Dynamic and Efficient Key Management for Access Hierarchies

MIKHAIL J. ATALLAH
Purdue University
MARINA BLANTON
University of Notre Dame
NELLY FAZIO
IBM Research
and
KEITH B. FRIKKEN
Miami University

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 of assigning a key to each class in the hierarchy so that keys for descendant classes can be obtained via efficient key derivation.

We propose a solution to this problem with the following properties: (1) the space complexity of the public information is the same as that of storing the hierarchy; (2) the private information at a class consists of a single key associated with that class; (3) updates (i.e., revocations and additions) are handled *locally* in the hierarchy; (4) the scheme is provably secure against collusion; and (5) 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 pseudorandom functions, without reliance on the Random Oracle Model.

Portions of this work were supported by Grants IIS-0325345 and CNS-06274488 from the National Science Foundation and by sponsors of the Center for Education and Research in Information Assurance and Security.

A preliminary version of our scheme appeared in the *Proceedings of the ACM Conference on Computer and Communications Security (CCS'05)*. Also, the work on improving key derivation time appeared in *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT'06)*; that version contains an error, and this publication corrects it.

Author's address: M. J. Atallah, Department of Computer Science, Purdue University, West Lafayette, IN 47907.

© 2009 ACM 1094-9224/2009/01-ART18 $5.00 DOI: 10.1145/1455526.1455531.
http://doi.acm.org/10.1145/1455526.1455531.

Another substantial contribution of this work is that we are able to lower the key derivation time at the expense of modestly increasing the public storage associated with the hierarchy. Insertion of additional, so-called shortcut, edges, allows to lower the key derivation to a small constant number of steps for graphs that are total orders and trees by increasing the total number of edges by a small asymptotic factor such as $O(\log^* n)$ for an n-node hierarchy. For more general access hierarchies of dimension d, we use a technique that consists of adding dummy nodes and dimension reduction. The key derivation work for such graphs is then linear in d and the increase in the number of edges is by the factor $O(\log^{d-1} n)$ compared to the one-dimensional case.

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.

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

General Terms: Security, Design, Algorithms

Additional Key Words and Phrases: Efficient key derivation, hierarchical access control, key management

ACM Reference Format:
Atallah, M. J., Blanton, M., Fazio, N., and Frikken, K. B. 2009. Dynamic and efficient key management for access hierarchies. ACM Trans. Inf. Syst. Secur. 12, 3, Article 18 (January 2009), 43 pages. DOI = 10.1145/1455526.1455531. http://doi.acm.org/10.1145/1455526.1455531.

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 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 the work needed 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 [Ferraiolo and Kuhn 1992; Sandhu et al. 1996] 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., Fraim [1983]), databases (e.g., Denning et al. [1986]), and networking (e.g., McHugh and Moore [1986]; Lu and Sundareshan [1988]).

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 and 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 [1996; 1997]) 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 necessary 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), 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.); and
- 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 Maheshwari [2003]; Rose and Gasteiger [1994] 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 (constant for trees and linear in the dimension of the graph for more general hierarchies) 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 pseudorandom (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 rekeying 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 pseudorandom functions. Additionally, the security of the extended scheme relies on encryption with certain properties.

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 [Bell and LaPadula 1973], but

[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.

can also apply, for instance, to RBAC (e.g., reverse limited-depth inheritance permits an employee to have access to documents stored at the level of the department of that employee); 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 [Crampton 2003], 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. Insertion of additional (so called "shortcut") edges in an n-node tree or chain allows to lower efficiency of key derivation to a small constant number of operations in the worst case with an asymptotically small increase in public information (such as $O(\log^* n)$). For more general graphs, we extend the solution by employing a technique that consists of addition of dummy vertices and allows us to perform dimension reduction on the graph. The result of this technique is $O(d)$ key derivation time for access hierarchies of dimension d with an increase in public storage of $O(\log^{d-1} n)$ compared to that of one-dimensional graphs. Our solution is flexible in that it allows a trade-off between key derivation time and public storage at the server.

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 Crampton [2003]. Section 8 presents our techniques to improve efficiency of key derivation. Finally, Section 9 concludes the article.

