CERIAS Tech Report 2004-49

HASH-BASED ACCESS CONTROL IN AN ARBITRARY HIERARCHY

by Keith Frikken, Mikhail Atallah, and Marina Bykova

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Hash-Based Access Control in an Arbitrary Hierarchy*

Keith Frikken **Mikhail Atallah** **Marina Bykova**
CERIAS and Department of Computer Sciences
Purdue University
{kbf,mja,mbykova}@cs.purdue.edu

November 5, 2004

Abstract

We give the first solution to the problem of access control in an arbitrary n-node hierarchy G (e.g., RBAC) where all of the following hold: (i) only hash functions are used for a node to derive a descendant's key from its own key, as opposed to the use of RSA public-key cryptography in many previous schemes (which requires slow modular exponentiations); (ii) the space complexity of the public information is the same as that of storing graph G (which is asymptotically optimal), as opposed to the quadratic space complexity of some other schemes; (iii) the derivation by a node of a descendant's access key takes $O(n)$ bit operations in the worst case, as opposed to $O(n^2)$ bit operations in some of the previous schemes; (iv) updates are handled locally in the hierarchy and do not "propagate" to descendants or ancestors of the affected part of the tree; and (v) the scheme is resistant to collusion in that no subset of nodes can conspire to gain access to any node that is not already a descendant of one of the conspirators (hence legally accessible). Similar to a number of previous schemes, the private information at a node consists of a single key associated with that node. The security of our scheme relies on the existence of cryptographic one-way hash functions and the random oracle model. Another (more minor) property of our scheme is that it does not require access graph G to be free of directed cycles. We provide simple modifications to our scheme so it can handle Crampton's extensions of the standard hierarchies to "limited depth" and reverse inheritance [6].

1 Introduction

The problem of access control in a hierarchy is not new: it has been addressed in the prior literature and became even more important after the development of Role-Based Access Control (RBAC) models (i.e., see [7, 14] and all subsequent work). In this work we provide a new solution to this problem which has advantages, over the previous work, that are subsequently listed in this section.

First of all, we consider an arbitrary hierarchy of access levels that can be modeled as a directed graph. To the best of our knowledge, this is the first scheme that works for arbitrary access graphs — even those that may contain cycles. When describing our scheme, we use a directed acyclic graph (DAG) to model access graphs because normally access graphs are not expected to have cycles, but the use of our scheme is not limited to DAGs. Throughout this paper we may also use RBAC

*Portions of this work were supported by Grants IIS-0325345, IIS-0219560, IIS-0312357, and IIS-0242421 from the National Science Foundation, Contract N00014-02-1-0364 from the Office of Naval Research, by sponsors of the Center for Education and Research in Information Assurance and Security, and by Purdue Discovery Park's e-enterprise Center.

as the most prominent example of our approach, even though the scheme is not limited to RBAC models alone.

Another vital aspect of access control schemes is computational and storage space constraints required for key management and processing. It is clear that low computation and space requirements allow a scheme to be used in a much wider spectrum of devices and applications than costly access control schemes. For example, not only inexpensive smartcards, but also small battery-operated sensors, embedded processors, and other kinds of computationally weak and memory-limited devices can now carry out the required computations. Thus in order to make our scheme acceptable for use with such weak clients, we do not use powerful cryptography but instead utilize only cryptographic hashes. Throughout this paper, we use the word "smartcard" as a convenient shorthand to refer to any such types of weak clients.

The scheme we propose is also optimal in terms of its storage space complexity. There is only one key needed per access level (vertex) in the graph, to be stored at the client side. The server stores information linear in the size of the representation of the access graph, which is asymptotically optimal. Note that in order for any access scheme to work, the structure of the access graph must be known (i.e., when a user that has privileges for access level a_i is requesting an object stored at level a_j, the user should know the structure of the graph to determine whether and how his key can be used to create a key for a_j). In our scheme, along with storing the required structure, every node and edge contains additional constant space data that is used for key generation.

