CERIAS Tech Report 2006-09

DYNAMIC AND EFFICIENT KEY MANAGEMENT FOR ACCESS HIERARCHIES

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

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
Purdue University

Marina Blanton
Purdue University

Nelly Fazio
New York University

Keith B. Frikken
Purdue University

Abstract

Hierarchies arise in the context of access control whenever the user population can be modeled as a set of partially ordered classes (represented as a directed graph). A user with access privileges for a class obtains access to objects stored at that class and all descendant classes in the hierarchy. The problem of key management for such hierarchies then consists in assigning a key to each class in the hierarchy so that keys for descendant classes can be obtained via an efficient key derivation process.

We propose a solution to this problem with the following properties: (i) the space complexity of the public information is the same as that of storing the hierarchy; (ii) the private information at a class consists of a single key associated with that class; (iii) updates (i.e., revocations and additions) are handled *locally* in the hierarchy; (iv) the scheme is provably secure against collusion; and (v) each node can derive the key of any of its descendant with a number of symmetric-key operations bounded by 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. The security of our scheme is based on pseudo-random functions, without reliance on the Random Oracle Model.

Another substantial contribution of this work is that for trees, we achieve a worst- and average-case key-derivation time 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 obtained at the cost of only a constant factor in the space to store the hierarchy. We also show how to extend our techniques to more general hierarchies.

Finally, by making simple modifications to our scheme, we show how to handle extensions proposed by Crampton [2003] of the standard hierarchies to "limited depth" and reverse inheritance.

1 Introduction

1.1 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

*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. A preliminary version of this work appeared in the Proceedings of the ACM Conference on Computer and Communications Security (CCS) 2005.

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 ([21]; [43]) 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 (e.g., [23]), databases (e.g., [18]), and networking (e.g., [38]; [35]).

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 public-key cryptography but instead utilize only efficient techniques.

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 objects 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., Anderson and Kuhn [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 operations linear in the depth of the graph in the worst case (see Section 2 for more information about complexity of key derivation), which for some graphs is $O(n)$ where n is the number of nodes in the access graph. While the number of 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 that database. See also [37]; [41] 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 operations) with modest increase in public storage space.

1.2 Our Results

Our approach can support arbitrary access graphs, but in this work we consider only acyclic graphs.[1] In this work we describe two schemes: a base scheme and an extended scheme. The base scheme is simple and extremely efficient – it can be implemented using only hash functions. We show its provable security against key recovery. The second, extended, scheme provides higher security guarantees: we prove that user keys are now pseudo-random (i.e., indistinguishable from random). The scheme, however, relies on additional use of symmetric-key encryption. Other properties shared by both of the schemes are:

- 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.
- The derivation by a node of a descendant node's access key requires the number of 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 schemes, 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 schemes relies on the use of pseudo-random functions.

We also show that our solution 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 [5], 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

[1] Even though the scheme can be applied to graphs that contain cycles, we do not foresee a setting in which such access graphs are useful. That is, since all nodes comprising a cycle have identical privileges, they can be merged into a single node. Thus, in this work we restrict our attention to directed acyclic graphs.

the department of that employee); this model can cover a much richer set of access control policies than that of other schemes. We model these extensions after Crampton's work [16], and they do not increase the space or computational complexity of our schemes.

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 $O(\log \log n)$ operations in the worst case with constant increase in public information, and to only 3 operations with public space usage of $O(n \log \log n)$. We also describe how to use our techniques with more general hierarchies. These techniques allow us to achieve the fastest key derivation known to date.

1.3 Organization

We give an overview the literature on key management for access control in Section 2, while Section 3 contains a formal description of the problem. Section 4 presents our base scheme along with its security proof against key recovery. In Section 5 we present an extension of the base scheme, which is proven secure w.r.t. the stronger security notion of key indistinguishability. In Section 6, we describe how to deal with dynamic changes to the access graph, while Section 7 suggests extensions that permit the scheme's usage with other access models given in [16]. Section 8 presents our techniques to improve efficiency of key derivation for trees and more general hierarchies. Finally, Section 9 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]. Since then a large number of publications ([7, 8, 9, 11, 12, 13, 14, 17, 22, 25, 26, 29, 28, 30, 32, 33, 34, 36, 39, 40, 42, 44, 45, 47, 48, 53, 54, 55, 56] 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 [52, 51, 46, 49, 27] or incorrect [10]. Therefore, we do not take these schemes into consideration in our further discussion.

A significant number of schemes, e.g., [1, 36, 25, 8, 28, 26, 12, 39, 30, 40, 34, 45], operate large numbers computed as a product of up to $O(n)$ coprime numbers or, alternatively, up to $O(n)$ large numbers, where n is the number of nodes in the graph. Such numbers can grow to n bits long and are prohibitively large for most hierarchies. 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-key crypto operations. Our key derivation, on the other hand, even without efficiency improvements is bounded by the depth of the access hierarchy and can be implemented using $O(n)$ hash operations in the worst case (i.e., then the depth of the hierarchy is $O(n)$).

Work of [32, 42, 44] is limited to trees and thus is of limited use. Work of [7, 47, 53] is

concerned with a slightly different model having a hierarchy of users and a hierarchy of resources. The scheme of [7], however, is not dynamic; and in [47, 53] 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 [22] 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 [50] uses modular exponentiation, and additions/deletions require rekeying of all descendants. A number of schemes [17, 48, 9] are based on interpolating polynomials and give reasonable performance. In [48, 17], 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 [9], 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) [54, 55] 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 [54] require rekeying of all descendants and in [55] all descendants should be rekeyed when a node is deleted.

A number of recent schemes [11, 13, 14, 33, 56] use overall structure similar to ours and have performance comparable to our base scheme. [14], however, does not address dynamic changes, and the scheme is less efficient than ours because of additional usage of modular multiplication. [11] 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 [33] and [56], 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. [13] 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). Another very important distinction between the present work and these publications is that our scheme is provably secure. In addition, our extended scheme provides even stronger security guarantees (i.e., key indistinguishability) that have not been shown before. Techniques for improving efficiency are also an important contribution of this work.

Table 1 gives a comparison of our base scheme and other schemes. Private storage is measured per access class. Public storage is measured for the entire access graph (overhead introduced by the scheme, without information needed to represent the graph itself), and only the dominant term is given. The key derivation time shown reflects maximum computation needed to derive the key of node w given the key of node v, assuming there is a path of length ℓ between v and w.