2. RELATED WORK

The first work that addressed the problem of key management in hierarchical access control was by Akl and Taylor [1983]. Since then a large number of publications [Birget et al. 2001; Chang and Buehrer 1993; Chang et al. 2004; Chen et al. 2004; Chick and Tavares 1990; Chien and Jan 2003; Chou et al. 2004; Das et al. 2005; Ferrara and Masucci 2003; Harn and Lin 1990; He et al. 2003; Hwang 1999b,a; Hwang and Yang 2003; Liaw et al. 1993; Lin 2001; Lin et al. 2003; MacKinnon et al. 1985; Ohta et al. 1991; Ray et al. 2002; Sandhu 1987, 1988; Santis et al. 2004; Sun and Liu 2004; Tsai and Chang 1995; Zhang and Wang 2004; Zheng et al. 1992, 1993; Zhong 2002, 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 [Yeh et al. 1998; Wu and Chang 2001; Shen and Chen 2002; Tzeng 2002; Huang and Chang 2004] or incorrect [Chen and Chung 2002]. Therefore, we do not take these schemes into consideration in our further discussion.

A significant number of schemes, for example, Akl and Taylor [1983]; MacKinnon et al. [1985]; Harn and Lin [1990]; Chang and Buehrer [1993]; Hwang [1999a]; He et al. [2003]; Chick and Tavares [1990]; Ohta et al. [1991]; Hwang and Yang [2003]; Ray et al. [2002]; Lin et al. [2003]; and Santis et al. [2004], operate large numbers computed as a product of up to $O(n)$ co-prime 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 Liaw et al. [1993] and Sandhu [1987; 1988] is limited to trees and thus is of limited use. Work of Birget et al. [2001]; Sun and Liu [2004] and Zhang and Wang [2004] is concerned with a slightly different model having a hierarchy of users and a hierarchy of resources. The scheme of Birget et al. [2001], however, is not dynamic; and in Sun and Liu [2004]; and Zhang and Wang [2004] 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 Ferrara and Masucci [2003] 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 Wu and Wei [2004] uses modular exponentiation, and additions/deletions require rekeying of all descendants. A number of schemes [Das et al. 2005; Tsai and Chang 1995; Chang et al. 2004] are based on interpolating polynomials and give reasonable performance. In Tsai and Chang [1995] and Das et al. [2005], 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 re-keying on additions/deletions and store only one key per node. In Chang et al. [2004], 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) [Zheng et al. 1992, 1993] are the only efficient approaches among early schemes. In

Table I. Comparison with Previous Work

Scheme	Private storage	Public storage	Key derivation	Changes I/D/R	Proof of security
[Lin 2001]	k	$2k\|E\|$	$(3c_{\mathsf{H}} + 4c_{\mathsf{XOR}})\ell$	L/NL/L	No
[Zhong 2002]	k	$(k + k_1)\|E\|$	$(c_{\mathsf{H}} + 2c_{\mathsf{XOR}})\ell$	L/NL/L	No
[Chien and Jan 2003]	k	$k\|E\|$	$(c_{\mathsf{H}} + c_{\mathsf{XOR}})\ell$	L/NL/L	No
[Chen et al. 2004]	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

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 Zheng et al. [1992] require rekeying of all descendants and in Zheng et al. [1993] all descendants should be rekeyed when a node is deleted.

A number of recent schemes [Chen et al. 2004; Chien and Jan 2003; Chou et al. 2004; Lin 2001; Zhong 2002] use overall structure similar to ours and have performance comparable to our base scheme. Chou et al. [2004], however, does not address dynamic changes, and the scheme is less efficient than ours because of additional usage of modular multiplication. Chen et al. [2004] 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 Lin [2001] and Zhong [2002], 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. Chien and Jan [2003] 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 rekeyed, 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 I 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 of H^2; c_{XOR} corresponds to computation needed to perform bitwise XOR of two strings of size $O(k)$; and c_{D} is computation needed for symmetric key decryption. In the table, changes to the hierarchy include insertion (I), deletion (D), and rekeying (R); L stands for "local" and NL for "non-local." In all of the schemes that list "non-local" for deletions, such operations require rekeying 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 Lin [2001] and Chien and Jan [2003] 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 $\mathcal{O} : V \to 2^O$ 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, that is, 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 $\mathcal{O}^* : V \to 2^O$ 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

[2] Our solution uses pseudorandom 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.

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 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 this intuition with the following definition.

Definition 3.1. A Key Allocation (KA) scheme is a pair of polynomial-time algorithms (Set, Derive), defined as follows:

—Set(1^ρ, G) is a randomized algorithm that on input a security parameter 1^ρ and an access graph G, outputs two mappings: (1) a public mapping 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; (2) a secret mapping 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.)

—Derive(G, 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*. Informally, in defining these notions of security, we allow an adversary to corrupt keys at various nodes in the graph. The adversary then chooses a node v^* on which it would like to be challenged (subject to the constraint that the adversary does not already have access to that node's key or a key of any of its ancestors). In the case of key recovery, the adversary's goal is to compute the cryptographic key associated with node v^*. In the case of key indistinguishability, the adversary is given either v^*'s real key or a random key chosen afresh (with the two possibility being equally likely) and is asked to determine whether such key corresponds to the v^*'s real key or not.