Our scheme also achieves full resilience against collusion, meaning that no subset of nodes in the access graph G can conspire to gain access to an object at a level to which they do not already have access. An adversary could obtain such a subset of nodes if: i) it compromises more than one smartcard or ii) it compromises a smartcard with multiple access levels. Even though the scheme was mainly designed to be used with tamper-resistant smartcards, a number of prior publications (see, e.g., [2, 3]) suggest that in some cases an attacker can obtain access to the contents of such tamper-resistant cards, which adds a lot of value to collusion-resilient solutions. In addition, the collusion-resilience of our scheme allows this approach to be used with other devices that do not have tamper-resistance.

Our scheme supports dynamic changes such as insertion or deletion of a node or an edge, which also allow nodes to be moved within the hierarchy. The scheme can also be easily extended to cover access models that go beyond the traditional inheritance of privilege. More precisely, we give extensions that enable normal as well as reverse inheritance in the graph (i.e., access to objects down or up in the hierarchy) and also allow for fixed-depth inheritance. Such extensions are useful not only in the context of other standard models such as Bell-LaPadula [4], but can also apply to RBAC (e.g., reverse limited-depth inheritance permits an employee to have access to documents stored at the level of the department of that employee). There extensions are modeled after Crampton's work [6] and do not increase the space or computational complexity of our scheme.

The rest of this paper is organized as follows: Section 2 provides an overview of related literature and of our contributions. In section 3, we give a formal description of the problem and list our notation. Section 4 presents our base model along with its security proofs. In sections 5 and 6 we give description of dynamic versions of the model. Section 7 describes extensions to our scheme that permit its usage with other access models given in [6]. Finally, section 8 concludes the paper.

2 Related Work/Our Contributions

2.1 Related literature

Many previous schemes [1, 8, 9, 10, 11, 12, 15, 16] (as do ours) rely on the idea that an ancestor can store only one key as long as there is a way in which it can derive the key at any one of its descendants. We briefly describe some of the schemes below. In what follows n is the number of nodes in G (it is *not* an RSA modulus – which we henceforth denote by pq rather than by n).

Most of the previous schemes that handle arbitrary hierarchies (i.e., are not restricted to trees) [1, 11, 10, 9] not only use expensive crypto operations (modular exponentiations required by their reliance on RSA) but, in addition, suffer from other inefficiencies that have a single root cause: The fact that the schemes require each node x of G to contain (and dynamically maintain) a public integer I_x that can grow to be $O(n)$ bits long. This huge number I_x is intended for use in a division followed by an exponentiation that are done when x derives, in what these papers call "direct access", the key k_y of a node y that is a descendant of x in G. The way the I_x numbers grow so huge is because each I_x is the product (not modular – the full, unreduced product) of a set P_x of secret primes associated with x, where P_x is the *union* of all the P_y's such that y is an immediate successor of x in G (these are different primes from the p and q of the RSA system they use). The use of such $O(n)$-bit integers in these schemes has many bad consequences: It implies that the storage space for the public information stored along with G has a worst-case complexity of $O(n^2)$ bits; it implies that "direct access" is more expensive than the "one operation" that is claimed in these papers: The division of two $O(n)$-bit integers takes $O(n^2)$ bit operations, resulting in an $O(n)$ bit integer that is used as the exponent in a modulo pq exponentiation (which takes time $O(n \log(pq))$ bit operations because the exponent cannot be reduced modulo $(p-1)(q-1)$ because p and q are not known to a node, although this cost is dominated by the above-mentioned $O(n^2)$ bit operations for doing the division). In addition to the above-mentioned issues, these previous techniques require very large constants that hide behind the "O" notation used an asymptotic analysis. The fact that P_x is the union of the P_y's of its successors is also what caused poor performance in the face of updates: A change (e.g., an insertion with a new prime) had to be reflected in every node on the path to the root, with changes occurring in both the private and public information of those nodes on the path. In our case the changes will be "contained" locally to where the modification (insertion or deletion) took place. Some of these previous schemes suffer from many other drawbacks not directly related to the above discussion: For example, the key generation in [12] depends (in the variable called c of the default key generation routine) on the number of paths in G between ancestor-descendant pairs, which can be exponential in n.

Papers that use only hashing (and no expensive crypto such as modular exponentiation) either restricted themselves to trees [15], or, in the case of [8], made a restrictive assumption that bounded n to be a polynomial of the number of public bits stored at a node and were also unable to locally "contain" the changes caused by insertions/deletions in G (i.e., a local change to G had long-range effects elsewhere in G).

