**CERIAS Tech Report 2006-02**

# DYNAMIC AND EFFICIENT KEY MANAGEMENT FOR ACCESS HIERARCHIES

by M. Atallah, K. Frikken, and M. Blanton

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

# Dynamic and Efficient Key Management for Access Hierarchies*

Mikhail J. Atallah, Keith B. Frikken, and Marina Blanton
Department of Computer Science
Purdue University
{mja,kbf,mbykova}@cs.purdue.edu

## ABSTRACT

The problem of key management in an access hierarchy has elicited much interest in the literature. The hierarchy is modeled as a set of partially ordered classes (represented as a directed graph), and a user who obtains access (i.e., a key) to a certain class can also obtain access to all descendant classes of her class through key derivation. Our solution to the above problem has the following properties: (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 the hierarchy; (iii) the private information at a class consists of a single key associated with that class; (iv) updates (revocations, additions, etc.) are handled *locally* in the hierarchy; (v) the scheme is provably secure against collusion; and (vi) key derivation by a node of its descendant's key is bounded by the number of bit operations linear in the length of the path between the nodes. Whereas many previous schemes had some of these properties, ours is the first that satisfies all of them. Moreover, for trees (and other "recursively decomposable" hierarchies), we are the first to achieve a worst- and average-case number of bit operations for key derivation that is exponentially better than the depth of a balanced hierarchy (double-exponentially better if the hierarchy is unbalanced, i.e., "tall and skinny"); this is achieved with only a constant increase in the space for the hierarchy. We also show how with simple modifications our scheme can handle extensions proposed by Crampton of the standard hierarchies to "limited depth" and reverse inheritance [13]. The security of our scheme relies only on the use of pseudo-random functions.

*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.


*CCS'05,* November 7–11, 2005, Alexandria, Virginia, USA.


## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; E.1 [**Data Structures**]: [Graphs and networks]

## General Terms

Security, Design, Algorithms.

## Keywords

Hierarchical access control, key management, efficient key derivation.

## 1. INTRODUCTION

**Background.** In this work, we address the problem of access control and, more specifically, the key management problem in an access hierarchy. Informally, the general model is that there is a set of access classes ordered using partial order. We use a directed graph $G$, where nodes correspond to classes and edges indicate their ordering, to represent such a hierarchy. Then a user who is entitled to have access to a certain class obtains access to that class and its descendants in the hierarchy. A key management scheme assigns keys to the access classes and distributes a subset of the keys to a user, which permit her to obtain access to objects at her class(es) and all of the descendant classes. Such key management schemes are usually evaluated by the number of total keys the system must maintain, the number of keys each user receives, the size of public information, the time required to derive keys for access classes, and work needed to perform when the hierarchy or the set of users change.

Hierarchies of access classes are used in many domains, and in many cases they are more general than trees. The most traditional example of such hierarchies is Role-Based Access Control (RBAC) models [16, 41] that can be used for many different types of organizations. Other areas where hierarchies are useful are content distribution (where the users receive content of different quality or resolution), cable TV (where certain programs are included in subscription packages), project development (different views of information flow and components at managerial, developers, etc. positions), defense in depth (at each stage of intrusion defense there is a specific set of resources that can be accessed), and others. Even more broadly, hierarchical access control is used in operating systems (see, e.g., [18]), databases (e.g., [15]), and networking (e.g., [36, 33]).

A vital aspect of access control schemes is computational

and storage space requirements for key management and processing. It is clear that low requirements allow a scheme to be used in a much wider spectrum of devices and applications (e.g., inexpensive smartcards, small battery-operated sensors, embedded processors, etc.) than costly schemes. Thus to make our scheme acceptable for use with weak clients, we do not use powerful cryptography but instead utilize only cryptographic hashes. Throughout this paper, we use the word "smartcard" as a shorthand to refer to any type of a weak client.

Security of access control models comes from their ability to deny access to unauthorized data. Also, if a scheme is *collusion-resilient*, then even if a number of users with access to different nodes conspire trying to derive additional keys, they cannot get access to more nodes than what they can already legally access. Even though we intend to use the scheme with tamper-resistant smartcards, a number of prior publications (e.g., [2, 3]) suggest that compromising cards is easier than is commonly believed. In addition, the collusion-resilience allows us to use the scheme with other devices that do not have tamper-resistance.

One of the key efficiency measures for hierarchical access control schemes is the number of operations needed to compute the key for an access class lower in the hierarchy because this operation must be performed in real-time by possibly very weak clients. The best schemes (including ours) require the number of bit operations linear in the depth of the graph in the worst case, which for some graphs is $O(n)$ where $n$ is the number of nodes in the access graph (see the related work section for more detail). While the number of bit operations for key derivation is going to be small on average and an organization's role hierarchy tends to be shallow rather than deep, deep hierarchies do arise in many situations such as:

- Hierarchically organized hardware, where the hierarchy is based on functional and control issues but also on how trusted the hardware components are;
- Hierarchically organized distributed control structures such as physical plants or power grids (involving thousands of possibly tiny networked devices such as sensors, actuators, etc.);
- Hierarchical design structures (e.g., aircraft, VLSI circuits, etc.);
- Task graphs where only an ancestor task should know about descendant tasks.

Also, deep access hierarchies can arise even in very simple databases where the hierarchical complexity can come from super-imposed classifications on the database that are based on functional, structural, etc. features of a database. See also [35, 39] for other examples of deep hierarchies. This is why a rather substantial part of this work is dedicated to improving key derivation time, which, as we describe below, can be decreased to a small number of operations ($O(\log \log n)$ or even only 3 hashes) with modest increase in public storage space.

**Our Results.** Our scheme works for arbitrary access graphs, even those that may contain cycles. In the scheme, only hash functions are used for a node to derive a descendant's key from its own key. The space complexity of the public information is the same as that of storing $G$ and is asymptotically optimal; the private information at a node consists of a single key associated with that node. The derivation by a node of a descendant node's access key requires the number of bit operations linear in the distance between the nodes. Updates are handled locally in the hierarchy and do not "propagate" to descendants or ancestors of the affected part of the graph, while many other schemes require re-keying of other nodes following a deletion. Our scheme is resistant to collusion in that no subset of nodes can conspire to gain access to any node that is not already legally accessible. We address key management at the levels of both access classes and individual users, while other schemes manage keys only at one of these levels.

In the scheme, we rely on the following assumptions: there is a trusted central authority that can generate and distribute keys (e.g., an administrator within the organization). The security of our scheme relies on the use of pseudo-random functions.

We also show that the scheme can 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, for instance, 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), the model can cover a much richer set of access control policies than that of other schemes. These extensions are modeled after Crampton's work [13] and do not increase the space or computational complexity of our scheme.

A substantial part of this work is dedicated to improving efficiency of key derivation time for deep hierarchies. Our technique is to insert additional (so called "shortcut") edges in the graph, that allow us to achieve somewhat surprising results: for $n$-node trees our techniques enable us to improve efficiency of key derivation to $O(\log \log n)$ bit operations in the worst case with constant increase in public information, and to only 3 hashes with public space usage of $O(n \log \log n)$. We also describe how to apply our techniques to more general hierarchies. These techniques allow us to achieve the fastest key derivation known to date.