In the table, k is a security parameter that corresponds to the size of the secret key (and in most cases is the size of the output produced by a cryptographic hash function H); k_1 is another security parameter (of comparable value); c_{H} denotes computation required by a single invocation

Scheme	Private storage	Public storage	Key derivation	Changes I/D/R	Proof of security
[33]	k	$2k\|E\|$	$(3c_{\mathsf{H}} + 4c_{\mathsf{XOR}})\ell$	L/NL/L	No
[56]	k	$(k + k_1)\|E\|$	$(c_{\mathsf{H}} + 2c_{\mathsf{XOR}})\ell$	L/NL/L	No
[13]	k	$k\|E\|$	$(c_{\mathsf{H}} + c_{\mathsf{XOR}})\ell$	L/NL/L	No
[11]	k	$k\|E\|$	$(c_{\mathsf{D}} + c_{\mathsf{H}} + c_{\mathsf{XOR}})\ell$	L/NL/L	No
Ours	k	$k\|E\|$	$(c_{\mathsf{H}} + c_{\mathsf{XOR}})\ell$	L/L/L	Yes

Table 1: Comparison with previous work.

of H[2]; c_{XOR} corresponds to computation needed to perform bitwise XOR of two strings; and c_{D} is computation needed for symmetric key decryption. In the table, changes to the hierarchy include insertion (I), deletion (D), and re-keying (R); L stands for "local" and NL for "non-local." In all of the schemes that list "non-local" for deletions, such operations require re-keying of all descendant classes in the hierarchy.

Note that in different schemes, the authors might make assumptions on what information is public and what is stored with the client, which differs from what we present here. For the sake of comparison, however, we unify the schemes and list their capabilities, which may or may not be different from the results reported by the authors. In addition, results of [33, 13] rely on tamper-resistance of the 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 brevity, 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 $o \in \mathcal{O}_j$ at class v_j. The function

[2]Our solution uses pseudo-random function F instead of using H directly. F, however, can be implemented using solely a hash function, and for the sake of uniformity we list c_{H} for our scheme as well.

$\mathcal{O}^* : V \to 2^O$ maps a node $v_i \in V$ to a set of objects accessible to a subject at class v_i (we use $\mathcal{O}_i^*$ as a shorthand for $\mathcal{O}^*(v_i)$); the function is defined as $\mathcal{O}_i^* = \bigcup_{v_j \in Desc(v_i)} \mathcal{O}_j$.

Intuitively, a key allocation mechanism aims at implementing such form of access control by assigning a cryptographic key k_i to each class v_i. Such key k_i is then used to guard access to objects of class v_i (for example, by encrypting object $o \in \mathcal{O}_i$ under key k_i), and is made available to every users at class v_i (and at any of its ancestor classes).

It follows that each user ought to store (or at least be able to derive) the cryptographic key k_i associated with the class v_i to which he belongs, as well as the keys k_j's of all classes v_j descendants of v_i. For the sake of generality, we do not impose any specific structure on the secret information actually stored by users at class v_i; we denote such information by S_i.

In summary, S_i denotes the secret information that each user at class v_i actually stores, while k_i (which is derivable from S_i) is the cryptographic key necessary to gain access to objects at class v_i.

We formalize the above intuition with the following definition.

Definition 3.1 *A Key Allocation (KA) scheme is a pair of polynomial-time algorithms* $(\mathsf{Set}, \mathsf{Derive})$*, defined as follows:*

- $\mathsf{Set}(1^\rho, G)$ *is a randomized algorithm that on input a security parameter* 1^ρ *and an access graph* G*, outputs two mappings: (i) a public mapping* $\mathsf{Pub} : V \cup E \to \{0,1\}^*$*, associating a public label* ℓ_i *to each node* v_i *and a public label* y_{ij} *to each edge* (v_i, v_j) *in the graph; (ii) a secret mapping* $\mathsf{Sec} : V \to \{0,1\}^\rho \times \{0,1\}^\rho$*, associating a secret information* S_i *and a cryptographic key* k_i *to each node* v_i *in* G*. (No secret information is associated to edges in* G*.)*

- $\mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i)$ *is a deterministic algorithm taking as input the access graph* G*, the public information* Pub *output by* Set*, a source node* v_i*, a target node* v_j *and the secret information* S_i *of node* v_i*. It outputs the cryptographic key* k_j *associated to node* v_j *if* $v_j \in Desc(v_i)$*, or a special rejection symbol* $\perp$ *otherwise.*

For correctness, the Set *and* Derive *algorithms of a Key Allocation scheme should also satisfy the following constraint:* $\forall v_i \in V$, $\forall v_j \in Desc(v_i)$,

$$\Pr\left[k_j = \mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i) \;\middle|\; \begin{array}{c} (\mathsf{Pub}, \mathsf{Sec}) \leftarrow \mathsf{Set}(1^\rho, G), \\ (S_i, k_i) \leftarrow \mathsf{Sec}(v_i), \\ (S_j, k_j) \leftarrow \mathsf{Sec}(v_j) \end{array} \right] = 1$$

where the probability is over the random choices of the Set *algorithm.*

We now formalize two levels of security: *Key Recovery* and *Key Indistinguishability.*

Definition 3.2 (Key Recovery) *A Key Allocation scheme is secure w.r.t. key recovery if no polynomial time adversary* $\mathcal{A}$ *has a non-negligible advantage (in the security parameter* ρ*) against the challenger in the following game:*

- **Setup***: The challenger runs* $\mathsf{Set}(1^\rho, G)$*, and gives the resulting public information* Pub *to the adversary* $\mathcal{A}$*.*

- **Attack***: The adversary issues, in any adaptively chosen order, a polynomial number of* $\mathsf{Corrupt}(v_i)$ *queries, which the challenger answers by retrieving* $(S_i, k_i) = \mathsf{Sec}(v_i)$ *and giving* S_i *to* $\mathcal{A}$*.*

- **Break**: *The adversary outputs a node v^*, subject to $v^* \notin Desc(v_i)$ for any v_i asked in* **Phase 1**, *along with her best guess k'_{v^*} to the cryptographic key k_{v^*} associated with node v^*.*

We define the adversary's advantage in attacking the scheme to be $\Pr[k'_{v^*} = k_{v^*}]$.

Definition 3.3 (Key Indistinguishability) *A Key Allocation scheme is key indistinguishable if no polynomial time adversary $\mathcal{A}$ has a non-negligible advantage (in the security parameter ρ) against the challenger in the following game:*

- **Setup**: *The challenger runs* $\mathsf{Set}(1^\rho, G)$, *and gives the resulting public information* Pub *to the adversary $\mathcal{A}$.*
- **Phase 1**: *The adversary issues, in any adaptively chosen order, a polynomial number of* $\mathsf{Corrupt}(v_i)$ *queries, which the challenger answers by retrieving* $(S_i, k_i) = \mathsf{Sec}(v_i)$ *and giving S_i to $\mathcal{A}$.*
- **Challenge**: *Once the adversary decides that* **Phase 1** *is over, it specifies a node v^*, subject to $v^* \notin Desc(v_i)$ for any v_i asked in* **Phase 1**. *The challenger picks a random bit $b^* \in \{0,1\}$: if $b^* = 0$, it returns to $\mathcal{A}$ the cryptographic key k_{v^*} associated with node v^*; otherwise, it returns to $\mathcal{A}$ a random key $\bar{k}_{v^*}$ of the same length ρ.*
- **Phase 2**: *The adversary can issue more* $\mathsf{Corrupt}(v_i)$ *queries, obtaining back the corresponding key S_i. Note that $\mathcal{A}$ cannot ask* $\mathsf{Corrupt}(v_i)$ *queries for $v_i \in Anc(v^*)$.*
- **Guess**: *The adversary outputs a bit $b \in \{0,1\}$ as her best guess to whether she was given the actual key k_{v^*} or a random key. $\mathcal{A}$ wins the game if $b = b^*$.*

We define the adversary's advantage in attacking the scheme to be $|\Pr[b = b^*] - \frac{1}{2}|$.

