Efficient time-bound hierarchical key assignment scheme. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1301–1304, October 2004.

[3] L. Harn and H. Lin. A cryptographic keys generation scheme for multilevel data security. *Computer Security*, 9:539–546, 1990.

[4] Q. Huang and C. Shen. A new mls mandatory policy combining secrecy and integrity implemented in highly classified secure level os. *Proc. of* 7^{th} *International Conf. on Signal Processing (ICSP '04)*, 3:2409–2412, 2004.

[5] A. V. D. M. Kayem, S. G. Akl, and P. Martin. An independent set approach to solving the collaborative attack problem. *Proc. of* 17^{th} *IASTED International Conf. on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, Arizona*, pages 594–599, November 2005.

[6] C. Laih and T. Hwang. A branch oriented key management solution to dynamic access control in a hierarchy. *Proc. of 1991 IEEE Symposium on Applied Computing*, pages 422–429, April 1991.

[7] X. Li, Y. Yang, M. Gouda, and S. Lam. Batch rekeying for secure group communications. *WWW10*, 99(7):525–534, January 1999.

[8] 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 Transactions on Computers*, c-34(9):797–802, September 1985.

[9] R. Rivest, A. Shamir, and L. Adleman. A method of obtaining digital signatures and public key cryptosystems. *ACM Communications*, 21(2):120–126, 1978.

[10] R. Sandhu. Cryptographic implementation of tree hierarchy for access control. *Information Processing Letters*, 27:1–100, January 1988.

[11] W. G. Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):182–188, Jan. 2002.

[12] S.-Y. Wang and C.-S. Laih. Merging: An efficient solution for time-bound hierarchical key assignment scheme. *IEEE Transactions on Dependable and Secure Computing*, 3(1):91–100, Jan. 2006.

[13] X. Yi and Y. Ye. Security of tzeng's time-bound key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1054–1055, July 2003.

[14] S. Zhu, S. Setia, and S. Jajodia. Performance optimizations for group key management schemes. *Proc. of* 23^{rd} *International Conference on Distributed Computing Systems (ICDCS '03)*, pages 163–171, 2003.