2.2 Sandhu's Method

Because our method is an outgrowth of Sandhu's [13, 15], we briefly review Sandhu's approach. Sandhu's method is simple and elegant, and uses only one-way hash functions. It proceeds from the root down to the leaves, repeatedly deriving a child's key by combining its own name with its parent's key (via a one-way hash of them). Specifically:

1. The root gets assigned a random key.

2. A child y of a node x, that has already been assigned a key k_x, gets assigned as key $k_y = H(y, k_x)$ where H is a one-way hash function.

The method is limited to trees, but it is more efficient than subsequent methods at generating a descendant's key from an ancestor's key because it follows a path from the latter to the former, doing an inexpensive hash each hop along the way (and the number of hops is bounded by $O(n)$). The previous subsection gave more details on how access keys are derived in a number of previous RSA-based schemes.

2.3 Our Contributions

Our contribution is to achieve simultaneously better storage performance and better access performance (see the itemized list below); our measure of access performance is the number of bit operations needed to derive a descendant's key, which we view as the correct measure of performance (i.e., when the operands are n-bit numbers, it is erroneous to count – as some papers in the literature do – the number of arithmetic operations as the performance metric).

More specifically, our solution has the following characteristics:

- Only hash functions are used for a node to derive a descendant's key from its own key (as opposed to the use of RSA public-key cryptography in a number of previous schemes). This implies that the security of our scheme relies only on the existence of cryptographic one-way hash functions along with the commonly assumed Random Oracle Model [5].
- The space complexity of the public information is the same as that of storing G and is asymptotically optimal, as opposed to the quadratic space complexity of many previous schemes (space linear in G was previously known only for the special case when G is a tree [15]).
- The derivation by a node of a descendant node's access key requires $O(n)$ bit operations in the worst case (as opposed to $O(n^2)$ bit operations in some of the previous schemes).
- Updates are handled locally in the hierarchy and do not "propagate" to descendants or ancestors of the affected part of the tree. In the "top down" previous schemes such as [15, 8] all the descendants of the modified node had to be updated, and in the "bottom up" schemes such as [1, 11, 9, 10, 12] a change at a node x had to be reflected in every node on the path from x to the root, with changes occurring in both the private and public information of those nodes on the path.
- Our scheme is resistant to collusion in that no subset of nodes can conspire to gain access to any node that is not already a descendant of one of the conspirators (hence legally accessible).
- The private information at a node consists of a single key associated with that node.
- Our scheme does not require G to be free of directed cycles, whereas previous schemes required acyclicity. Although we do not see immediate applications to allowing cycles, because RBACs and most other practical situations involve an acyclic G, it is intriguing that none of the previous schemes could handle cycles, as they required as "starting points" nodes of zero in-degree (for the "top down" schemes such as [15, 8]) or nodes of out-degree zero (for the "bottom up" schemes such as [1, 11, 9, 10, 12]).
- We provide simple modifications to our scheme so it can handle Crampton's extensions of the standard hierarchies to "limited depth" and reverse inheritance [6]

3 Problem Definition/Notation

3.1 Problem Definition

There is a directed access graph $G = (V, E, O)$ s.t. V is a set of vertices $V = \{v_1, \ldots, v_n\}$ of cardinality $|V| = n$, E is a set of edges $E = \{e_1, \ldots, e_m\}$ of cardinality $|E| = m$, and O is a set of objects $O = \{o_1, \ldots, o_k\}$ of cardinality $|O| = k$. Each vertex v_i represents an access level in the access hierarchy (e.g., G could be an RBAC graph of roles). Each access level v_i has a set of objects associated with it. Function $\mathcal{O} : V \rightarrow 2^O$ maps an access level to a unique set of objects such that $|\mathcal{O}(v_i)| \geq 0$ and $\forall i \, \forall j$, $\mathcal{O}(v_i) \cap \mathcal{O}(v_j) = \emptyset$ if $i \neq j$. (For the sake of brevity of exposition we use notation $\mathcal{O}_i$ to mean $\mathcal{O}(v_i)$.)