Remark. In formalizing the security of a key allocation scheme, $\mathsf{Corrupt}$ queries are answered with respect to the secret info S_i, whereas the **Break**/**Challenge** phases relate to the cryptographic key k_{v^*}. This is because access to an object at class v^* is granted by the cryptographic key k_{v^*}; thus, to 'test' the ability of the adversary to break the access control mechanism, we challenge her to either recover the real cryptographic key (for *Key Recovery*) or to tell the real cryptographic key apart from some random string (for *Key Indistinguishability*).

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 ℓ levels below it can be done in ℓ evaluations of a pseudo-random function, which can be implemented as, e.g., HMAC [6] built using only a cryptographic hash function. Here we focus on key allocations for a static access hierarchy. An extension of this scheme is given in Section 5, and its support for dynamic access hierarchies is discussed in Section 6.

Our construction is based on the use of pseudo-random functions:

Definition 4.1 (Pseudo-Random Function (PRF) Family) *Let $\{F^\rho\}_{\rho\in\mathbb{N}}$ be a family of functions where $F^\rho : K^\rho \times D^\rho \to R^\rho$. For $k \in K^\rho$, denote by $F_k^\rho : D^\rho \to R^\rho$ the function defined by $F_k^\rho(x) \doteq F^\rho(k, x)$. Let $Rand^\rho$ denote the family of all functions from D^ρ to R^ρ,* i.e., *$Rand^\rho \doteq \{g \mid g : D^\rho \to R^\rho\}$.*

*Let $A(1^\rho)$ be an algorithm that takes as oracle a function $g : D^\rho \to R^\rho$, and returns a bit. Function g is either drawn at random from $Rand^\rho$ (*i.e., *$g \xleftarrow{r} Rand^\rho$), or set to be F_k^ρ, for a random $k \xleftarrow{r} K^\rho$. Consider the two experiments:*

Experiment $\boldsymbol{Exp}_{F,A}^{PRF-1}(\rho)$	*Experiment* $\boldsymbol{Exp}_{F,A}^{PRF-0}(\rho)$
$k \xleftarrow{r} K^\rho$	$g \xleftarrow{r} Rand^\rho$
$d \leftarrow A^{F_k^\rho}(1^\rho)$	$d \leftarrow A^g(1^\rho)$
Return d	*Return* d

The PRF-advantage of A is then defined as:

$$\mathsf{Adv}_{F,A}^{PRF}(\rho) = \mid \Pr[\boldsymbol{Exp}_{F,A}^{PRF-1}(\rho) = 1] - \Pr[\boldsymbol{Exp}_{F,A}^{PRF-0}(\rho) = 1] \mid .$$

$\{F^\rho\}_{\rho\in\mathbb{N}}$ is a PRF family if for every $\rho \in \mathbb{N}$, the function F^ρ is computable in time polynomial in ρ, and if the function $\mathsf{Adv}_{F,A}^{PRF}(\rho)$ is negligible (in ρ) for every polynomial-time distinguisher $A(1^\rho)$ that halts in time poly(ρ).

Assume that we are given a PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$ where $F^\rho : \{0,1\}^\rho \times \{0,1\}^\rho \to \{0,1\}^\rho$.[3] Given an access graph $G = (V, E)$ and a security parameter ρ, the $\mathsf{Set}(1^\rho, G)$ algorithm proceeds as follows:

- For each vertex $v_i \in V$, pick a random label $\ell_i \in \{0,1\}^\rho$ and a random value $S_i \in \{0,1\}^\rho$, and set $k_i \doteq S_i$. An entity that is assigned access levels $V' \subseteq V$ is given all keys for their access levels $v_j \in V'$.
- For each edge $(v_i, v_j) \in E$, compute $y_{ij} \doteq k_j \oplus F(k_i, \ell_j)$.

The output of $\mathsf{Set}(1^\rho, G)$ consists of the two mappings $\mathsf{Pub} : V \cup E \to \{0,1\}^*$ and $\mathsf{Sec} : V \to \{0,1\}^\rho \times \{0,1\}^\rho$, defined as:

$$\mathsf{Pub} : v_i \mapsto \ell_i \qquad \mathsf{Pub} : (v_i, v_j) \mapsto y_{ij}$$
$$\mathsf{Sec} : v_i \mapsto (S_i, k_i)$$

We now describe the Derive algorithm. To obtain the cryptographic key k_j of a descendant v_j, a node v_i sequentially processes every edge $(v_{\bar{\imath}}, v_{\bar{\jmath}})$ on the path between v_i and v_j. Given an edge $(v_{\bar{\imath}}, v_{\bar{\jmath}})$ for which both $v_{\bar{\imath}}$'s private key $k_{\bar{\imath}}$ and the stored public information $\ell_{\bar{\jmath}}$ and $y_{\bar{\imath}\bar{\jmath}}$ are known, $v_{\bar{\imath}}$ can generate $v_{\bar{\jmath}}$'s private information $k_{\bar{\jmath}}$ thanks to the fact that $y_{\bar{\imath}\bar{\jmath}}$, is defined as $y_{\bar{\imath}\bar{\jmath}} \doteq k_{\bar{\jmath}} \oplus F(k_{\bar{\imath}}, \ell_{\bar{\jmath}})$.

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.

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 $F(k_1, \ell_2)$ 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 F function.

[3] To simplify the notation, we will omit the superscript ρ from F^ρ wherever the security parameter is clear by the context.

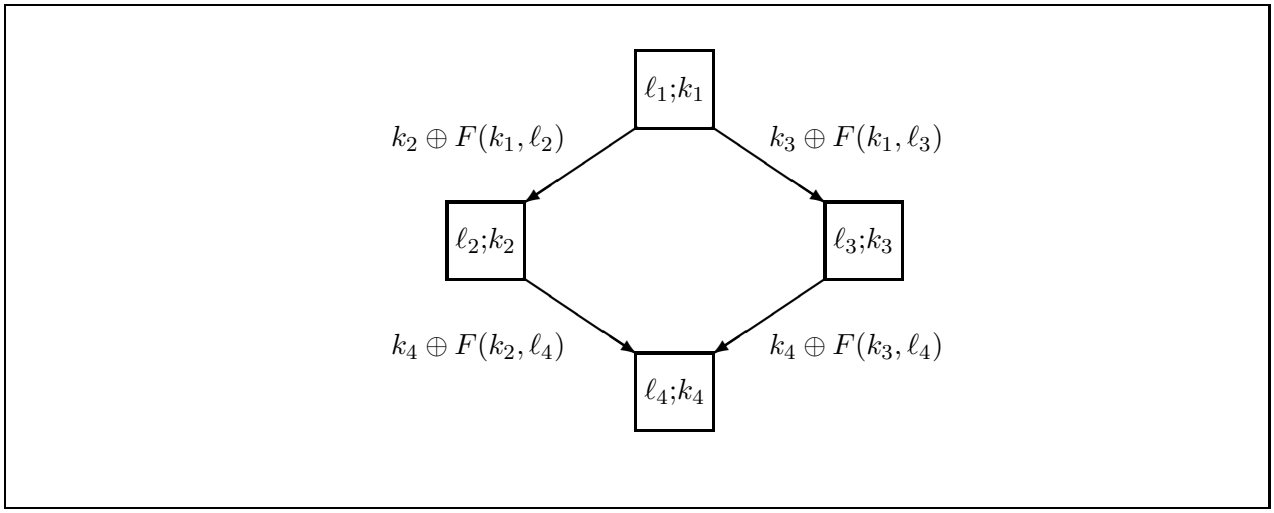

Figure 1: Key allocation for example access graph.