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 as:

$$Adv_{\mathcal{A}}^{KR} \doteq \Pr[k'_{v^*} = k_{v^*}].$$

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 as:

$$Adv_{\mathcal{A}}^{KI} \doteq \left| \Pr[b = b^*] - \frac{1}{2} \right|.$$

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 pseudorandom function, which can be implemented as, e.g., HMAC [Bellare et al. 1996] 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 pseudorandom functions:

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

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

Experiment $\mathbf{Exp}^{\mathsf{PRF}-1}_{F,A}(\rho)$	Experiment $\mathbf{Exp}^{\mathsf{PRF}-0}_{F,A}(\rho)$
$k \overset{r}{\leftarrow} K^\rho$	$g \overset{r}{\leftarrow} Rand^\rho$
$d \leftarrow A^{F^\rho_k}(1^\rho)$	$d \leftarrow A^{g}(1^\rho)$
Return d	Return d

The PRF-advantage of A is then defined as:

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

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

Assume that we are given a PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$ where $F^\rho : \{0, 1\}^\rho \times \{0, 1\}^\rho \rightarrow \{0, 1\}^\rho$.[3] Given an access graph $G = (V, E)$ and a security parameter ρ, the 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 $y_{ij} \doteq k_j \oplus F(k_i, \ell_j)$.

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

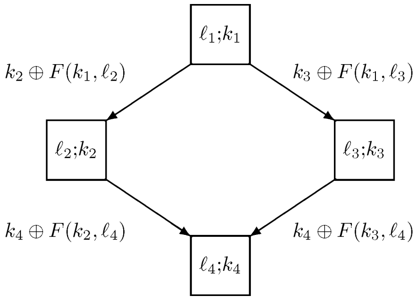
$\ell_1;k_1$
$k_2 \oplus F(k_1, \ell_2)$
$k_3 \oplus F(k_1, \ell_3)$
$\ell_2;k_2$
$\ell_3;k_3$
$k_4 \oplus F(k_2, \ell_4)$
$k_4 \oplus F(k_3, \ell_4)$
$\ell_4;k_4$

Fig. 1. Key allocation for example access graph.

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

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

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 $(v_{\bar{i}}, v_{\bar{j}})$ on the path between v_i and v_j. Given an edge $(v_{\bar{i}}, v_{\bar{j}})$ for which both $v_{\bar{i}}$'s private key $k_{\bar{i}}$ and the stored public information $\ell_{\bar{j}}$ and $y_{\bar{i}\bar{j}}$ are known, $v_{\bar{i}}$ can generate $v_{\bar{j}}$'s private information $k_{\bar{j}}$ thanks to the fact that $y_{\bar{i}\bar{j},}$ is defined as $y_{\bar{i}\bar{j}} \doteq k_{\bar{j}} \oplus F(k_{\bar{i}}, \ell_{\bar{j}})$.

Due to the sequential nature of key generation on the path between v_i and v_j, v_i will be able to derive keys of all necessary nodes and produce key k_j.

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.

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 pseudorandom 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 Dodis et al. [2005], first advocated in Cramer and Shoup [2003]. 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 pseudorandom function F. To this aim, we need to show how to turn an adversary $\mathcal{A}$ attacking the scheme into an adversary $\mathcal{A}_F$ 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 $\mathcal{A}_F$ 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 $\mathcal{A}_F$ 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.*, $y_{1j} \overset{r}{\leftarrow} \{0,1\}^\rho$.

Note that such modification essentially amounts to substituting any occurrences of the pseudorandom 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 $\{F^\rho\}_{\rho\in\mathbb{N}}$. Formally, Lemma 4.3 below shows that any non-negligible difference in $\mathcal{A}$'s behavior between game $\mathbf{G}_0$ and $\mathbf{G}_1$ can be used to construct a probabilistic polynomial-time algorithm $\mathcal{A}_F$ that is able to break the pseudorandom function F with non-negligible advantage. Hence,

$$\left|\Pr[T_1] - \Pr[T_0]\right| \le \epsilon_{\mathsf{PRF}} \qquad (1)$$

where ϵ_{PRF} is an upper bound on the advantage $\mathsf{Adv}^{\mathsf{PRF}}_{F,\mathcal{A}_F}(\rho)$ of any probabilistic polynomial-time adversary $\mathcal{A}_F$ against the security of the pseudorandom function F. Notice that ϵ_{PRF} is negligible by our security assumption on the PRF family.