**Organization.** Section 2 provides an overview of related literature. In section 3, we give a formal description of the problem. Section 4 presents our base model along with its security proof; then in section 5 we give description of dynamic versions of the model and extensions that permit the scheme's usage with other access models given in [13]. Section 6 presents our techniques to improve efficiency of key derivation for trees and also comments on more general hierarchies. Finally, section 7 concludes the paper.

## 2. RELATED WORK

The first work that addressed the problem of key management in hierarchical access control was by Akl and Taylor [1] in 1983. Since then a large number of publications ([5, 6, 7, 9, 10, 11, 12, 14, 17, 22, 23, 26, 25, 27, 29, 30, 31, 34, 37, 38, 40, 42, 43, 45, 46, 51, 52, 53, 54] and others) have improved existing key assignment schemes, especially in the recent years. All of these approaches assume existences of a central authority (CA) that maintains the keys and related information. Most of them (and our scheme as well) are also

based on the idea that a node in the hierarchy can derive keys for its descendants. Due to the large number of previous publications, we only briefly comment on their basic ideas and efficiency in comparison to our scheme.

A relatively large number of schemes on this topic have been shown to be either insecure with respect to the security statements made in these works [50, 49, 44, 47, 24] or incorrect [8]. Therefore, we do not take these schemes into consideration in our further discussion.

A significant number of schemes, e.g., [1, 34, 22, 6, 25, 23, 10, 37, 27, 38, 31, 43], operate large numbers computed as a product of up to $O(n)$ coprime numbers or, alternatively, a product of up to $O(n)$ large numbers, where $n$ is the number of nodes in the graph. Such numbers can grow up to $O(n \log n)$ (respectively, $O(n)$) bits long and are prohibitively large for most hierarchies (in case of [26] numbers grow up to $O(n^d)$, where $d$ is the number of immediate descendants). While in many of these approaches key derivation might seem consisting of one division and one modular exponentiation operation, in practice, division of two numbers even $O(n)$ bits long involves $O(n^2)$ operations, in addition to the use of expensive public crypto operations. Our key derivation, on the other hand, even in the base scheme is bounded by the depth of the access hierarchy and is $O(n)$ hash operations in the worst case.

Work of [29, 40, 42] is limited to trees and thus is of limited use. Work of [5, 45, 51] is concerned with a slightly different model having a hierarchy of users and a hierarchy of resources. The scheme of [5], however, is not dynamic; and in [45, 51] there are high rekeying overheads for additions/deletions (particularly because of slightly different requirements of the scheme) and the number of keys for a class is large for large hierarchies.

The work of [17] gives an information-theoretic approach, in which each user might have to store a large number of keys (up to $O(n)$), and insertions/deletions result in many changes. The scheme of [48] uses modular exponentiation, and additions/deletions require rekeying of all descendants. A number of schemes [14, 46, 7] are based on interpolating polynomials and give reasonable performance. In [46, 14], however, private storage at a node is up to $O(n)$ and additions/deletions require rekeying of ancestors. As was already mentioned above, we avoid rekeying on additions/deletions and store only one key per node. In [7], key derivation is less efficient than in our scheme, also public storage space is larger. Even though the authors speculate that schemes that perform the key derivation process iteratively are inefficient (which is the case in our scheme), their key derivation is less efficient due to usage of expensive modular exponentiation operations and interpolating polynomial evaluation.

Schemes that utilize sibling intractable function families (SIFF) [52, 53] are the only efficient approaches among early schemes. In these schemes, there is only one secret key per class, key derivation is a chain of SIFF function applications which can be implemented using polynomials. However, additions and deletions in [52] require rekeying of all descendants and in [53] all descendants should be rekeyed when a node is deleted.

A number of recent schemes [9, 11, 12, 30, 54] use overall structure similar to ours and have performance comparable to our base scheme. [12], however, does not address dynamic changes, and the scheme is less efficient than ours because of additional usage of modular multiplication. [9] requires larger public storage, key derivation is slower because of additional usage of encryption, and the ex-member problem is not addressed that will require to rekey all descendants on deletions. Compared to the schemes [30] and [54], our approach is simpler than both of them. It is also more efficient than the first scheme (by a constant factor), and uses less space than both of them (by a constant factor). In addition, in both of these schemes, all descendants have to be rekeyed when a class is being deleted to combat the ex-member problem. [11] uses only hash functions and achieves performance closest to our base scheme; deletions, however, require rekeying of all descendants. In our scheme, on the other hand, dynamic changes to the graph are handled locally (i.e., private information at other nodes is not affected and no other nodes need to be re-keyed, only public information associated with the graph changes). In addition, the above schemes do not provide formal proofs of security.

Results achieved in this work can also be achieved using broadcast encryption, since broadcast encryption schemes are more powerful than our scheme. Such schemes, however, require significantly higher overheads than our scheme due to their added power and are not suitable in our setting (where the goal is to make the scheme work with weak clients).

# 3. 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 a class in the access hierarchy and has a set of objects associated with it. Function $\mathcal{O} : V \rightarrow 2^O$ maps a node to a unique set of objects such that $|\mathcal{O}(v_i)| \geq 0$ and $\forall i \, \forall j$, $\mathcal{O}(v_i) \cap \mathcal{O}(v_j) = \emptyset$ iff $i \neq j$. (For the sake of brevity of exposition we use notation $\mathcal{O}_i$ to mean $\mathcal{O}(v_i)$.) When the set of edges $E$ or the set of objects $O$ is not essential to our current discussion, we may omit it from the definition of the graph and instead use notation $G = (V, O)$ or $G = (V, E)$, respectively.

In a directed graph $G = (V, E)$, we define an ancestry function $Anc(v_i, G)$ which is a set such that $v_j \in Anc(v_i, G)$ if there is a path from $v_j$ to $v_i$ in $G$. We also define the set of descendants of node $v_i$ as $Desc(v_i, G)$, where $v_j \in Desc(v_i, G)$ if there is a path from $v_i$ to $v_j$ in $G$. For a directed graph $G = (V, E)$, we use a function $Pred(v_i, G)$ to denote the set of immediate predecessors of $v_i$ in $G$, i.e., if $v_j \in Pred(v_i, G)$ then there is a directed edge from $v_j$ to $v_i$ in $G$. Similarly, we define $Succ(v_i, G)$ to be the set of immediate successors of $v_i$ in $G$. When it is clear what graph we are discussing, we omit $G$ from the notation and instead use the shorthand notation $Anc(v_i)$, $Desc(v_i)$, $Succ(v_i)$, and $Pred(v_i)$. 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.

In the access hierarchy, a path from node $v_i$ to node $v_j$ means that any subject that can assume access rights at class $v_i$ is also permitted to access any object at class $v_j$ 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 a node $v_i \in V$ to a set of objects accessible to a subject at class $v_i$; the function is defined as $\mathcal{O}_i^* = \bigcup_{v_j \in Desc(v_i)} \mathcal{O}_j$.

We 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 that object. The goal is to minimize the number of keys per access class and the number of keys with which an object is encrypted. Formally, the key allocation policy is defined as:

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

In our schemes, to keep an ancestor's key space small, a node's key is computable from any of its ancestors keys via a hash 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.*

Now we are 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 graph $G = (V, O)$ 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 class has rights to access, that access class 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 class does not have rights to access, there is no key in that access class key space that can be used to generate any of the keys used to encrypt the object.*

DEFINITION 4. *A key allocation $KA$ that implements an access graph $G = (V, O)$ is fully collusion-resilient (or just collusion-resilient) iff for any set of adversaries with access to nodes $V' = v_{i_1}, \ldots, v_{i_r}$, 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.*

# 4. BASE SCHEME

This section describes our scheme in which every node has one key associated with it, the public information is linear in the size of the access graph $G$, and computation by node $v$ of a key that is $\ell$ levels below it can be done in $\ell$ evaluations of a hash function. Here we focus on key allocations for a static access hierarchy; extensions of this base scheme are given in section 5.