Theorem 4.2 *The above scheme is secure against key-recovery (*c.f. *Definition 3.2) for any directed acyclic graph (DAG) G, assuming the security of the pseudo-random function family $\{F^\rho\}_{\rho\in\mathbb{N}}$ (*c.f. *Definition 4.1).*

Proof: In the security proof, we will follow the same structural approach used in [19], first advocated in [15]. Starting from the actual attack scenario, we consider a sequence of hypothetical games, all defined over the same probability space. In each game, the adversary's view is obtained in different ways, but its distribution is still indistinguishable among the games.

Roughly speaking, proving the theorem amounts to showing that the only way to break the key recovery security of the base scheme of Section 4 is by breaking the pseudo-random function F. To this aim, we need to show how to turn an adversary $\mathcal{A}$ attacking the scheme into an adversary $\mathcal{B}_F$ attacking F.

One difficulty with this approach is that whereas $\mathcal{A}$ can choose which part of the public info to attack (via the challenge query), the adversaries $\mathcal{B}_F$ does not have such flexibility. The standard way to solve this technical problem is to "guess" the node v^* for which adversary $\mathcal{A}$ will ask the challenge query and construct adversaries $\mathcal{B}_F$ based on the assumption that this guess is correct.

In the rest of the proof, we will assume that we correctly guessed the challenge node v^*. Since such *a priori* guess is correct with $1/n$ chance, this affects the exact security of the reduction proof by a factor of n.

Let $G' = (V', E')$ be the subgraph of G induced by restricting the set of vertices V to the set V' of the ancestors of v^*, including v^* itself. Let $v_1, \ldots, v_h \equiv v^*$ be any topological ordering of the vertices in G'.

To prove the theorem, we define a sequence of "indistinguishable" games $\mathbf{G}_0$, $\mathbf{G}_1$, ..., $\mathbf{G}_h$, all operating over the same underlying probability space. Starting from the actual adversarial game $\mathbf{G}_0$ (as defined in Definition 3.2), we incrementally make slight modifications to the behavior of the challenger, thus changing the way the adversary's view is computed, while maintaining the views' distributions indistinguishable among the games. In the last game, it will be clear that the adversary has (at most) a negligible advantage; by the indistinguishability of any two consecutive games, it will follow that also in the original game the adversary's advantage is negligible. Recall

that in each game $\mathbf{G}_j$, the goal of adversary $\mathcal{A}$ is to guess the cryptographic key k_{v^*} associated with node v^*. Let T_j be the event that $k'_{v^*} = k_{v^*}$ in game $\mathbf{G}_j$.

Game $\mathbf{G}_0$. Define $\mathbf{G}_0$ to be the original game as described in Definition 3.2.

Game $\mathbf{G}_1$. This game is identical to game **Game $\mathbf{G}_0$**, except that in $\mathbf{G}_1$ the $\mathsf{Set}(1^\rho, G)$ algorithm is modified in such a way that the secret key k_{v_1} of node v_1 is never used in the creation of the public information. Instead, for each edge (v_1, v_j) in the graph G coming out of node v_1, the public information y_{1j} associated with the edge (v_1, v_j) is selected at random from $\{0,1\}^\rho$, *i.e.*, $y_{1j} \xleftarrow{r} \{0,1\}^\rho$.

Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F(k_{v_1}, \cdot)$ in $\mathbf{G}_0$ with a truly random function. Since k_{v_1} does not occur anywhere else in the attack game, such modification is warranted by the security of the PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$. In other words, using a standard reduction argument, any non-negligible difference in behavior between game $\mathbf{G}_0$ and $\mathbf{G}_1$ can be used to construct a PPT algorithm $\mathcal{B}_F$ that is able to break the pseudo-random function F with non-negligible advantage. Hence,

$$\big|\Pr[T_1] - \Pr[T_0]\big| \leq \epsilon_{PRF} \tag{1}$$

where ϵ_{PRF} is the (negligible) advantage $\mathsf{Adv}^{PRF}_{F,B_F}(\rho)$ of any PPT adversary B_F against the security of the pseudo-random function F.

We now generalize the description of game $\mathbf{G}_1$ to any game in the sequence $\mathbf{G}_1, \ldots, \mathbf{G}_h$.

Game $\mathbf{G}_i$ $(1 \leq i \leq h)$. This game is identical to game $\mathbf{G}_{i-1}$, except that the $\mathsf{Set}(1^\rho, G)$ algorithm is modified in such a way that the secret key k_{v_i} of node v_i is never used in the creation of the public information. Observe that the cryptographic key k_{v_i} occurs in game $\mathbf{G}_{i-1}$ only as the key to the pseudo-random function $F(\cdot,\cdot)$. In particular, no information about k_{v_i} is present in the public information associated with the edges going into node v_i thanks to the modifications carried out in games $\mathbf{G}_0, \ldots, \mathbf{G}_{i-1}$, and to the fact that we are working through the ancestors of v^* in the topological ordering.

Thus, to change game $\mathbf{G}_{i-1}$ into game $\mathbf{G}_i$, for each edge (v_i, v_j) coming out of node v_i, we draw the public information y_{ij} at random from $\{0,1\}^\rho$ (rather than computing it as $y_{ij} \doteq F(k_{v_i}, \ell_j)$). Such modification amounts to substituting all occurrences of $F(k_{v_i}, \cdot)$ in $\mathbf{G}_{i-1}$ with a truly random function. Since k_{v_i} does not occur anywhere else in $\mathbf{G}_{i-1}$, we can conclude (as above) that such modification is warranted by the security of the PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$, *i.e.*:

$$\big|\Pr[T_i] - \Pr[T_{i-1}]\big| \leq \epsilon_{PRF} \tag{2}$$

To conclude the proof, observe that no information about the secret key $k_{v^*} (= k_{v_h})$ is present in the adversary's view for game $\mathbf{G}_h$. It follows that the probability of a correct guess for k_{v^*} by the adversary in game $\mathbf{G}_h$ is just $1/2^\rho$, *i.e.*:

$$\Pr[T_h] = \frac{1}{2^\rho} \tag{3}$$

Combining Equation 3 with the intermediate results in Equations 2, we can conclude that

$$\Pr[T_0] \leq \frac{1}{2^\rho} + h \cdot \epsilon_{PRF}.$$

■

5 The Extended Scheme

We now present an extension of the scheme described in Section 4 and prove it secure w.r.t. Key Indistinguishability (see Definition 3.3), without random oracles.