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

Game $\mathbf{G}_i$ ($1 \le i \le h$). 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 pseudorandom 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 $\mathbf{G}_1, \ldots, \mathbf{G}_{i-1}$, 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 $\mathbf{G}_{i-1}$, using a reduction argument along the same line as that in Lemma 4.3, we can conclude that such modification is warranted by the security of the PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$, that is:

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

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 $1/2^\rho$, that is:

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

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

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

□

LEMMA 4.3. $\left|\Pr[T_1] - \Pr[T_0]\right| \leq \epsilon_{\mathsf{PRF}}$

PROOF. Let's assume we have an adversary $\mathcal{A}$ that is able to distinguish between game $\mathbf{G}_0$ and game $\mathbf{G}_1$. We describe below how to construct an algorithm $\mathcal{A}_F$ that, using $\mathcal{A}$ as a black box, is able to distinguish between pseudorandom and truly random functions.

Algorithm $\mathcal{A}_F$ plays the PRF game described in Definition 4.1, and is thus given oracle access to a function $g(\cdot)$ that is either a pseudorandom function, keyed with a secret value k, or a truly random function.

In order to use algorithm $\mathcal{A}$, $\mathcal{A}_F$ simulates the environment of $\mathcal{A}$ in a way that interpolates between game $\mathbf{G}_0$ and game $\mathbf{G}_1$. In other words, if $\mathcal{A}_F$ is interacting with a pseudorandom function, then the simulation seen by $\mathcal{A}$ proceeds exactly as in game $\mathbf{G}_0$; otherwise, the simulation proceeds exactly as in game $\mathbf{G}_1$.

The first step in $\mathcal{A}_F$'s simulation consists of setting up the access hierarchy for graph G by running the $\mathsf{Set}(1^\rho, G)$ algorithm with the following modification: 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 computed via oracle g, *i.e.*, $y_{1j} \overset{r}{\leftarrow} k_j \oplus g(\ell_j)$. This is equivalent to game $\mathbf{G}_0$ when $\mathcal{A}_F$'s oracle computes g

according to a pseudorandom function, and equivalent to game $\mathbf{G}_1$ when $\mathcal{A}_F$'s oracle computes g completely at random.

After feeding $\mathcal{A}$ with the resulting public information, $\mathcal{A}_F$ can readily reply to any Corrupt query that $\mathcal{A}$ may issue, since $\mathcal{A}_F$ knows all the secret keys except for the one associated to node v_1. Note that according to the attack game in Definition 3.3, $\mathcal{A}$ cannot ask for such key, since node v_1 is among the ancestors of the challenge node v^* that adversary $\mathcal{A}$ will attack in the **Break** stage of the attack game.

Eventually, $\mathcal{A}$ outputs her best guess k'_{v^*} at the secret key corresponding to node v^*: if $k'_{v^*} = k_{v^*}$, then $\mathcal{A}_F$ outputs 1, guessing for a pseudorandom function; otherwise, $\mathcal{A}_F$ outputs 0, guessing for a truly random function.

Now we have:

$$\begin{aligned}\epsilon_{\mathsf{PRF}} &\geq Adv^{\mathsf{PRF}}_{F,\mathcal{A}_F}(\rho)\\ &\doteq \left| \Pr[\mathbf{Exp}^{\mathsf{PRF}-1}_{F,A}(\rho) = 1] - \Pr[\mathbf{Exp}^{\mathsf{PRF}-0}_{F,A}(\rho) = 1] \right|\\ &= \left| \Pr[\mathcal{A}_F \text{ outputs } 1 \mid g \text{ is a PRF}] - \Pr[\mathcal{A}_F \text{ outputs } 1 \mid g \text{ is random}] \right|\\ &= \left| \Pr[\mathcal{A} \text{ guesses } k_{v^*} \text{ correctly} \mid g \text{ is a PRF}] \right.\\ &\qquad \left. - \Pr[\mathcal{A} \text{ guessed } k_{v^*} \text{ correctly} \mid g \text{ is random}] \right|\\ &= \left| \Pr[T_0] - \Pr[T_1] \right| .\end{aligned}$$

□

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. Note that our base scheme fails to achieve this stronger notion of security because the adversary will be able to easily test whether the challenge key it is given corresponds to the actual key of the challenge node v^* or not. More precisely, suppose the adversary requests the secret information $S_j = k_j$ corresponding to a child node v_j of v^* (in general, it does not have to be a child node, any descendant node will permit this attack) and tests it using public information. Then, the adversary checks $F(c_{v^*}, \ell_j) \oplus y_{v^*,j} \overset{?}{=} k_j$, where c_{v^*} is the challenge key. If the values match, the adversary can be sure that it was given the actual key of v^*. Note that the problem is that v_i's key k_i is used for two different purposes: to grant access to objects at v_i and to aid derivation of child keys. To address this issue, we ensure that two different keys are used for these purposes, both of which can be derived from the node's secret information S_i: the key k_i is used to obtain access to objects, while the key t_i is used for key derivation only. Then, when the adversary is presented with a challenge key c_{v^*}, it is no longer possible to use the public information to test whether c_{v^*} corresponds to k_{v^*} or a randomly chosen key.