If there is a directed path in G from node v_i to node v_j, node v_i is a ancestor of node v_j, i.e., $v_i \in Anc(v_j)$. In the access hierarchy, a path from node v_i to node v_j means that any subject that can assume access rights at access level v_i is also permitted to access any object at access level v_j such that $o \in \mathcal{O}_j$. The function $\mathcal{O}^* : V \rightarrow 2^O$ (we use $\mathcal{O}_i^*$ as a shorthand for $\mathcal{O}^*(v_i)$) maps an access level $v_i \in V$ to a set of objects accessible to a subject at access level v_i; the function is recursively defined as $\mathcal{O}_i^* = (\bigcup_{v_i \in Anc(v_j)} \mathcal{O}_j^*) \cup \mathcal{O}_i$. When the set of edges E is not essential to our current discussion, we may omit it from the definition of the graph and use notation $G = (V, \ O)$ instead.

Our goal is to define a key allocation mechanism that implements such an access graph, that is, an assignment of keys to users and objects where a user can access an object iff he has a key for the object. It is desirable to minimize the number of keys required to be held for an access level and to minimize the number of keys assigned to an object. We now formally define the key allocation policy.

Definition 1 *Suppose we are given a key-space $\mathcal{K}$. A key allocation, $KA : V \cup O \rightarrow 2^{\mathcal{K}}$, maps objects and access levels to a subset of keys.*

One of our techniques for improving the efficiency of our schemes is to compress an ancestor's key space by having its' descendants keys be computable from one of their keys via a one-way function.

Definition 2 *Given two keys k and k', we say k* generates *k', denoted by $k \overset{G}{\Rightarrow} k'$, iff there exists a polynomial-time algorithm D such that $D(k) = k'$. When there does not exist a Probabilistic Polynomial Time algorithm that outputs k' when given k with more than a negligible probability, we say $k \overset{G}{\not\Rightarrow} k'$. Sometimes we allow these algorithms to have auxiliary information.*

We are now ready to formally define what is meant by "implementing" an access control policy.

Definition 3 *We say that a key allocation KA implements an access DAG $G = (V, O)$ iff the following two conditions are true:*

1. *Completeness: $\forall (v_i, o_j) \in V \times \mathcal{O}_i^*, \exists (k, k') \in KA(v_i) \times KA(o_j)$ s.t. $k \overset{G}{\Rightarrow} k'$. In other words, for every object that an access level has rights to access, that access level should be assigned a key that can generate a key that is used to encrypt the object.*

2. *Soundness: $\forall (v_i, o_j) \in V \times O \setminus \mathcal{O}_i^*, \forall (k, k') \in KA(v_i) \times KA(o_j), k \overset{G}{\not\Rightarrow} k'$. In other words, for every object that an access level does not have rights to access, there is no key in that access level key space that can be used to generate any of the keys used to encrypt the object.*

Definition 4 *We say that a key allocation KA that implements an access DAG $G = (V, O)$ is fully collusion-resilient (or just collusion-resilient) iff for any set of attackers with access to levels $V' = v_{i_1}, \ldots, v_{i_r}$, where $V' \subset V$ and $1 < r < n$, we have that $\forall (v_{i_j}, o_\ell) \in V' \times O \setminus \bigcup_{v_{i_j} \in V'} \mathcal{O}^*_{i_j}, \forall (k, k') \in KA(v_{i_j}) \times KA(o_\ell), k \overset{G}{\not\Rightarrow} k'$. In other words, collusion does not allow the coalition V' to produce decryption keys for objects to which they did not already have access.*

3.2 Notations

We now list some of our notations:

1. Given a directed graph $G = (V, E)$, we define an ancestry function $Anc(x, G)$ which is a set such that $y \in Anc(x, G)$ if there is a path from y to x in the graph G. In terms of access control, $y \in Anc(x, G)$ iff y inherits x's attributes.

2. In a directed graph $G = (V, E)$ we also define the set of descendants of node x as $Desc(x, G)$, where $y \in Desc(x, G)$ if there is a path from x to y in G.

3. For a directed graph $G = (V, E)$, we use a function $Pred(x, G)$ to denote the set of immediate predecessors of x in G, i.e., if $y \in Pred(x, G)$ then there is a directed edge from y to x in G. Similarly, we define $Succ(x, G)$ to be the set of immediate successors of x in G.