The scheme maintains essentially the same parameters as the one in Section 4: Every node stores only one random ρ-bit number; the public information is linear in the size of the access graph G; to derive the key of a descendant node located ℓ levels below, each node performs ℓ evaluations of a pseudo-random function $F : \{0,1\}^\rho \times \{0,1\}^* \to \{0,1\}^\rho$ (*cf.* Definition 4.1). Additionally, the extended scheme makes use of a secure[4] encryption scheme $\mathcal{E}$: we denote with Enc and Dec the corresponding encryption and decryption algorithms.

In details, given an access graph $G = (V, E)$ and a security parameter ρ, the $\mathsf{Set}(1^\rho, G)$ algorithm proceeds as follows:

- For each vertex $v_i \in V$, first pick a random label $\ell_i \in \{0,1\}^\rho$ and a random value $S_i \in \{0,1\}^\rho$; then compute $t_i \doteq F_{S_i}(0||\ell_i)$ and $k_i \doteq F_{S_i}(1||\ell_i)$.
- For each edge $(v_i, v_j) \in E$, compute $r_{ij} \doteq F_{t_i}(\ell_j)$ and $y_{ij} \doteq \mathsf{Enc}_{r_{ij}}(t_j||k_j)$.

The output of $\mathsf{Set}(1^\rho, G)$ consists of the two mappings $\mathsf{Pub} : V \cup E \to \{0,1\}^*$ and $\mathsf{Sec} : V \to \{0,1\}^\rho \times \{0,1\}^\rho$, defined as:

$$\mathsf{Pub} : v_i \mapsto \ell_i \qquad \mathsf{Pub} : (v_i, v_j) \mapsto y_{ij}$$
$$\mathsf{Sec} : v_i \mapsto (S_i, k_i)$$

We now describe the Derive algorithm. Given G, the public information Pub, a source node v_i, a target node v_j and the secret information S_i of node v_i, it derives the cryptographic key k_j of node v_j by considering each edge on the path[5] from v_i down to v_j in turn, and repeatedly decrypting the public info associated to such edge. More precisely, $\mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i)$ proceeds as follows:

- If there is no path from v_i to v_j in G, return $\perp$;
- If $i = j$, retrieve ℓ_i from Pub and return $k_j \leftarrow F_{S_i}(1||\ell_i)$;
- Else, compute $t_i \leftarrow F_{S_i}(0||\ell_i)$ and let $\bar{\imath} \doteq i$ and $t_{\bar{\imath}} \doteq t_i$; then

  repeat
  let $\bar{\jmath}$ be the successor of $\bar{\imath}$ in the path from v_i to v_j;
  retrieve $\ell_{\bar{\jmath}}$ and $y_{\bar{\imath}\bar{\jmath}}$ from Pub;
  $r_{\bar{\imath}\bar{\jmath}} \leftarrow F_{t_{\bar{\imath}}}(\ell_{\bar{\jmath}})$;
  $t_{\bar{\jmath}}||k_{\bar{\jmath}} \leftarrow \mathsf{Dec}_{r_{\bar{\imath}\bar{\jmath}}}(y_{\bar{\imath}\bar{\jmath}})$;
  $\bar{\imath} \leftarrow \bar{\jmath}$; $\quad t_{\bar{\imath}} = t_{\bar{\jmath}}$;
  until $\bar{\jmath} = j$;
  return k_j.

Figure 2 shows how the key derivation mechanism works for the same toy example given in Figure 1.

[4] We require the encryption scheme to be chosen ciphertext secure; see the definition in [20]

[5] If there is more than one path, pick one arbitrarily, *e.g.*, the shortest path from v_i to v_j.

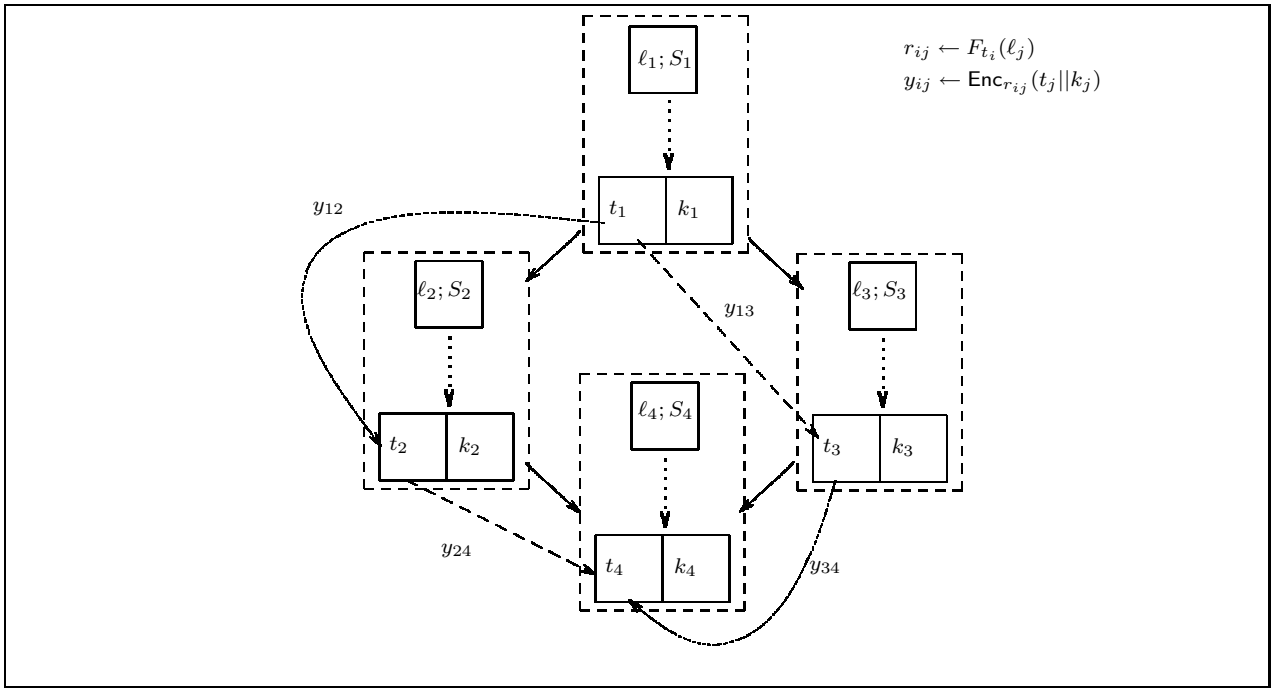

Figure 2: Key allocation for extended example access graph.

We now prove that the extended scheme described in this section is key indistinguishable (*cf.* Definition 3.3), following the same approach as in the proof of Theorem 4.2.

Theorem 5.1 *The above extended scheme is key indistinguishable for any directed acyclic graph G, assuming the security of the pseudo-random function family $\{F^\rho\}_{\rho\in\mathbb{N}}$ and the security of the encryption scheme $\mathcal{E}$.*

Proof: Roughly speaking, proving the theorem amounts to showing that the only way to break the key indistinguishability of the extended scheme of Section 5 is by either breaking the pseudo-random function F or the encryption scheme $\mathcal{E}$. To this aim, we need to show how to turn an adversary $\mathcal{A}$ attacking the scheme into either an adversary $\mathcal{B}_F$ attacking F or an adversary $\mathcal{B}_\mathcal{E}$ attacking $\mathcal{E}$.