Our construction is based on the use of a semantically secure symmetric-key encryption scheme $\mathcal{E}$. Several equivalent formalizations of this security notion have been proposed in the literature. We report one such definition below; we refer the reader to Section 5.2 of Goldreich [2004] for a thorough treatment.

Definition 5.1. A symmetric-key encryption scheme $\mathcal{E}$ is a triple of polynomial-time algorithms (Gen, Enc, Dec) where:

—$\mathsf{Gen}(1^\lambda)$ is a randomized algorithm that on input a security parameter 1^λ, outputs a secret key SK and a message space $\mathcal{M}$;
—$\mathsf{Enc}_{SK}(m)$ is a randomized algorithm that on input a secret key SK and a message $m \in \mathcal{M}$, outputs a ciphertext $c \in \{0,1\}^*$; and
—$\mathsf{Dec}_{SK}(c)$ is a deterministic algorithm that on input a secret key SK and a ciphertext c, outputs a message $\bar{m} \in \mathcal{M}$ or a special reject symbol $\perp$.

For correctness, Gen, Enc, and Dec ought to satisfy the following constraint:

$$\Pr[\bar{m} = m \mid (SK, M) \overset{r}{\leftarrow} \mathsf{Gen}(1^\lambda); \forall m \in M; \bar{m} \leftarrow \mathsf{Dec}_{SK}(\mathsf{Enc}_{SK}(m))] = 1.$$

Definition 5.2. An encryption scheme $\mathcal{E}$ is semantically secure if no polynomial-time adversary $\mathcal{A}$ has a non-negligible advantage (in the security parameter λ) against a challenger in the following game:

—**Setup**: The challenger runs $\mathsf{Set}(1^\lambda)$: it keeps secret the resulting secret key SK and gives the message space $\mathcal{M}$ to the adversary $\mathcal{A}$.
—**Challenge**: The adversary specifies a message $m_0 \in \mathcal{M}$. The challenger picks a random bit $b^* \in \{0,1\}$: if $b^* = 0$, then it computes $c^* = \mathsf{Enc}_{SK}(m_0)$; otherwise it sets $c^* = \mathsf{Enc}_{SK}(\$)$, where \$ is a random string of the same length as m_0. The challenger returns c^* to $\mathcal{A}$.
—**Guess**: The adversary outputs a bit $b \in \{0,1\}$ as her best guess to whether she was given the encryption of m_0 or of a random string of the same length as m_0. $\mathcal{A}$ wins the game if $b = b^*$.

We define the adversary's advantage in attacking the scheme to be

$$Adv_{\mathcal{A}}^{\mathsf{SEM}} \doteq \left|\Pr[\mathcal{A} \text{ outputs } 1 \mid c^* = \mathsf{Enc}_{SK}(m_0)] - \Pr[\mathcal{A} \text{ outputs } 1 \mid c^* = \mathsf{Enc}_{SK}(\$)]\right|.$$

The extended 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 operations. Additionally, the extended scheme makes use of a semantically secure symmetric-key encryption scheme $\mathcal{E}$ (*c.f.* Definition 5.1 and Definition 5.2), and key derivation involves one evaluation of a pseudorandom function $F : \{0,1\}^\rho \times \{0,1\}^* \to \{0,1\}^\rho$ (*c.f.* Definition 4.1) and one decryption.

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:

$$\begin{array}{ll} \mathsf{Pub} : v_i \mapsto \ell_i & \quad \mathsf{Pub} : (v_i, v_j) \mapsto y_{ij} \\ \mathsf{Sec} : v_i \mapsto (S_i, k_i) & \end{array}$$

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[4] 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;
    $r_{\bar\imath\bar\jmath} \leftarrow F_{t_{\bar\imath}}(\ell_{\bar\jmath})$;
    $t_{\bar\jmath}||k_{\bar\jmath} \leftarrow \mathsf{Dec}_{r_{\bar\imath\bar\jmath}}(y_{\bar\imath\bar\jmath})$;
    $\bar\imath \leftarrow \bar\jmath$;    $t_{\bar\imath} = t_{\bar\jmath}$;
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.

Next, we prove that the extended scheme described in this section is key indistinguishable (*c.f.* Definition 3.3), following the same approach as in the proof of Theorem 4.2.