4. When it is clear what graph we are discussing, we omit G from the notation and use instead the shorthand notation $Anc(x)$, $Desc(x)$, $Succ(x)$, and $Pred(x)$.

5. We consider a node to be its own ancestor and descendant, but we do not consider it to be a predecessor or successor of itself.

4 Base Scheme

This section describes a scheme in which every node has one key associated with it, the public information is linear (rather than quadratic) in the size of the access DAG G, and a computation by node v of a key that is ℓ levels below it can be done in $O(\ell)$ evaluations of a hash function. In this section we focus on key allocations for a static access hierarchy. We describe our scheme in Section 4.1 and then prove its security properties in Section 4.2. Extensions of this base scheme are given in sections 5, 6, and 7.

4.1 Scheme

In this section we describe the static version of the scheme that does not include insertions or deletions. We assume that we are given a cryptographic hash function $H : \{0,1\}^\star \to \{0,1\}^\rho$. Below we provide a description of the private key generation process and the nature of public information stored at each node of the graph.

Private key Each vertex v_i is assigned a random private key k_i in $\{0,1\}^\rho$. An entity that is assigned access levels $V' \subseteq V$ is given a smartcard with all keys for their access levels $v_j \in V'$.

Public information For each vertex v_i there is a unique label ℓ_i in $\{0,1\}^\rho$ that is assigned to the vertex. Also for each edge (v_i, v_j), the value $y_{i,j} = k_j - H(k_i, \ell_j) \bmod \rho$ is stored publicly for this edge.

Example: We give an example of how the key allocation mechanism assigns keys to a graph that is more complicated than a tree. Our sample graph is depicted in Figure 1, and we show two examples regarding this figure. First, it is possible for the node with k_1 to generate key k_2. This is because that node can compute $H(k_1, \ell_2)$ and use it, along with the public edge information, to obtain k_2. Similarly, the node with k_3 cannot generate k_2, since this would require inversion of the H function.

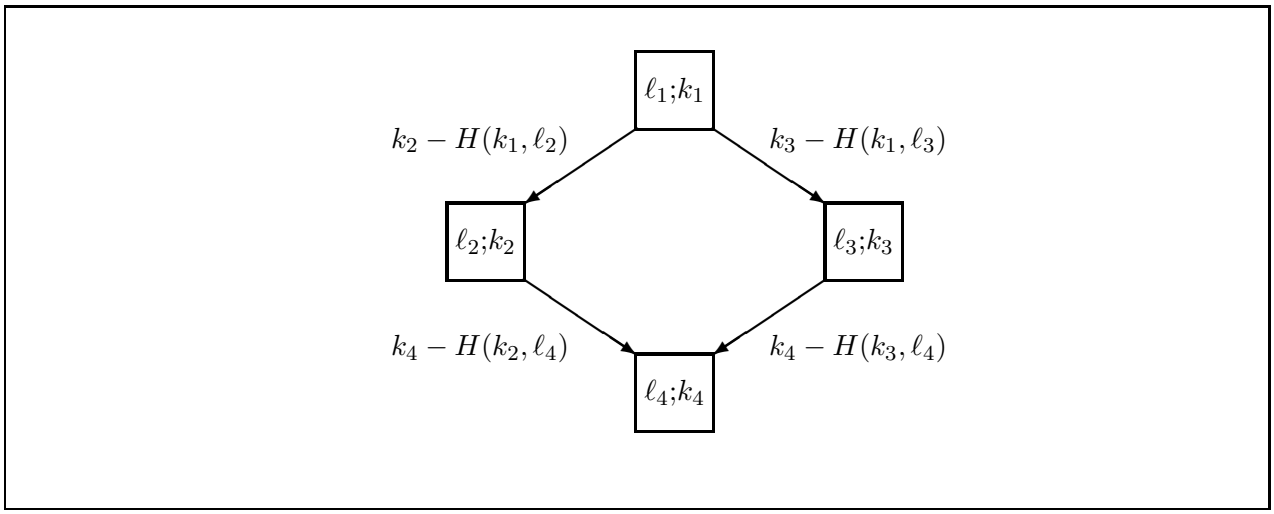

Figure 1: Allocation for example access graph; all arithmetic is modulo ρ

4.2 Proof of Security

For the proof of our scheme we assume that H is a random oracle.