One difficulty with this approach is that whereas $\mathcal{A}$ can choose which part of the public info to attack (via the challenge query), the adversaries $\mathcal{B}_F$ and $\mathcal{B}_\mathcal{E}$ do not have such flexibility. As noted in Theorem 4.2, the standard way to solve this technical problem is to "guess" the node v^* for which adversary $\mathcal{A}$ will ask the challenge query and construct adversaries $\mathcal{B}_F$ (or $\mathcal{B}_\mathcal{E}$) based on the assumption that this guess is correct.

In the rest of the proof, we will assume that we correctly guessed the challenge node v^*. Since such *a priori* guess is correct with $1/n$ chance, this affects the exact security of the reduction proof by a factor of n.

To prove the theorem, we again define a sequence of "indistinguishable" games $\mathbf{G}_0, \mathbf{G}_1, \ldots$, where $\mathbf{G}_0$ is the actual adversarial game (as defined in Definition 3.3), and where the adversary's advantage in the last game will only be negligible. Recall that in each game $\mathbf{G}_j$, the goal of adversary $\mathcal{A}$ is to output $b \in \{0,1\}$ which is her best guess to the bit b^* chosen by the challenger in the attack game described in Definition 3.3. Let T_j be the event that $b = b^*$ in game $\mathbf{G}_j$.

For clarity of exposition, we first discuss two special cases, which exemplify the most technical aspects of the proof. Afterwards, we describe how to tackle the general case.

First special case. *v^* is one of the roots[6] in G.*

Game $\mathbf{G}_0$. Define $\mathbf{G}_0$ to be the original game as described in Definition 3.3.

Game $\mathbf{G}_1$. This game is identical to game $\mathbf{G}_0$, except that in $\mathbf{G}_1$ the $\mathsf{Set}(1^\rho, G)$ algorithm is modified in such a way that the cryptographic key k_{v^*} is information theoretically hidden from the view of adversary $\mathcal{A}$. To this aim, we compute

$$t_{v^*} \leftarrow R_1(0||\ell_{v^*}), \qquad k_{v^*} \leftarrow R_1(1||\ell_{v^*})$$

where $R_1 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function.

Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{S_{v^*}}(\cdot)$ with a truly random function $R_1(\cdot)$. Since S_{v^*} does not occur anywhere else in the attack game, such modification is warranted by the security of a pseudo-random function. In other words, using a standard reduction argument, any non-negligible difference in behavior between game $\mathbf{G}_0$ and $\mathbf{G}_1$ can be used to construct a PPT algorithm $\mathcal{B}_F$ that is able to break the pseudo-random function F with non-negligible advantage. Hence,

$$\big|\Pr[T_1] - \Pr[T_0]\big| \leq \epsilon_{PRF} \tag{4}$$

where ϵ_{PRF} is the (negligible) advantage $\mathsf{Adv}^{PRF}_{F,B_F}$ of any PPT adversary B_F against the security of the pseudo-random function F.

It remains to notice that in game $\mathbf{G}_1$, the challenge no longer contains any information about b^*. This is because k_{v^*} is now a random value, exactly as $\bar{k}_{v^*}$. Moreover, since v^* is a root of G, it has no incoming edges and thus the public info Pub does not contain any label y_{iv^*} (which would be an encryption of $t_{v^*}||k_{v^*}$). Therefore, k_{v^*} is independent of any other info in the adversary view, and thus it is indistinguishable from $\bar{k}_{v^*}$. It follows that the adversary's view is exactly the same regardless of the value of b^*, and thus:

$$\Pr[T_1] = 1/2 \tag{5}$$

Combining Equations 4 and 5, the thesis follows.

Second special case. *v^* has a single predecessor p which is one of G's roots.*

Game $\mathbf{G}_0$, Game $\mathbf{G}_1$. The first two games are defined as in the first special case.

Game $\mathbf{G}_2^{(a)}$. This game is identical to game $\mathbf{G}_1$, except that in **Game** $\mathbf{G}_2^{(a)}$ we further modify the $\mathsf{Set}(1^\rho, G)$ algorithm so that the secret information t_p is information theoretically hidden from the view of adversary $\mathcal{A}$. To this aim, we compute

$$t_p \leftarrow R_1(0||\ell_p), \qquad k_p \leftarrow R_1(1||\ell_p)$$

where $R_1 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function.

Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{S_p}(\cdot)$ with a truly random function $R_1(\cdot)$. Since S_p does not occur anywhere else

[6] By root in a DAG we mean any minimal node in the topological order of G.

in the attack game, such modification is warranted by the security of a pseudo-random function; hence,

$$\left| \Pr[T_2^{(a)}] - \Pr[T_1] \right| \leq \epsilon_{PRF} \tag{6}$$

.

Game $\mathbf{G}_2^{(b)}$. To turn game $\mathbf{G}_2^{(a)}$ into game $\mathbf{G}_2^{(b)}$, for any child s of p, we compute

$$r_{ps} \leftarrow R_2(\ell_s)$$

where $R_2 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function.

Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{t_p}(\cdot)$ with a truly random function $R_2(\cdot)$, which is safe since p is a root of G, and thus t_p does not occur anywhere else in the adversarial view (in particular, it is not encrypted within any label in Pub). Therefore, using a standard reduction argument, any non-negligible difference in behavior between game $\mathbf{G}_1$ and $\mathbf{G}_2^{(b)}$ can be used to construct a PPT algorithm $\mathcal{B}_F$ that is able to break the pseudo-random function F with non-negligible advantage. Hence,

$$\left| \Pr[T_2^{(b)}] - \Pr[T_2^{(a)}] \right| \leq \epsilon_{PRF} \tag{7}$$

Game $\mathbf{G}_2^{(c)}$. This game is exactly as $\mathbf{G}_2^{(b)}$ except that the label y_{pv^*} associated with edge $(p, v^*) \in E$ is now computed as

$$y_{pv^*} \leftarrow \mathsf{Enc}_{r_{pv^*}}(\$||\$)$$

where \$ denotes a random value.

Note that this modification amounts to changing the plaintext within a ciphertext, which was encrypted under a key that is independent from the adversary view (thanks to the changes in game $\mathbf{G}_2^{(b)}$). Therefore, using a standard reduction argument, any non-negligible difference in behavior between games $\mathbf{G}_2^{(b)}$ and $\mathbf{G}_2^{(c)}$ can be used to construct a PPT algorithm $\mathcal{B}_\mathcal{E}$ that is able to break the security of the encryption scheme $\mathcal{E}$ with non-negligible advantage. Hence,

$$\left| \Pr[T_2^{(c)}] - \Pr[T_2^{(b)}] \right| \leq \epsilon_{Enc} \tag{8}$$

where ϵ_{Enc} is the (negligible) advantage of any PPT adversary against the security of the encryption scheme $\mathcal{E}$.