Assume that we are given a cryptographic hash function $H : \{0,1\}^\star \to \{0,1\}^\rho$.

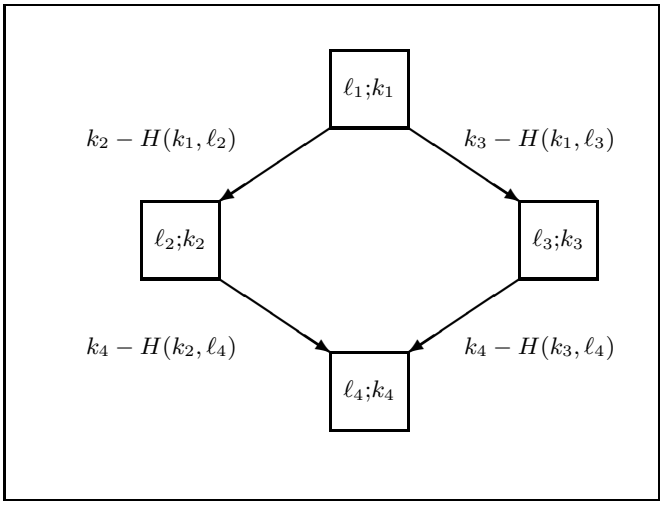


**Figure 1:** Key allocation for example access graph; all arithmetic is modulo $2^\rho$.

**Key generation.** The private key generation process and the nature of public information stored at each node of the graph 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 $\ell_i$ in $\{0,1\}^\rho$ that is assigned to the vertex. Also for each edge $(v_i, v_j)$, the value $y_{i,j} = k_j - H(k_i, \ell_j) \bmod 2^\rho$ is stored publicly for this edge[1].

**Key derivation.** All that needs to be shown is how to generate a child's key from the parent's private information and the public information. Suppose $v_i$ is a parent of $v_j$ with respective keys $k_i$ and $k_j$. Now, $\ell_j$ and $y_{i,j} = k_j - H(k_i, \ell_j) \bmod 2^\rho$ are public information. Clearly, node $v_i$ can generate $k_j$ with this information.

*Example:* Figure 1 shows key allocation for a graph more complicated than a tree, for which we give two examples. First, it is possible for the node with $k_1$ to generate key $k_2$, because that node can compute $H(k_1, \ell_2)$ and use it, along with the public edge information, to obtain $k_2$. The node with $k_3$, on the other hand, cannot generate $k_2$, since this would require inversion of the $H$ function.

THEOREM 1. *The above scheme is complete.*

PROOF. Suppose that a card has key $k_i$ corresponding to node $v_i \in V$. Also assume that access to object $o_j$ at node $v_j$ (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 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 $\ell_y$ and $y_{x,y}$ are known,

[1]Here we use subtraction to hide the value of $k_j$ with $H(k_i, \ell_j)$, but other ways of hiding $k_j$ using $H(k_i, \ell_j)$ as the key (e.g., using XOR operation) are also possible.

$v_i$ can generate $v_y$'s private information $k_y$. 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 above scheme is sound, even in the presence of collusion.*

To prove this, we first need to provide additional definitions and formally state the adversarial model. Our proof of security is based on the standard model assuming that $H(k, \ell)$ can be implemented as a pseudo-random function (PRF) $F_k(\ell)$. We show security of the scheme against active adversary who is allowed to adaptively corrupt nodes in the graph. After corrupting some nodes, the adversary is presented with a challenge: it is asked to recover the key of a node that is not a descendant of a corrupted node (the adversary is allowed to corrupt additional nodes that comply with this condition). We claim that if the adversary wins this game with a non-negligible probability, then we can construct an adversary who obtains non-negligible advantage in breaking the security of PRF, contradicting the definition of PRF.

Before proceeding further, we give a definition of a pseudo-random function. Let us fix a family of functions $F : Keys(F) \times D \rightarrow R$, and let $A$ be an algorithm that takes an oracle for a function $g : D \rightarrow R$ and returns a bit. Function $g$ can be either drawn at random from $Rand^{D\rightarrow R}$, i.e., $g \xleftarrow{R} Rand^{D\rightarrow R}$, or it is drawn at random from $F$, i.e., $g \xleftarrow{R} F$.

DEFINITION 5. *Consider two experiments:*

| *Experiment* $Exp^{prf-1}_{F,A}$ | *Experiment* $Exp^{prf-0}_{F,A}$ |
|---|---|
| $K \xleftarrow{R} Keys(F)$ | $g \xleftarrow{R} Rand^{D\rightarrow R}$ |
| $d \leftarrow A^{F_K}$ | $d \leftarrow A^{g}$ |
| *Return* $d$ | *Return* $d$ |

*The prf-advantage of $A$ is then defined as*

$$Adv^{prf}_{F,A} = Pr[Exp^{prf-1}_{F,A} = 1] - Pr[Exp^{prf-0}_{F,A} = 1]$$

For simplicity of exposition we do not take resources used by an adversary into account, and it is assumed that, given the same resources, we choose an adversary that gains maximum advantage.

DEFINITION 6. *Let $F^\rho$ denote a family of functions with input length $l(\rho)$, output length $L(\rho)$, and key length $k(\rho)$, where $\rho$ is a security parameter. Then $F^\rho$ is PRF if $F^\rho(K, x)$ is polynomial-time computable (i.e., in time poly$(\rho)$) and also the function $Adv^{prf}_{F^\rho,D^\rho}$ is negligible (in $\rho$) for every polynomial-time distinguisher $D^\rho$ that halts in time poly$(\rho)$.*

Now assume that adversary $B$ is given access to the public information associated with the key assignment of $G$ and is allowed to adaptively corrupt nodes from $V$. That is, $B$ obtains $k_i \leftarrow KA(v_i)$, where $v_i \in V$ and can compute $h \leftarrow F_{k_i}(\ell)$ for arbitrary labels $\ell \in \{0,1\}^\rho$. At some point, $B$ makes a single query to a challenge oracle $v_c \leftarrow C(G)$, where $v_c$ is a node of the graph not a descendant of any corrupted nodes and is chosen by the oracle. After that, $B$ may corrupt more nodes that do not have the challenge node $v_c$ among their descendants. At some point $B$ outputs a key $\hat{k} \in \{0,1\}^\rho$ and wins if $\hat{k} = k_c$.