Theorem 1 *The scheme described in Section 4.1 is complete.*

Proof Suppose that a card has key k_i corresponding to access level $v_i \in V$. Also assume that access to object o_j at access level v_j (i.e., $o_j \in \mathcal{O}(v_j)$) is requested such that $o_j \in \mathcal{O}_i^*$. This means that access to o_j can be obtained using key k_j and also there is a directed path in G from node v_i to v_j. Then in order for v_i to generate k_j, v_i sequentially processes every edge (v_x, v_y) on the path between v_i and v_j.

Given an edge (v_x, v_y) for which both v_x's private key k_x and the stored public information is known, v_i can generate v_y's private information k_y (as the values ℓ_y and $y_{x,y} = k_y - H(k_x, \ell_y) \bmod \rho$ are in the public information, it is trivial to compute k_j). Due to the sequential nature of key generation on the path between v_i and v_j, v_i will be able to derive keys of all necessary nodes and produce key k_j that will enable access to o_j. □

Theorem 2 *The scheme described in Section 4.1 is sound, even in the presence of collusion.*

Proof Suppose this is not the case, i.e., that there is a an adversary with vertex set $A \subset V$ that can obtain access to at least one object $o_j \in O \setminus \bigcup_{v_{i_j} \in A} \mathcal{O}_{i_j}^*$. This means that, if $o_j \in \mathcal{O}_j$, then the adversary can generate key k_j and there is no direct path from any node in A to v_j, i.e., $Anc(v_j) \cap A = \emptyset$. Let w be a vertex such that (i) $o_j \in \mathcal{O}(w)$ and (ii) this property does not hold for any of its ancestors, i.e., there is no object above w in the hierarchy to which the adversary can gain unauthorized access. By the well ordering principle such a vertex w must exist. There are two cases to consider: 1) w has no ancestors and 2) w has at least one ancestor.

1. In this case the only public information about k_w is the edges from w. Suppose $Succ(w) = \{v_{s_1}, \ldots, v_{s_m}\}$, then the public information will be $k_{s_i} - H(k_w, \ell_{s_i}) \bmod \rho$, and in the worst case the adversary could create a set of values of the form $H(k_w, \ell_{s_i})$. However, since H is a random oracle, this reveals no information about k_w.

2. In this case the public information that k_w is the edges to and from w. Recall that w has no ancestors for which the adversary knows the key. Suppose $Pred(w) = \{v_{p_1}, \ldots, v_{p_n}\}$ and $Succ(w) = \{v_{s_1}, \ldots, v_{s_m}\}$. The only public information about k_w is of the form $k_{s_i} - H(k_w, \ell_{s_i}) \bmod \rho$ or of the form $k_w - H(k_{p_i}, \ell_w) \bmod \rho$. From Case (1) it is clear that the first set of values is of no help to an adversary, however learning k_w from the second group of values requires that the adversary know k_{p_i}, which is a contradiction to our assumption that w has no ancestors for which the system is not sound.

In either case, the adversary cannot learn a node's secret values unless it contains an access level that is an ancestor of that vertex. This implies that the adversary cannot obtain the key to gain access any unauthorized object $o_j \in O \setminus \bigcup_{v_{i_j} \in A} \mathcal{O}^*_{i_j}$. □

5 Extension: Preliminary Dynamic Version

In this section we describe the way dynamic changes are handled in our base scheme. The types of changes we consider are: insertion and deletion of an edge and insertion and deletion of a node. Deletion of an edge is the only change that involves recarding (node deletion will result in recarding as well, if it requires edge removal); more precisely, in this preliminary solution, there is recarding of descendants of the node from which the edge is removed. We eliminate the problem of recarding in the next section.

5.1 Insertion of an Edge

Suppose the edge (v_i, v_j) is inserted into G. Then we simply add $y_{i,j} = k_j - H(k_i, \ell_j) \bmod \rho$ to the description of G by attaching it to the edge (v_i, v_j).