It remains to notice that in game $\mathbf{G}_2^{(c)}$, the challenge no longer contains any information about b^*. This is because, thanks to the changes in this game, the label y_{pv^*} of the only incoming edge $(p, v^*) \in E$ no longer contains any info about k_{v^*}, which is therefore independent from the adversary view. Thus,

$$\Pr[T_2^{(c)}] = 1/2 \tag{9}$$

Combining Equations 4, 6, 7, 8 and 9, the thesis follows.

The general case.
The second special case demonstrated how to "purge" the adversary view from the information on k_{v^*} (which could be leaked by the label y_{pv^*} in Pub associated with the single edge (p, v^*) going into v^*). In the general case, there could be several edges going into v^*, and in particular it is necessary to consider each path going from one of the roots of G into v^*.

To this aim, we start the sequence of games with games **Game** $\mathbf{G}_0$ and **Game** $\mathbf{G}_1$, defined as in the first special case; then for each of the ancestor of v^* (considered in turn according to any topological sorting), we introduce three games mimicking the structure of games $\mathbf{G}_2^{(a)}$, $\mathbf{G}_2^{(b)}$ and $\mathbf{G}_2^{(c)}$ as defined in the second special case.

At a high level, this can be thought of as a "pebbling argument", by which we successively pebble all the ancestors of v^*, until we reach v^* itself, according to the following rules:

1. A node can be pebbled only after all its ancestors have already been pebbled.
2. To pebble a node u, we introduce the games $\mathbf{G}_u^{(a)}$, $\mathbf{G}_u^{(b)}$ and $\mathbf{G}_u^{(c)}$ in the sequence, following the same approach employed in the second special case. In particular, first we define a game $\mathbf{G}_u^{(a)}$ in which the secret information t_u is computed as:

$$t_u \leftarrow R_u^{(a)}(0||\ell_u), \qquad k_u \leftarrow R_u^{(a)}(1||\ell_u)$$

where $R_u^{(a)} : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function.

Second, we define a game $\mathbf{G}_u^{(b)}$ in which, for every child s of u, we compute

$$r_{us} \leftarrow R_u^{(b)}(\ell_s)$$

where $R_u^{(b)} : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function.

Third, we define a game $\mathbf{G}_u^{(c)}$ in which we set

$$y_{us^*} \leftarrow \mathsf{Enc}_{r_{us^*}}(\$||\$)$$

where s^* is the successor of u in the path from u to v^* and \$ denotes a random value.

Reasoning along the lines of the argument for the second special case, we can argue that each tuple of games $\mathbf{G}_u^{(a)}$, $\mathbf{G}_u^{(b)}$ and $\mathbf{G}_u^{(c)}$ negligibly alter the adversary view (by a term $2\epsilon_{PRF} + \epsilon_{Enc}$). Overall, once all the ancestors of v^* have been pebbled, we can argue that no info about k_{v^*} is present in Pub, and hence k_{v^*} is independent from the adversary view, and it is thus indistinguishable from $\bar{k}_{v^*}$. From this we can derive that in the last game $\mathbf{G}_{v^*}^{(c)}$,

$$\Pr[T_{v^*}^{(c)}] = 1/2 \tag{10}$$

Combining all the intermediate equations, we can conclude that

$$\Pr[T_0] \leq 1/2 + \epsilon_{PRF} + n_{v^*}(2\epsilon_{PRF} + \epsilon_{Enc})$$

where n_{v^*} is the number of ancestors of v^*. This concludes the proof. ■

6 Supporting Changes to the Access Hierarchy

In this section we show how dynamic changes to the access hierarchy, such as addition and deletion of edges and nodes, as well as replacing a node's key, are handled in the scheme of Section 5.

Insertion of an edge. Suppose the edge (v_i, v_j) is to be inserted into G. First, compute $r_{ij} \doteq F_{t_i}(\ell_j)$ and $y_{ij} = \mathsf{Enc}_{r_{ij}}(t_j||k_j)$. Then, augment Pub to contain the mapping $(v_i, v_j) \mapsto y_{ij}$.

Deletion of an edge. In deleting an edge, the difficulty is in preventing access by ex-members. Suppose the edge (v_i, v_j) is to be deleted from G. Then the following updates are done: for each node $v_h \in Desc(v_j, G)$, perform:

1. Change the label of v_h, call it ℓ'_h. Note that S_h remains unchanged, but the keys t_h and k_h need to be recomputed as $t'_h \doteq F_{S_h}(0||\ell'_h)$ and $k'_h \doteq F_{S_h}(1||\ell'_h)$.

2. For each edge (v_p, v_h) where $v_p \in Pred(v_h)$, update the value of y_{ph} to be an encryption of the newly compute keys, *i.e.*, $y'_{ph} \doteq \mathsf{Enc}_{r_{ph}}(t'_h||k'_h)$, where $r'_{ph} \doteq F_{t_p}(\ell'_h)$.

Insertion of a new node. If a new node v_i is inserted, together with new edges into and out of it, then we do the following:

1. Create the node v_i without any incoming or outgoing edges; this requires just generating a random public label $\ell_i \in \{0,1\}^\rho$ and a random secret value $S_i \in \{0,1\}^\rho$, computing $k_i \doteq F_{S_i}(1||\ell_i)$ and augmenting Pub with the mapping $v_i \mapsto \ell_i$ and Sec with the mapping $v_i \mapsto (S_i, k_i)$.

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

Deletion of a node. Deletion of a node amounts to the following two steps:

1. Deletion of all the edges coming into and out of v_i, using the above procedure for edge-deletions.

2. Removal of the public and secret information associated with v_i from the maps Pub and Sec.

Key replacement. Key replacement for a node v_i is performed as follows:

1. Update the secret information S_i with a new random value $S'_i \xleftarrow{r} \{0,1\}^\rho$.

2. Update the vertex's keys to $t'_i \doteq F_{S'_i}(0||\ell_i)$ and $k'_i \doteq F_{S'_i}(1||\ell_i)$.

3. Update Sec to map $v_i \mapsto (S'_i, k'_i)$.

4. For each edge (v_j, v_i) (i.e., where $v_j \in Pred(v_i)$), compute y'_{ji} according to the new keys t'_i and k'_i and updates Pub to map $(v_j, v_i) \mapsto y'_{ji}$.

5. For each edges (v_i, v_l) (i.e., where $v_l \in Succ(v_i)$), compute y'_{il} according to the new key t'_i and update Pub to map $(v_i, v_l) \mapsto y'_{il}$.

No node other than v_i 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, [56] considers only a hierarchy of security classes without mentioning individual users, and [33] 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 class(es), and for all descendants of this access class(es) 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 be rekeyed.

2. Make the access 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 classes. By creating such a graph, removing a user is as easy as removing his node, and thus does not require rekeying.

7 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 [16] 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" DAGs (defined later) when there is no collusion.

7.1 Downward Inheritance

To support such inheritance, 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 or downwards.