DEFINITION 7. *Let $KA$ be a key allocation that implements an access graph $G = (V, O, E)$ and let $B$ be an algorithm that has access to oracles as above and returns a string in $\{0,1\}^\rho$. We consider the following experiment:*

*Experiment* $Exp^{kr}_{KA,B}$
$\hat{k} = B^{KA(v_i),C(G)}$
*if after a call to $v_c = C(G)$ $B$ makes a query $KA(v_i)$ where $v_c \in Desc(v_i)$, return 0*
*if $\hat{k} = k_c$ then return 1*
*else return 0*

*The* kr-advantage *of $B$ is defined as*

$$Adv^{kr}_{KA,B} = Pr[Exp^{kr}_{KA,B} = 1].$$

While the above definition assumes an adaptive adversary, in our case this adversary is no more powerful than a static adversary that is given the maximum amount of information. That is, if an adversary $B'$ is given a challenge node $v_c$, keys for every child of $v_c$, and keys for every sibling of each node on the way from the root to $v_c$ (i.e., $B'$ obtains keys for siblings of nodes $v_j$ s.t. $v_j \in Anc(v_c)$), then $B'$ can (efficiently) generate keys for all nodes in the graph except $v_c$ and its ancestors. To be more specific, adversary $B'$ obtains access to a single oracle that returns a challenge node $v_c$ along with all of the node keys as described above and $B'$ eventually outputs its guess for $k_c$.

Since usage of a static adversary makes our presentation easier, in our further discussion we will assume that a static adversary with maximal power is used.

If the adversary $B'$ has non-negligible advantage in the key recovery experiment, then we can construct an adversary $A_{B'}$ that uses $B'$ and can distinguish between a PRF and a random function with non-negligible probability (i.e., break the security of PRFs) and with a very small increase in the resource usage (namely, one application of a PRF and one subtraction operation).

LEMMA 1. $Adv^{prf}_{KA,A_{B'}} \geq Adv^{kr}_{KA,B'} - \frac{1}{2^\rho}$

PROOF. We construct an adversary $A_{B'}$ that will distinguish between random and pseudo-random functions using algorithm $B'$. Instead of using public information associated with the graph $G = (O, V, E)$ constructed according to the above key assignment scheme, in this experiment public information is constructed in such a way that with 50% probability the key assignment is performed in the usual way, and with 50% probability one of the functions $F_{k_c}$ ($v_c \in V$) is replaced with a random function $g$. $A_{B'}$ obtains access to the same oracle $C(G)$ as $B'$ did, and when querying this oracle obtains a challenge node $v_c$ along with the keys of the children of $v_c$ and siblings of ancestors of $v_c$ (let this set of keys be denoted as $\mathcal{K}_c$ so that $\{v_c, \mathcal{K}_c\} = C(G)$). $A'_B$ is then asked to decide whether $F_{k_c}$ or $g$ was used in the key assignment. It can be constructed as the following:

Adversary $A_{B'}$
$\{v_c, \mathcal{K}_c\} = C(G)$
Run adversary $B'$ replying to its oracle query with $\{v_c, \mathcal{K}_c\}$
When $B'$ outputs a key $\hat{k}$, compute $F_{\hat{k}}(l_j)$ where $v_j$ is one of the children nodes of $v_c$
if $y_{c,j} = k_j - F_{\hat{k}}(l_j) \bmod 2^\rho$, then return 1, else return 0

In the above algorithm, if $B'$ guesses the key correctly, $A_{B'}$ assumes that the PRF was used. If $B'$ doesn't return the correct key, $A_{B'}$ bets on the random function. Now the *prf-advantage* of $A_{B'}$ is:

$$\begin{aligned} Adv^{prf}_{KA,A_{B'}} &= Pr[1 = A^{C(G)}_{B'} | F_{k_c} \text{ was used}] \\ &- Pr[1 = A^{C(G)}_{B'} | g \text{ was used}] \\ &\geq Adv^{kr}_{KA,B'} - \frac{1}{2^\rho} \end{aligned}$$

because if $F_{k_c}$ was used, $A_{B'}$ will guess correctly at least with the same probability as $B'$, and if $g$ was used, the probability that $F_{\hat{k}}(l_j)$ results in the same value as $g(l_j)$ is $\frac{1}{2^\rho}$. □

PROOF OF THEOREM 2. Now the proof of security follows directly from Lemma 1, which states that if an adversary can break the scheme with non-negligible probability, it will also be able to break the security of PRFs. □

With only minimal changes to the scheme, security under the key indistinguishability can be shown. We leave these results to the full version of the paper.

# 5. EXTENSIONS

## 5.1 Dynamic Version

While section 4 described the base scheme without dynamic changes, in this section we show how with one simple modification we can perform all dynamic changes locally.

**Modified private key** Each vertex $v_i$ is assigned a random private key $\hat{k}_i$ in $\{0,1\}^\rho$. As before, an entity that is assigned access levels $V' \subseteq V$ is given a smartcard with all keys for their access levels $v_j \in V'$. The actual key used for this access level is now $k_i = H(\hat{k}_i, \ell_i)$. Note that by changing a vertex's label one can change its key.

The rest of the scheme remains unchanged. Now we describe how to handle dynamic changes.

**Insertion of an edge.** Suppose the edge $(v_i, v_j)$ is inserted into $G$. Then we simply add $y_{i,j} = k_j - H(k_i, \ell_j) \bmod 2^\rho$ to the description of $G$ by attaching it to the edge $(v_i, v_j)$.

**Deletion of an edge.** Deleting an edge is trivial, but the difficulty is in preventing ex-member access. 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. Change the label of $v_t$, call it $\ell'_t$; note that this changes the key for $v_t$ to $H(\hat{k}_t, \ell'_t)$.
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.

**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 do the following:

1. Create 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.
2. Add the edges one by one, using each time the above procedure for edge-insertions.

**Deletion of a node.** If a node $v_i$ is deleted, together with all the edges that touch it, we need to perform two steps:

1. Delete the edges touching $v_i$ one by one, using the above procedure for edge-deletions.
2. Now that $v_i$ has no edges touching it, removing it is trivial.

**Key replacement.** Key replacement for a node $v_i$ is performed as the following:

1. Update the node's key $\hat{k}_i$ with a new key $\hat{k}'_i$.
2. Update the vertex's access key to $k'_i = H(\hat{k}'_i, \ell_i)$.
3. Update edges $(v_j, v_i)$ where $v_j \in Pred(v_i)$ with $y_{j,i}$ computed according to the new key $\hat{k}'_i$.
4. Update edges $(v_i, v_l)$ where $v_l \in Succ(v_i)$ with $yi, l$ computed according to the new key $\hat{k}'_i$.

No other node is affected.

**User revocation.** To the best of our knowledge, no prior work on hierarchical access control considered key management at the level of access classes and at the same time at the level of individual users. For instance, among the schemes closest to ours, [54] considers only a hierarchy of security classes without mentioning individual users, and [30] considers a hierarchy of users without grouping them into classes. However, it is important to group users with the same privileges together and on the other hand permit revocation of individual users. In our scheme, revoking a single user can be done with two approaches:

1. Recard every user at that user's access level(s) and for all descendants of this access level(s) perform the operation described for edge deletion (i.e., change all keys by changing the labels and then update the public information). Note that the descendants do not have to rekeyed.
2. Make the graph such that each user is represented by a single node in the graph with edges from this node to each of that user's access levels. By creating such a graph, removing a user is as easy as removing this node, and thus does not require rekeying.

## 5.2 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 [13] 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 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 node for a special class of access graphs that are "layered" directed acyclic graphs (DAGs) (we define this later) when there is no collusion.

### 5.2.1 Downward inheritance

To handle such queries, we construct the reverse of the graph $G = (V, E, O)$, which is $G^R = (V, E', O)$ 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 or downwards.