5.2 Deletion of an Edge

Suppose the edge (v_i, v_j) is deleted from G. Then the following updates are done. For each node $v_t \in Desc(v_j, G)$, perform:

1. Replace key k_t with a new arbitrary key k'_t.

2. For each edge (v_p, v_t) where $v_p \in Pred(v_t)$, update the value of $y_{p,t}$ according to the new key k_t.

5.3 Insertion of a new node

If a new node u is inserted, together with new edges into it and new edges out of it, then we consider this as happening in two stages:

- First, the creation of the node u without any edges touching it, which is trivial to do since all it requires is generation of a random key k_u for that node.

- Second, we add the edges one by one, using each time the procedure outlined above for edge-insertions.

5.4 Deletion of a node

If a node v_i is deleted, together with all the edges that touch it, then we consider this as happening in two stages:

- First, we delete the edges touching v_i one by one, using each time the procedure outlined above for edge-deletions.
- Second, now that v_i has no edges touching it, removing it is trivial.

6 Extension: Better Dynamic Version, without Recarding

A downside to the previous scheme for dynamic changes to the access hierarchy is that the scheme requires some access levels to be recarded (i.e., obtain new cards). In this section we introduce a variation on the previous scheme that will allow up to t removals (these can be batched into groups of removals) to the access hierarchy without having to recard any access level. After t removals have been done, then the system will need to recard the users. However, t can be chosen such that this even is unlikely to happen before the devices expire.

6.1 Modifications to our scheme

Our new scheme is as follows:

Private key Each vertex v_i is assigned a random private key k_i in $\{0,1\}^\rho$. An entity that is assigned access levels $V' \subseteq V$ is given a smartcard with all keys for their access levels $v_j \in V'$.

Public information For each vertex v_i there is a unique label ℓ_i in $\{0,1\}^\rho$ that is assigned to the vertex. There is a revocation number P that is initialized to t. It is worth noting that the key used to send things to an access level v_i is $H^P(k_i)$ (i.e., H applied P times to the value k_i) Also for each edge (v_i, v_j), the value $y_{i,j} = H^P(k_j) - H(k_i, \ell_j) \bmod \rho$ is stored publicly for that edge.

Note that an alternative construction it to assign each node its own P value. In such a case we would be able to allow t removals that effect each node. However, for simplicity we describe the scheme with a single P value.

6.2 Insertion of an Edge

Suppose the edge (v_i, v_j) is inserted into G. Then we simply add $y_{i,j} = H^P(k_j) - H(k_i, \ell_j) \bmod \rho$ to the description of G by attaching it to the edge (v_i, v_j).

6.3 Deletion of an Edge

Suppose the edge (v_i, v_j) is deleted from G. Then the following updates are done:

1. If $P = 0$, then the system must recard, otherwise proceed as follows:
2. Remove the edge (v_i, v_j).
3. For each edge $(v_x, v_y) \in E$, change $y_{x,y} = H^{P-1}(k_y) - H(k_x, \ell_y) \bmod \rho$.
4. Decrease P by 1.

6.4 Insertion of a new node

This is the same as node insertion that was described for the previous protocol.

6.5 Deletion of a node

If a node v_i is deleted, together with all the edges that touch it, then we consider this as happening in two stages:

- First, we delete the edges touching v_i in one large batch as this requires us to only reduce P by 1.
- Second, now that v_i has no edges touching it, removing it is trivial.

7 Extension: Other Access Models

Traditionally, the standard notion of permission inheritance in access control is that permissions are transfered "up" the access graph G. In other words, any vertex in $Anc(v_i, G)$ has a superset of the permissions held by v_i. Crampton [6] suggested other access models, including:

1. Permissions that are transfered down the access graph. For these permissions, any node in $Desc(v_i, G)$ has a superset of the permissions held by v_i.
2. Permissions that are transfered either up or down the access graph but only to a limited depth.

In this section, we discuss how to extend our scheme to allow such permissions. We can achieve upward and downward inheritance with only two keys per node. Also, we can achieve all of these permissions with four keys at each card for a special class of access graphs that are "layered" DAGS (we define this later) when there is not collusion.

7.1 Downward inheritance

To handle such queries, we construct the reverse of the graph $G = (V, E, O)$, which is a graph $G^R = (V, E', O)$ where for each edge $(v_i, v_j) \in E$ there is an edge $(v_j, v_i) \in E'$. Then we use our base scheme for both G and G^R, which results in each node having two keys, but the scheme now supports permissions that are inherited upwards and downwards.