7.2 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 $F(k_i, k_L)$. 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 Improving Efficiency

As the scheme described in the previous sections supports any access 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 (and then more general hierarchies) 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. Throughout this section we assume that the access structure is a tree with n nodes, unless mentioned otherwise. Sections 8.1–8.3 describe our first approach that reduces the path between any two nodes to $O(\log \log n)$ with $O(n)$ public space, and Section 8.4 describes an alternative approach that results in any two nodes being at most 3 edges away with $O(n \log \log n)$ public storage space. Then Section 8.5 addresses dynamic behavior, and Section 8.6 extends the techniques to more general hierarchies.

We also would like to mention that a recent paper [4] provides an alternative way of reducing the distance between any two access classes in the hierarchy. Neither approach, however, is superior to the other. That is, the techniques of [4] for trees result in the distance of at most five edges for nodes with $O(n \log n)$ public space, while solutions presented in this work provide better performance. The approach of [4], on the other hand, has the advantage that it can easily applied to any graph of dimension d, for which a d-tuple representation can be constructed.

8.1 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 $n/2$ [31]. 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.

8.2 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, [24]).

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 $\mathsf{Length}(n)$ 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(T).

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(T).) 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 $\leq \mathsf{Length}(|T_i|)$, which is $\leq \mathsf{Length}(\sqrt{|T|})$. 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 $\leq \mathsf{Length}(|T_i|)$, which is at most $\mathsf{Length}(\sqrt{|T|})$. 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(T). 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(T).

(f) Return $\mathcal{P}$.

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

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

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

8.3 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 8.2 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 non-distinguished 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 8.2 (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.

8.4 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 8.2, 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(T) algorithm of Section 8.2.

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 8.2; thus this scheme adds only $O(n\log\log n)$ edges. Furthermore, the Algorithm FindPath(x, y, T) is very similar to the Section 8.2 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(T)). 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.

8.5 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 σ such operations, if t_σ 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 αn insertions/deletions. The cost is small and is only O(1) per operation because the $O(n)$ time it takes for one shortcut re-computation is amortized over the αn 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 re-defining 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 recomputation 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.

8.6 More General Hierarchies

In this section, we extend the shortcut techniques beyond tree hierarchies and introduce an algorithm for adding shortcut edges to general access graphs. This algorithm can be applied to any hierarchy and addition of shortcuts results in key derivation being at most $O(\log \log n)$ steps. The algorithm, however, guarantees efficient storage only for certain hierarchies; specifically, if the number of nodes with multiple parents is relatively small, for instance, $\sqrt{n}$. We believe that this limited notion of an access graph captures most real life access hierarchies.

Suppose we are given a transitively-reduced access graph $G = (V, E)$. Let T denote the set of nodes in G with more than one parent (these nodes are viewed as "troublesome" because of more than one parent). Define the set of edges E_T to be the set of edges in E that are incident on one or more nodes in T. Now we create two sets of "shortcut" edges.

1. Invoke the previous technique of Section 8.3 on the graph $(V - T, E - E_T)$. Note that this graph is a forest of trees (since all "troublesome" nodes have been removed). Let E_1 denote the set of edges $E - E_T$. By the correctness of the previous scheme, any ancestry relation in this graph is captured by a path of length $O(\log \log n)$ and $|E_1| = O(n)$.

2. Add a set of edges, call it E_2, that form the transitive closure of T. That is, if given two nodes $t_1, t_2 \in T$, where $t_1 \neq t_2$, $t_1 \in Anc(t_2, G)$, and $(t_1, t_2) \notin E$, then add edge (t_1, t_2) to E_2.

The shortcut edges added to G are $E_1 \cup E_2$, i.e., the new graph is $G' = (V, E \cup E_1 \cup E_2)$. We now show that this construction satisfies the necessary properties.

Lemma 8.1 *If there is path from node x to node y in G', then there is a path from x to y in G.*

Proof: We must show that every added edge is part of the transitive closure of G. By the correctness of the previous scheme, this is true for every edge in E_1. Furthermore, it is clearly true for edges in E_2, since the edges are added to E_2 only when the source is an ancestor of the destination. ■

Lemma 8.2 *If there is a path from x to y in G, then there is a path from x to y in G' of length $O(\log \log n)$.*

Proof: There are 3 cases to consider:

Case 1: There is a path from x to y with no troublesome nodes on the path (including x and y). In this case, there will be a path from x to y in $(V - T, E - E_T)$, and thus the shortcuts in E_1 will create a path with length $O(\log \log n)$.

Case 2: All paths from x to y contain at least one troublesome node, but there is a path from x to y with a single troublesome node on it. Let us for now assume that neither x nor y is troublesome. Then there will be a path $x, \ldots, a, t, b, \ldots, y$ such that t is a troublesome node. In this case, x can reach a with $O(\log \log n)$ hops, a can reach t with a single hop, t can reach b with a single hop, and b can reach y with $O(\log \log n)$ hops. The case when x or y is troublesome easily follows from the above, so the overall complexity of this case is $O(\log \log n)$ hops.

Case 3: All paths from x to y contain at least two troublesome nodes. Similar to Case 2, here we describe the case when neither x nor y is troublesome. Cases when x, y, or both are troublesome directly follow from this description. Let $x, \ldots, a, t_1, \ldots, t_2, b, \ldots, y$ be such a path where t_1 and t_2 are troublesome and no node between (x and a) and (y and b) is troublesome. Note that x can reach a in $O(\log \log n)$ hops, a can reach t_1 in a single hop, t_1 can reach t_2 in a single hop (because of the edges in E_2), t_2 can reach b in a single hop, and b can reach y in $O(\log \log n)$ hops. Thus the total path length from x to y is $O(\log \log n)$.

■

All that is left to analyze is the space complexity. Now according to our earlier discussion, $|E_1| = O(n)$ and obviously $|E_2| = O(|T|^2)$. Thus, there will be at most $O(n + |T|^2)$ new edges introduced. And if $|T|$ is relatively small, e.g., $O(\sqrt{n})$, then the space complexity is $O(n)$.

9 Conclusions

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 $O(\ell)$ operations, where ℓ 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. A formal security analysis (based on standard cryptographic assumptions) guarantees that the scheme is strongly resistant to collusions in that no subset of nodes can conspire to gain access to the key of any node to which they do not have legitimate access;
6. The private information at a node consists of a single key.

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 [16], and gave shortcut schemes that permit to significantly reduce key derivation time for trees and more general hierarchies.

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] M. Atallah, M. Blanton, and K. Frikken. Key management for non-tree access hierarchies. In *ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, 2006.

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

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO'96*, volume 1109, 1996.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] R. Cramer and V. Shoup. Design and Analysis of Practical Public-Key Encryption Scheme Secure against Adaptive Chosen Ciphertext Attack. *SIAM Journal of Computing*, 33(1):167–226, 2003.

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

[17] 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.

[18] 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.

[19] Y. Dodis, N. Fazio, A. Kiayias, and M. Yung. Scalable Public-Key Tracing and Revoking. *Journal of Distributed Computing*, 17(4):323–347, 2005.

[20] D. Dolev, C. Dwork, and M. Naor. Nonmalleable Criptography. *SIAM Journal on Discrete Mathematics*, 30(2):391–437, 2000.

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

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

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

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

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

[49] 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.

[50] 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/.

[51] 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.

[52] 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.

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

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

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

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