### 5.2.2 Limited depth permission inheritance

We say that an access graph is *layered* if the nodes can be partitioned into 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 already layered, but in general 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 there is 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 that permit access to all nodes higher than some level and to 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 to access "all ancestors of node $v_i$" is $k_i$ and the key for permission requirement to access "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.

# 6. IMPROVING EFFICIENCY

As the scheme described in the previous sections supports arbitrary graphs, it is possible to add edges to an access structure in order to reduce the path length between two nodes. In this section we consider how to add edges to trees so that the distance between any two nodes is small. This is essential for deep hierarchies since the key derivation time in our base scheme is the depth of the access graph in the worst case. In the remainder of this section we assume that the access structure is a tree with $n$ nodes. Sections 6.1–6.3 describe our first approach that achieves $O(\log \log n)$ hash functions for key derivation with $O(n)$ public space, and section 6.4 describes an alternative approach that requires only 3 hash function applications for key derivation with $O(n \log \log n)$ public storage space. Then section 6.5 addresses dynamic behavior and section 6.6 covers more general hierarchies.

## 6.1 A Preliminary Scheme

First we review some background material that is needed for our scheme. A *centroid* of an $n$-node tree $T$ is a node whose removal from $T$ leaves no connected component of size greater than $n/2$ [28]. The tree $T$ does not need to be binary or even have constant-degree nodes. It is easy to prove that there are at most two centroids, and if there are two centroids, then they must be adjacent. However, if the tree is rooted and has two centroids, we can break the tie by arbitrarily selecting the parent among the two centroids. Thus we shall refer to "the" centroid of a rooted tree. Now we are ready to describe the preliminary scheme for computing the edges that we add to the tree and to which we refer as shortcut edges.

**Input:** The tree $T$.

**Output:** A set of $O(n \log n)$ shortcut edges such that there is a path of length less than $\log n$ between any ancestor-descendant pair.

**Algorithm Steps:** For every node $v$ of $T$, do the following:

1. Let $T_v$ be the subtree of $T$ rooted at $v$. Compute the centroid of $T_v$ (call it $c_v$).
2. Add a shortcut edge from $v$ to $c_v$ (unless such a tree edge already exists or $v = c_v$).
3. Remove from $T_v$ its subtree rooted at $c_v$. Note that the new $T_v$ is now at most half its previous size (and could in fact be empty if $v = c_v$).
4. Repeat the above process for the new $T_v$ until the final $T_v$ is empty.

The number of shortcut edges leaving each $v$ in the above description is no more than $\log n$ because each addition of a shortcut edge results in at least halving the size of $T_v$. Therefore the total number of shortcut edges is no more than $n \log n$.

Now we show that the shortcut edges make it possible for every ancestor $v$ to reach any of its descendants $w$ in a path of no more than $\log n$ edges. When we trace the path from $v$ to $w$, we distinguish two cases, depending on whether $w$ is in the subtree of the centroid $c_v$ of $T_v$. The tracing algorithm is as follows:

**Case 1:** $w$ is in the subtree of the centroid $c_v$ of $T_v$. Then if $v \neq c_v$, we follow the edge from $v$ to $c_v$, and we continue recursively down from $c_v$. If, on the other hand, $v = c_v$, then we follow the tree edge from $v$ to that child of $v$ whose subtree contains $w$ and we continue recursively down from there.

**Case 2:** $w$ is not in the subtree of $c_v$ in $T_v$. Then we recursively continue down with a $T_v$ that is "truncated" by the (implicit) removal of $T_{c_v}$ from it (so it is now half its previous size).

The fact that the path traced by the above approach consists of no more than $\log n$ edges follows from the observation that every time we follow an edge (whether it is a tree edge or a shortcut edge), we end up at a node whose subtree is at most half the size of the subtree we were at.

## 6.2 Improving the Time Complexity

Before describing the improved scheme, we need to review the concept of centroid decomposition of a tree: If we compute the centroid of a tree, then remove it, and recursively repeat this process with the remaining trees (of size no more than $n/2$ each), we obtain a decomposition of the tree into what is called a "centroid decomposition". Such a decomposition can be easily computed in linear time (see, for example, [21]).

Our improved scheme is based on doing a pre-processing step of $T$ that consists of carrying out what might be called a "prematurely terminated centroid decomposition". This is similar to the above-described centroid decomposition, except that we stop the recursion not when the tree becomes

a single node, but when the tree size becomes $\leq \sqrt{n}$. This means that there are at most $\sqrt{n}$ successive centroids that are affected by the "prematurely terminated" decomposition (as opposed to $n$ of them for the standard decomposition). We call these centroids, as well as the root of $T$, the *special* nodes. Note that, by construction, removing the special nodes from $T$ leaves connected components of size at most $\sqrt{n}$ each; we call these connected components (which are trees) the "residual" trees and denote them by $T_1, \ldots, T_k$.

We also use the notion of a "reduced tree" $\hat{T}$. The tree $\hat{T}$ consists of the $O(\sqrt{n})$ special nodes and of edges that satisfy the following condition: There is an edge from node $x$ to node $y$ in $\hat{T}$ iff (i) $x$ is an ancestor of $y$ in $T$, and (ii) there is no other node of $\hat{T}$ on the $x$-to-$y$ path in $T$.

Now we are ready to describe the overall recursive procedure for adding shortcuts. In what follows, $|T|$ denotes the number of vertices in $T$.

**AddShortcuts**($T$):

1. If $|T| \leq 4$ then return an empty set of shortcuts. Otherwise continue with the next step.
2. Compute the special nodes of $T$ in linear time. Initialize the set of shortcuts $S$ to be empty.
3. Create, from $T$, the reduced tree $\hat{T}$ and add to $S$ a shortcut edge between every ancestor-descendant pair in $\hat{T}$ (unless the ancestor is a parent of the descendant, in which case there is already such an edge in $T$). Because $\hat{T}$ has $O(\sqrt{|T|})$ vertices, the size of $S$ is $O(|T|)$.
4. For every residual tree $T_i$ in turn ($i = 1, \ldots, k$), add to $S$ a shortcut edge from the root of $T_i$ to every node in $T_i$ that is not a child of that root. This increases the size of $S$ by no more than $\sum_{i=1}^{k} |T_i|$, which is $\leq |T|$.
5. For every residual tree $T_i$ in turn ($i = 1, \ldots, k$), recursively call **AddShortcuts**($T_i$) and, if we let $S_i$ be the set of shortcuts returned by that recursive call, then we update $S$ by doing $S = S \cup S_i$.
6. Return $S$.

The number $f(|T|)$ of shortcut edges added by the above recursive procedure obeys the recurrence

$$f(|T|) = \begin{cases} 0 & \text{if } |T| \leq 4 \\ f(|T|) \leq c_1|T| + \sum_{i=1}^{k} f(|T_i|) & \text{if } |T| > 4 \end{cases}$$

where every $|T_i|$ is $\leq \sqrt{n}$, and $c_1$ is a constant. A straightforward induction proves that this recurrence implies that $f(|T|) = O(|T| \log\log |T|)$. Therefore the space for the public data is $O(n \log\log n)$, due to the creation of the $f(n)$ shortcut edges.

We now turn our attention to showing that, for every ancestor-descendant pair $x$ and $y$ in $T$, there is now, due to the shortcuts, an $x$-to-$y$ path of length $O(\log\log |T|)$. The recursive procedure for finding such a path is given next, and mimics the recursion of **AddShortcuts**() (uses same $\hat{T}$, same $T_i$'s, etc.). In it, we use $Length(n)$ to denote the worst-case length of a shortest ancestor-to-descendant path that can avail itself of the shortcuts generated in the above **AddShortcuts**().