THEOREM 5.3. *The above extended scheme is key indistinguishable for any directed acyclic graph G, assuming the security of the pseudorandom 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 pseudorandom 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 $\mathcal{A}_F$ attacking F or an adversary $\mathcal{A}_\mathcal{E}$ 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 $\mathcal{A}_F$ and $\mathcal{A}_\mathcal{E}$ 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 $\mathcal{A}$

[4] If there is more than one path, pick one arbitrarily, for example, the shortest path from v_i to v_j.

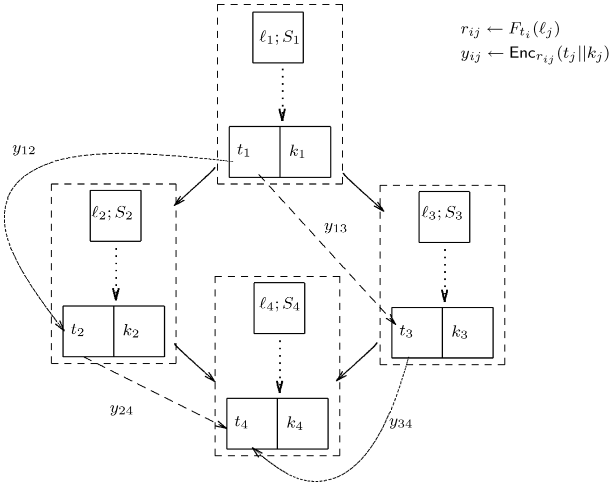

Fig. 2. Key allocation for extended example access graph.

will ask the challenge query and construct adversaries $\mathcal{A}_F$ (or $\mathcal{A}_\mathcal{E}$) 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[5] 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.

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

Note that such modification essentially amounts to substituting any occurrences of the pseudorandom 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 pseudorandom function. Formally, Lemma 5.4 below shows that any non-negligible difference in behavior between game $\mathbf{G}_0$ and $\mathbf{G}_1$ can be used to construct a probabilistic polynomial-time algorithm $\mathcal{A}_F$ that is able to break the pseudorandom function F with non-negligible advantage. Hence,

$$\left|\Pr[T_1] - \Pr[T_0]\right| \leq \epsilon_{\mathsf{PRF}}, \tag{4}$$

where ϵ_{PRF} is an upper bound on the advantage $\mathsf{Adv}^{\mathsf{PRF}}_{F,\mathcal{A}_F}$ of any probabilistic polynomial-time adversary $\mathcal{A}_F$ against the security of the pseudorandom function F. Notice that ϵ_{PRF} is negligible by our security assumption on the PRF family.

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. □

LEMMA 5.4. $\left|\Pr[T_1] - \Pr[T_0]\right| \leq \epsilon_{\mathsf{PRF}}$

PROOF. Let's assume we have an adversary $\mathcal{A}$ that is able to distinguish between game $\mathbf{G}_0$ and game $\mathbf{G}_1$. We describe below how to construct an algorithm $\mathcal{A}_F$ that, using $\mathcal{A}$ as a black-box, is able to distinguish between pseudorandom and truly random functions.

Algorithm $\mathcal{A}_F$ plays the PRF game described in Definition 4.1, and is thus given oracle access to a function $g(\cdot)$ that is either a pseudorandom function, keyed with a secret value k, or a truly random function.

In order to use algorithm $\mathcal{A}$, $\mathcal{A}_F$ simulates the environment of $\mathcal{A}$ in a way that interpolates between game $\mathbf{G}_0$ and game $\mathbf{G}_1$. In other words, if $\mathcal{A}_F$ is interacting with a pseudorandom function, then the simulation seen by $\mathcal{A}$ proceeds exactly as in game $\mathbf{G}_0$; otherwise, the simulation proceeds exactly as in game $\mathbf{G}_1$.

The first step in $\mathcal{A}_F$'s simulation consists of setting up the access hierarchy for graph G by running the $\mathsf{Set}(1^\rho, G)$ algorithm with the following modification: the cryptographic keys t_{v^*} and k_{v^*} are computed via oracle g as follows:

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

This is equivalent to game $\mathbf{G}_0$ when $\mathcal{A}_F$'s oracle computes g according to a pseudorandom function, and equivalent to game $\mathbf{G}_1$ when $\mathcal{A}_F$'s oracle computes g completely at random.

After feeding $\mathcal{A}$ with the resulting public information, $\mathcal{A}_F$ can readily reply to any Corrupt query that $\mathcal{A}$ may issue, since $\mathcal{A}_F$ knows all the secret keys except for the one associated to nodes v^*. Note that according to the attack game in Definition 3.3, $\mathcal{A}$ cannot ask for S_{v^*}, since adversary $\mathcal{A}$ will attack v^* in the **Break** stage of the attack game.