7.2 Limited depth permission inheritance

We say that an access graph is *layered* if the nodes can be partitioned into a group of sets, denoted by $S_1, S_2, \ldots, S_r$ where for all edges (v_i, v_j) in the access graph it holds that if $v_i \in S_m$ then $v_j \in S_{m+1}$. We claim that many interesting access graphs are either layered, but any DAG can be made layered by adding enough virtual nodes.

Given such a layering, we can then support limited depth permissions. This is done by creating another graph which is a linear list that has a node for each layer, and an edge has an edge from each layer to the next layer. The reverse of this graph is also constructed, and these graphs are assigned keys according to our scheme. A node is given the keys corresponding to its layers. Clearly, with such a technique we can support permission requirements of the form all nodes higher than some level and of the form all nodes lower than some level.

We now show how to utilize these four key assignments to support permission sets of the form "all ancestors of some node v_i that are lower than a specific layer L" (an analogous technique can be used for permission sets of the form "all descendants of v_i above some specific layer"). Suppose the key for the permission requirement "all ancestors of node v_i" is k_i and the key for permission requirement "all nodes lower than layer L" is k_L. Then we establish a key for both permission requirements by setting the key to $H(k_i, k_L)$. Clearly, only nodes that are an ancestor of v_i can generate k_i and only nodes lower than level L can generate k_L, so the only nodes that could generate both keys would be an ancestor of k_i AND below level L, assuming that there is no collusion.

8 Conclusions and Future Work

In summary, we give the first solution to the problem of access control in an arbitrary hierarchy G where all of the following hold:

(i) only hash functions are used for a node to derive a descendant's key from its own key;

(ii) the space complexity of the public information is the same as that of storing graph G;

(iii) the derivation by a node of a descendant's access key requires $O(n)$ bit operations in the worst case;

(iv) updates are handled locally in the hierarchy and do not "propagate" to descendants or ancestors of the affected part of the tree;

(v) the scheme is resistant to collusion in that no subset of nodes can conspire to gain access to any node to which they do not have permission to have access.

We do all of the above while requiring the private information at a node to consist of a single key associated with that node. Finally, we provide simple modifications to our scheme that allow to handle Crampton's extensions of the standard hierarchies to "limited depth" and reverse inheritance [6].

Future directions of this work include:

1. Extend our scheme to support temporal constraints.
2. Extend our scheme to support "limited depth" permission inheritance in access graphs that are not layered without adding virtual nodes and in a collusion resilient manner.
3. Investigate time-space tradeoffs: Decreasing the number of bit operations needed to access a descendant's key, at the expense of some increase in the storage space of the public information.

References

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

[2] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *USENIX Workshop on Electronic Commerce*, pages 1–11, November 1996.

[3] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *Security Protocols Workshop*, volume 1361 of *LNCS*, pages 125–136, April 1997.

[4] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR–2547, MITRE Corporation, March 1973.

[5] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM Press, 1993.

[6] J. Crampton. On permissions, inheritance and role hierarchies. In *ACM Conference on Computer and Communications Security (CCS)*, pages 85–92, October 2003.

[7] D. Ferraiolo and D. Kuhn. Role based access control. In *National Computer Security Conference*, 1992.

[8] T. Hardjono, Y. Zheng, and J. Seberry. New solutions to the problem of access control in a hierarchy. Technical report, 1993.

[9] M. Hwang. A new dynamic key generation scheme for access control in a hierarchy. *Nordic Journal of Computing*, 6(4):363–371, Winter 1999.

[10] M. Hwang and W. Yang. Controlling access in large partially ordered hierarchies using cryptographic keys. *Journal of Systems and Software*, 67(2):99–107, August 2003.

[11] S. MacKinnon, P. Taylor, H. Meijer, and S. Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Transactions on Computers*, 34(9):797–802, September 1985.

[12] I. Ray, I. Ray, and N. Narasimhamurthi. A cryptographic solution to implement access control in a hierarchy and more. In *ACM Symposium on Access Control Models and Technologies*, June 2002.

[13] R. Sandhu. On some cryptographic solutions for access control in a tree hierarchy. In *Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 405–410, December 1987.

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

[15] R.S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Information Processing Letters*, 27(2):95–98, January 1988.

[16] W. 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, 2002.