**FindPath**($x, y, T$):

1. If $T| \leq 4$ then trace a path from $x$ to $y$ along $T$ and return that path. If $|T| > 4$ continue with the next step.
2. If $x$ and $y$ are both special in $T$ (i.e., both are nodes of $\hat{T}$) then return the edge $(x, y)$. (Note that such an edge exists because of Step 3 in **AddShortcuts**().) If $x$ and/or $y$ is not special, then proceed to the next step.
3. Let $T_i$ be the residual tree containing $x$, and let $T_j$ be the residual tree containing $y$. If $i = j$ then we recursively call **FindPath**($x, y, T_i$), which returns a path in $T_i$ that is of length $\leq Length(|T_i|)$, which is $\leq Length(\sqrt{|T|})$. We return that path. If $i \neq j$ (i.e., $x$ and $y$ are in different residual trees) then we proceed as follows:
   (a) We recursively call **FindPath**($x, z, T_i$) where $z$ is the node of $T_i$ that is nearest to $y$ in $T$ (hence $z$ is a leaf of $T_i$, and one of its children $z'$ in $T$ is a special node that is ancestor of $y$ in $T$). The length of this $x$-to-$z$ path is $\leq Length(|T_i|)$, which is at most $Length(\sqrt{|T|})$. This path is the initial portion of the path $\mathcal{P}$ that will be returned by the recursive call ($\mathcal{P}$ will be further built in the steps that follow).
   (b) Follow the edge in $T$ from $z$ to the special node $z'$ that is ancestor of $y$ in $T$, and append that edge $(z, z')$ to $\mathcal{P}$.
   (c) Follow (and append to $\mathcal{P}$) the edge in $\hat{T}$ from special node $z'$ to the special node (call it $u$) that is the special ancestor of $y$ that is nearest to $y$ (hence $u$ is parent of the root of the residual tree $T_j$ that contains $y$). Note that such an edge exists because of Step 3 in **AddShortcuts**(). If $u = y$ then return $\mathcal{P}$, otherwise continue with the next step.
   (d) Follow (and append to $\mathcal{P}$) the edge in $T$ from $u$ to the root of $T_j$.
   (e) Follow (and append to $\mathcal{P}$) the edge from the root of $T_j$ to $y$; such an edge exists because of Step 4 in **AddShortcuts**().
   (f) Return $\mathcal{P}$.

The recurrence for $Length$ implied by the above recursive procedure is:

$$Length(|T|) = \begin{cases} Length(|T|) \leq c_2 & \text{if } |T| \leq 4 \\ Length(|T|) \leq c_3 + Length(\sqrt{|T|}) & \text{if } |T| > 4 \end{cases}$$

where every $|T_i|$ is $\leq \sqrt{n}$, and the $c_i$'s are constants. A straightforward induction proves that this recurrence implies that $Length(|T|) = O(\log\log |T|)$. Therefore the worst-case time for key derivation is $O(\log\log n)$.

The next section deals with decreasing the space complexity of the public information to $O(n)$.

## 6.3 Improving the Space Complexity

We begin with a pre-processing step of $T$ that consists of carrying out "prematurely terminated centroid decomposition" similar to the one used in the previous section, except

that we stop the recursion not when the tree becomes of size $\leq \sqrt{n}$, but when the tree size becomes $\leq \log\log n$. This means that there are at most $O(n/\log\log n)$ successive centroids that are affected by this new form of "prematurely terminated" decomposition. We call these $O(n/\log\log n)$ nodes, as well as the root of $T$, the *distinguished* nodes (these will be treated differently from the "special" nodes defined in the previous section). Note that, by construction, removing the distinguished nodes from $T$ leaves connected components of size at most $\log\log n$ each; we call these connected components (which are trees) the "tiny trees".

The next thing that we use is the notion of a "reduced tree" $T'$ that is conceptually similar to the $\hat{T}$ of the previous section: The nodes of $T'$ are the distinguished nodes plus the root – hence there are $O(n/\log\log n)$ nodes in $T'$ (whereas there were $O(\sqrt{n})$ nodes in $\hat{T}$). The edges of $T'$ satisfy the following condition: There is an edge from node $x$ to node $y$ in $T'$ if and only if (i) $x$ is an ancestor of $y$ in $T$, and (ii) there is no other node of $T'$ on the $x$-to-$y$ path in $T$.

Now we are ready to put the pieces together:

1. Compute the distinguished nodes of $T$ in linear time.
2. Create the tree $T'$.
3. Use the method of Section 6.2 on the tree $T'$. Any edge of $T'$ that was not in $T$ must be considered a new (i.e., a shortcut) edge. Note that the public space this takes is $O(n)$ because $|T'| = O(n/\log\log n)$. It allows computing an ancestor-to-descendant path of length at most $\log\log n - \log\log\log n$ between any ancestor-descendant pair of distinguished nodes in $T'$.
4. To find an ancestor-to-descendant path from $x$ to $y$ when $x$ and/or $y$ is not distinguished, do the following:
   (a) First trace a path in $T$ from $x$ to the nearest distinguished node (call it $z$) that is ancestor of $y$. The length of this path is at most $\log\log n$ because the "prematurely terminated centroid decomposition" that we described above stops at tiny trees of size $\leq \log\log n$. If there does not exist such a distinguished node $z$ that is both a descendant of $x$ and ancestor of $y$, then $x$ and $y$ are in the same $O(\log\log n)$ sized tiny tree of nondistinguished nodes. In this case we can directly go along edges of $T$ from $x$ to $y$ and stop.
   (b) Next, trace a path in $T'$ from $z$ to the distinguished node (call it $u$) that is the nearest distinguished ancestor of $y$. As stated above, the length of this path is at most $\log\log n - \log\log\log n$. If $u = y$ then stop, otherwise continue with the next step.
   (c) Trace a path in $T$ from $u$ to $y$. Because that path does not go through any distinguished node (other than $u$), it stays in one of the tiny trees and thus has length at most $\log\log n$.

   The above implies that the concatenation of the paths from $x$ to $z$, $z$ to $u$, $u$ to $y$, has length $O(\log\log n)$. The space is clearly linear.

Although the above method uses a different partitioning scheme from Section 6.2 (and in fact uses the scheme of that section as a subroutine), its spirit is the same: The use of a $T'$ as a "beltway" that connects the subtrees in which $x$ and $y$ reside.

## 6.4 A Time/Space Tradeoff

In this section we introduce schemes with constant time complexity. Our first scheme has space complexity $O(n\log\log n)$ and requires at most 3 hops to reach any node. Like the scheme outlined in Section 6.2, we start with prematurely terminated centroid decomposition that stops when the tree size is $\leq \sqrt{n}$. We also use the reduced tree $\hat{T}$. The approach is as follows.

**AddShortcuts**($T$):

1–4. The same as in the **AddShortcuts**() algorithm of Section 6.2.