Upon receiving the challenge query from $\mathcal{A}$, $\mathcal{A}_F$ picks a random bit $b^* \in \{0, 1\}$: if $b^* = 0$, then $\mathcal{A}_F$ returns to $\mathcal{A}$ the cryptographic key k_{v^*} associated to node v^*; otherwise, $\mathcal{A}_F$ returns to $\mathcal{A}$ a random key $\bar{k}_{v^*}$ of the same length of k_{v^*}.

Eventually, $\mathcal{A}$ a outputs bit b as her best guess at whether she was given the actual key k_{v^*} or a random key: if $b = b^*$, then $\mathcal{A}_F$ outputs 1, guessing for a pseudorandom function; otherwise, $\mathcal{A}_F$ outputs 0, guessing for a truly random function.

Now we have

$$\begin{aligned}\epsilon_{\mathsf{PRF}} \geq\ & Adv^{\mathsf{PRF}}_{F,\mathcal{A}_F}(\rho) \\ \doteq & \mid \Pr[\mathbf{Exp}^{\mathsf{PRF}-1}_{F,A}(\rho) = 1] - \Pr[\mathbf{Exp}^{\mathsf{PRF}-0}_{F,A}(\rho) = 1] \mid \\ = & \mid \Pr[\mathcal{A}_F \text{ outputs } 1 \mid g \text{ is a PRF}] - \Pr[\mathcal{A}_F \text{ outputs } 1 \mid g \text{ is random}] \mid \\ = & \mid \Pr[\mathcal{A} \text{ guesses } b^* \text{ correctly} \mid g \text{ is a PRF}] \\ & \quad - \Pr[\mathcal{A} \text{ guessed } b^* \text{ correctly} \mid g \text{ is random}] \mid \\ = & \mid \Pr[T_0] - \Pr[T_1] \mid .\end{aligned}$$

□

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 pseudorandom function $F_{S_p}(\cdot)$ with a truly random function $R_1(\cdot)$. Since S_p does not occur anywhere else in the attack game, using a reduction argument along the same line as that in Lemma 5.4, we can conclude that such modification is warranted by the security of the PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$, that is:

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

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 pseudorandom 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 reduction argument along the same line as that in Lemma 5.4, we can conclude that such modification is warranted by the security of the PRF family $\{F^\rho\}_{\rho\in\mathbb{N}}$, that is:

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

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 $\mathbf{G}_2^{(b)}$). Formally, Lemma 5.5 below shows that any non-negligible difference in behavior between games $\mathbf{G}_2^{(b)}$ and $\mathbf{G}_2^{(c)}$ can be used to construct a probabilistic polynomial-time algorithm $\mathcal{A}_\mathcal{E}$ that is able to break the security of the encryption scheme $\mathcal{E}$ with non-negligible advantage. Hence,

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

where ϵ_{SEM} is an upper bound on the advantage $Adv_{\mathcal{A}_\mathcal{E}}^{\mathsf{SEM}}$ of any probabilistic polynomial-time adversary $\mathcal{A}_\mathcal{E}$ against the security of the encryption scheme $\mathcal{E}$. Notice that ϵ_{SEM} is negligible by our security assumption on the encryption scheme.

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.

LEMMA 5.5. $\left|\Pr[T_2^{(c)}] - \Pr[T_2^{(b)}]\right| \leq \epsilon_{\mathsf{SEM}}$

PROOF. Let's assume we have an adversary $\mathcal{A}$ that is able to distinguish between game $\mathbf{G}_2^{(b)}$ and game $\mathbf{G}_2^{(c)}$. We describe below how to construct an algorithm $\mathcal{A}_\mathcal{E}$ that, using $\mathcal{A}$ as a black-box, is able to break the semantic security of the encryption scheme $\mathcal{E}$ with non-negligible advantage.

Algorithm $\mathcal{A}_\mathcal{E}$ plays the SEM game described in Definition 5.2. In order to use algorithm $\mathcal{A}$, $\mathcal{A}_\mathcal{E}$ simulates the environment of $\mathcal{A}$ in a way that interpolates between game $\mathbf{G}_2^{(b)}$ and game $\mathbf{G}_2^{(c)}$. In other words, if $\mathcal{A}_\mathcal{E}$'s challenge is answered with the encryption of the message $\mathcal{A}_\mathcal{E}$ asked for, then the simulation seen by $\mathcal{A}$ proceeds exactly as in game $\mathbf{G}_2^{(b)}$; otherwise (i.e., $\mathcal{A}_\mathcal{E}$ gets back a random encryption as challenge), the simulation proceeds exactly as in game $\mathbf{G}_2^{(c)}$.