5. For every residual tree $T_i$ in turn ($i = 1, \ldots, k$), add to $S$ a shortcut edge from each node $N$ in $T_i$ (other than the root) to all nodes in $\hat{T}$ that are both: (i) descendants of $N$ and (ii) children of the root of $T_i$ in $\hat{T}$. This adds at most $O(|T|)$ edges to the shortcut set: For each node $SN$ in $\hat{T}$, all of the new edges that point to $SN$ come from at most one tree (as $SN$ has at most one parent in $\hat{T}$). Furthermore, since each tree has at most $O(\sqrt{|T|})$ nodes, there are at most $O(\sqrt{|T|})$ edges pointing to $SN$ that are added during this step. Finally, there are only $O(\sqrt{|T|})$ nodes in $\hat{T}$, and so there are at most $O(|T|)$ edges added during this step.
6. For every residual tree $T_i$ in turn ($i = 1, \ldots, k$), recursively call **AddShortcuts**($T_i$) and, if we let $S_i$ be the set of shortcuts returned by that call, then we update $S$ by doing $S = S \cup S_i$.
7. Return $S$.

The number of edges added to the shortcut set in the above scheme follows a recurrence similar to the scheme in section 6.2; thus this scheme adds only $O(n\log\log n)$ edges. Furthermore, the Algorithm **FindPath**($x, y, T$) is very similar to the section 6.2 algorithm. To avoid unnecessarily repeating the above mentioned techniques, we describe only the case of the **FindPath**() algorithm that differs from its previous version. It corresponds to the situations where $x$ and $y$ are not in the same residual tree and neither of them are "special nodes". In this case, it takes at most one hop to get to a "special node" (call it $s_1$) that is an ancestor of $y$ (by Step 5 of **AddShortcuts**()). Once we are at a special node, we can get to the special node of the residual tree containing $y$ in a single hop (call this node $s_2$) by Step 3. From there we can reach $y$ with a single hop by Step 4. The path from $x$ to $y$ is thus $x, s_1, s_2, y$ which is 3 hops.

The above scheme requires only three hops to reach a specific node. It is trivial to show that a one-hop solution must add $O(n^2)$ edges, but a two-hop solution exists with only $O(n^{1.5})$ public space, which we briefly sketch here. First, we compute the special nodes of $T$ as in the above scheme and add two kinds of edges to this tree. The first kind of edge that we add to $S$ connects every ancestor-descendant pair in $T_i$ for each $T_i$ (unless the ancestor is a parent of the descendant, in which case there is already such an edge in $T$). Since each $T_i$ has $O(\sqrt{n})$ nodes, this step adds at most $O(n)$ edges to each $T_i$. There are $O(\sqrt{n})$ such trees, and so

the space required by this step is $O(n^{1.5})$. After this step all nodes in the same subtree can reach each other with a single hop. The second type of edge is from each node $N$ in $T$ to all special nodes that are descendants of $N$. As there are $O(\sqrt{n})$ special nodes and each node adds at most $O(\sqrt{n})$ such edges, there are at most $O(n^{1.5})$ such edges in total. If $x$ and $y$ are in different trees, then $x$ can get to the root of $y$'s subtree in one hop (by the second type of edge) and then to $y$ with one hop (by the first type of edge), thus any two nodes are no more than 2 hops away from each other.

## 6.5 Dynamic Behavior

This section examines the cost of maintaining the shortcut edges as the tree changes dynamically as a result of edge and node insertions and deletions. In the uniform-distribution random model for such dynamic updates, nothing else needs to be done: The structure retains its claimed properties (to within a constant factor) essentially for the same reason that an initially balanced tree data structure tends to remain balanced (to within a constant factor) as random insertions and deletions are carried out. If, on the other hand, the updates are not uniformly distributed, then the initial set of shortcuts may, over time, deviate from the properties we claimed. We can, however, show that the extra cost introduced by the need to maintain our shortcuts in the face of insertion and deletion operations, is $O(1)$ per operation in an *amortized* sense: After a sequence of $\sigma$ such operations, if $t_\sigma$ is the additional time (compared to without shortcuts) taken to maintain shortcut edges, then $t_\sigma/\sigma = O(1)$. The rest of this section proves this.

One possible strategy is the following: When the non-uniform updates have caused deviations from the desired performance bounds by more than a constant multiplicative factor $d$ (e.g., instead of the shortcuts providing an upper bound of $b \log\log n$, they now provide an upper bound of worse than $db \log\log n$), we discard all the shortcuts and replace them with new ones that reflect the new situation. Note that this does not affect the tree itself, only the shortcuts, so there is no need for re-keying any node. Note also that (i) this takes no more than linear time in the size of the new tree (because it is a re-computation of the shortcuts), and (ii) it suffices to know the current $n$ and the above-mentioned "flexibility factor" $d$ in order to determine when to initiate such a re-computation. The latter does not require us to detect and report every case when the path exceeds $db \log\log n$, but instead the shortcuts can be recomputed periodically every $\alpha n$ insertions/deletions. The cost is small and is only O(1) per operation because the $O(n)$ time it takes for one shortcut recomputation is amortized over the $\alpha n$ operations that occurred before the restructuring took place.

To complete the proof of $O(1)$ amortized shortcuts maintenance time, we must now show that a linear number of operations must have taken place before we were forced into a linear-time shortcuts-recomputation. To do this, we can think of the effect of insertions/deletions as *implicitly redefining the notion of centroid* to be more flexible than that of "no subtree of size more than $n/2$ when the centroid is removed". That is, hypothetically suppose that in our construction we replaced the notion of centroid by that of *c-approximate-centroid*: A node whose removal leaves subtrees with respective sizes of no more than $cn$ for a constant $c \geq 0.5$ (e.g., $c = 3/4$). If we did that, our claim of $O(\log\log n)$ performance would obviously still hold (only the constant factor hiding behind the "$O$" notation would change). Now note that, before any insertions and deletions, we have shortcuts that are consistent with the "rigid" (i.e., $n/2$) notion of a centroid. When we initiate a re-computation of shortcuts, the shortcut edges violate every $c$-approximate-centroid notion (because otherwise, as just stated, all ancestor-to-descendant paths would have double-logarithmic length). In order for shortcuts in a specific subtree with $n$ nodes to go from being initially consistent with a rigid $n/2$ notion of a centroid, to not being consistent with an (e.g.) $(3n/4)$-approximate-centroid notion, a linear number of insertions/deletions must have occurred in that subtree. This completes the proof.

## 6.6 More general hierarchies

Although this section dealt mainly with trees, the basic ideas can be adapted to more general hierarchies. If we moved from a tree to a more general graph, the notion of centroid is replaced by that of a *separator* — a "small" subset of the vertices whose removal leaves connected components of size no more than $cn$ for some constant $c = O(1)$. Such subsets are known to exist: A separator of size $O(\sqrt{n})$ exists for planar graphs [32], and the whole (recursive) separator structure for such a graph can be computed in linear time [20]. More general classes of graphs also have small separators, e.g., graphs of bounded genus [19]. Bounded genus seems to capture most RBACs (in fact, even the class of genus zero – planar graphs – seems to capture many commonly found "practical" RBACs).

# 7. CONCLUSIONS AND FUTURE WORK

In summary, we give the first solution to the problem of access control in an arbitrary hierarchy $G$ with the following properties:

1. Only hash functions are used for a node to derive a descendant's key from its own key;
2. The space complexity of the public information is the same as that of storing graph $G$;
3. The derivation by a node of a descendant's access key requires $O(\ell)$ bit operations, where $\ell$ is the length of the path between the nodes, for arbitrary hierarchies and $\log\log n$ or less for trees;
4. Updates are handled locally and do not "propagate" to descendants or ancestors of the affected part of $G$;
5. The scheme is resistant to collusion in that no subset of nodes can conspire to gain access to any node access to which they cannot legally obtain;
6. The private information at a node consists of a single key associated with that node.

We also provided simple modifications to our scheme that allow to handle Crampton's extensions of the standard hierarchies to "limited depth" and reverse inheritance [13], and gave shortcut schemes that permit to significantly reduce key derivation time for tree hierarchies.

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. Give a more developed shortcut scheme for general hierarchies.

## Acknowledgments

The authors thank anonymous reviewers for useful feedback on this work. The authors are also grateful to Nelly Fazio for valuable comments on the security proof of the scheme and her suggestions on improving security of the scheme. Her work that shows security of the scheme under key indistinguishability will be included in the full version of the paper and reported elsewhere.

## 8. 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] J. Birget, X. Zou, G. Noubir, and B. Ramamurthy. Hierarchy-based access control in distributed environments. In *ICC Conference 2001*, June 2001.

[6] C. Chang and D. Buehrer. Access control in a hierarchy using a one-way trapdoor function. *Computers and Mathematics with Applications*, 26(5):71–76, 1993.

[7] C. Chang, I. Lin, H. Tsai, H. Wang, and T. Taichung. A key assignment scheme for controlling access in partially ordered user hierarchies. In *International Conference on Advanced Information Networking and Application (AINA'04)*, 2004.

[8] T. Chen and Y Chung. Hierarchical access control based on chinese remainder theorem and symmetric algorithm. *Computers & Security*, 2002.

[9] T. Chen, Y. Chung, and C. Tian. A novel key management scheme for dynamic access control in a user hierarchy. In *IEEE Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 396–401, September 2004.

[10] G. Chick and S. Tavares. Flexible access control with master keys. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *LNCS*, pages 316–322, 1990.

[11] H. Chien and J. Jan. New hierarchical assignment without public key cryptography. *Computers & Security*, 22(6):523–526, 2003.

[12] J. Chou, C. Lin, and T. Lee. A novel hierarchical key management scheme based on quadratic residues. In *Internation Symposium on Parallel and Distributed Processing and Applications (ISPA'04)*, volume 3358, pages 858–865, December 2004.

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

[14] M. Das, A. Saxena, V. Gulati, and D. Phatak. Hierarchical key management scheme using polynomial interpolation. *ACM SIGOPS Operating Systems Review*, 39(1):40–47, January 2005.

[15] D. Denning, S. Akl, M. Morgenstern, and P. Neumann. Views for multilevel database security. In *IEEE Symposium on Security and Privacy*, pages 156–172, April 1986.

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

[17] A. Ferrara and B. Masucci. An information-theoretic approach to the access control problem. In *Italian Conference on Theoretical Computer Science (ICTCS'03)*, volume 2841, pages 342–354, October 2003.

[18] L. Fraim. Scomp: a solution to multilevel security problem. *IEEE Computer*, 16(7):126–143, July 1983.

[19] J. Gilbert, J. Hutchinson, and R. Tarjan. A separation theorem for graphs of bounded genus. *Journal of Algorithms*, 5:391–407, 1984.

[20] M. Goodrich. Planar separators and parallel polygon triangulation. In *Annual ACM Symposium on Theory of Computing*, pages 507–516, 1992.

[21] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Annual ACM Symposium on Computational Geometry*, pages 1–13, 1986.

[22] L. Harn and H. Lin. A cryptographic key generation scheme for multilevel data security. *Computers & Security*, 9(6):539–546, October 1990.

[23] M. He, P. Fan, F. Kaderali, and D. Yuan. Access key distribution scheme for level-based hierarchy. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'03)*, pages 942–945, August 2003.

[24] H. Huang and C. Chang. A new cryptographic key assignment scheme with time-constraint access control in a hierarchy. *Computer Standards & Interfaces*, 26:159–166, 2004.

[25] M. Hwang. An improvement of novel cryptographic key assignment scheme for dynamic access control in a hierarchy. *IEICE Trans. Fundamentals*, E82–A(2):548–550, March 1999.

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

[27] 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.

[28] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[29] H. Liaw, S. Wang, and C. Lei. A dynamic cryptographic key assignment scheme in a tree structure. *Computers and Mathematics with Applications*, 25(6):109–114, 1993.

[30] C. Lin. Hierarchical key assignment without public-key cryptography. *Computers & Security*, 20(7):612–619, 2001.

[31] I. Lin, M. Hwang, and C. Chang. A new key assignment scheme for enforcing complicated access

control policies in hierarchy. *Future Generation Computer Systems*, 19(4):457–462, 2003.

[32] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal Applied Mathemathics*, 36:177–189, 1979.

[33] W. Lu and M. Sundareshan. A moredle for multilevel security in computer networks. In *INFOCOM'88*, pages 1095–1104, 1988.

[34] 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.

[35] P. Maheshwari. Enterprise application integration using a component-based architecture. In *IEEE Annual International Computer Software and Applications Conference (COMSAC'03)*, pages 557–563, 2003.

[36] J. McHugh and A. Moore. A security policy and formal top level specification for a multi-level secure local area network. In *IEEE Symposiom on Security and Privacy*, pages 34–49, 1986.

[37] K. Ohta, T. Okamoto, and K. Koyama. Membership authentication for hierarchical multigroups using the extended fiat-shamir scheme. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, pages 446–457, February 1991.

[38] 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.

[39] J. Rose and J. Gasteiger. Hierarchical classification as an aid to database and hit-list browsing. In *International Conference on Information and Knowledge Management*, pages 408–414, 1994.

[40] 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.

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

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

[43] A. De Santis, A. Ferrara, and B. Masucci. Cryptographic key assignment schemes for any access control policy. *Information Processing Letters (IPL)*, 92(4):199–205, November 2004.

[44] V. Shen and T. Chen. A novel key management scheme based on discrete logarithms and polynomial interpolations. *Computers & Security*, 21(2):164–171, 2002.

[45] Y. Sun and K. Liu. Scalable hierarchical access control in secure group communication. In *IEEE INFOCOM 2004*, 2004.

[46] H. Tsai and C. Chang. A cryptographic implementation for dynamic access control in a user hierarchy. *Computers & Security*, 14(2):159–166, 1995.

[47] 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.

[48] J. Wu and R. Wei. An access control scheme for partially ordered set hierarchy with provable security. Cryptology ePrint Archive, Report 2004/295, 2004. texttthttp://eprint.iacr.org/.

[49] T. Wu and C. Chang. Cryptograpic key assignment scheme for hierarchical access control. *International Journal of Computer Systems Science and Engineering*, 1(1):25–28, 2001.

[50] J. Yeh, R. Chow, and R. Newman. A key assignment for enforcing access control policy exceptions. In *International Symposium on Internet Technology*, pages 54–59, 1998.

[51] Q. Zhang and Y. Wang. A centralized key management scheme for hierarchical access control. In *IEEE Global Telecommunications Conference (Globecom'04)*, 2004.

[52] Y. Zheng, T. Hardjono, and J. Pieprzyk. Sibling intractable function families and their applications. In *Advances in Cryptology – AsiaCrypt'91*, LNCS, 1992.

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

[54] S. Zhong. A practical key management scheme for access control in a user hierarchy. *Computers & Security*, 21(8):750–759, 2002.