The first step in $\mathcal{A}_{\mathcal{E}}$'s simulation consists of setting up the access hierarchy for graph G by running the $\mathsf{Set}(1^{\rho}, G)$ algorithm with the following modification: first, $\mathcal{A}_{\mathcal{E}}$ asks to be challenged on the message $m_0 \doteq t_{v^*}||k_{v^*}$ and gets back a ciphertext c^* which is the encryption of either m_0 or a random string of the same length. Then, $\mathcal{A}_{\mathcal{E}}$ computes the label y_{pv^*} associated with edge $(p, v^*) \in E$ as follows:

$$y_{pv^*} \leftarrow c^*.$$

This is equivalent to game $\mathbf{G}_2^{(b)}$ when $\mathcal{A}_{\mathcal{E}}$'s challenger computes c^* as the encryption of m_0, and equivalent to game $\mathbf{G}_2^{(c)}$ otherwise.

After feeding $\mathcal{A}$ with the resulting public information, $\mathcal{A}_{\mathcal{E}}$ can readily reply to any $\mathsf{Corrupt}$ query that $\mathcal{A}$ may issue, since $\mathcal{A}_{\mathcal{E}}$ knows all the secret keys except for the one associated to node p. Note that according to the attack game in Definition 3.3, $\mathcal{A}$ cannot ask for such key, since node p is among the ancestors of the challenge node v^* that adversary $\mathcal{A}$ will attack in the **Break** stage of the attack game.

Upon receiving the challenge query from $\mathcal{A}$, $\mathcal{A}_{\mathcal{E}}$ picks a random bit $b^* \in \{0, 1\}$: if $b^* = 0$, then $\mathcal{A}_{\mathcal{E}}$ returns to $\mathcal{A}$ the cryptographic key k_{v^*} associated to node v^*; otherwise, $\mathcal{A}_{\mathcal{E}}$ returns to $\mathcal{A}$ a random key $\bar{k}_{v^*}$ of the same length of k_{v^*}.

Eventually, $\mathcal{A}$ outputs a bit b as her best guess to whether she was given the actual key k_{v^*} or a random key: if $b = b^*$, then $\mathcal{A}_{\mathcal{E}}$ outputs 1, guessing for encryption of m_0; otherwise, $\mathcal{A}_{\mathcal{E}}$ outputs 0, guessing for encryption of a random string.

Now we have

$$\begin{aligned}
\epsilon_{\mathsf{SEM}} \geq\ & Adv^{\mathsf{SEM}}_{\mathcal{A}_{\mathcal{E}}} \\
=& |\ \Pr[\mathcal{A}_{\mathcal{E}} \text{ outputs } 1 \mid c^* \text{ is the encryption of } m_0] \\
& - \Pr[\mathcal{A}_{\mathcal{E}} \text{ outputs } 1 \mid c^* \text{ is the encryption of a random string}]\ | \\
=& |\ \Pr[\mathcal{A} \text{ guesses } b^* \text{ correctly} \mid c^* \text{ is the encryption of } m_0] \\
& - \Pr[\mathcal{A} \text{ guessed } b^* \text{ correctly} \mid c^* \text{ is the encryption of a random string}]\ | \\
=& |\ \Pr[T_2^{(b)} - \Pr[T_2^{(c)}\ |\ .
\end{aligned}$$

□

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 $R_u^{(a)} : \{0, 1\}^* \to \{0, 1\}^*$ 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 $R_u^{(b)} : \{0, 1\}^* \to \{0, 1\}^*$ is a truly random function.
Third, for each successor s^* of u in the (possibly multiple) path(s) from u to v^*, we define a game $\mathbf{G}_u^{(c)}$ in which we set

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

where \$ 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 $2\epsilon_{\mathsf{PRF}} + \epsilon_{Enc}$). 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

$$\Pr[T_0] \leq 1/2 + \epsilon_{\mathsf{PRF}} + 2n_{v^*}\epsilon_{\mathsf{PRF}} + e_{v^*}\epsilon_{\mathsf{SEM}}),$$

where n_{v^*} and e_{v^*} are respectively the number of nodes and the number of edges in the subgroup G' induced by restricting G to the 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. Compared to the base scheme, the extended scheme of Section 5 has more appealing properties in supporting dynamic changes to an existing hierarchy, and, in particular, at handling the so-called ex-member problem after deleting an edge. The ex-member problem can be described as follows: Suppose the edge (v_i, v_j) is to be deleted from G.

Then we will need to change the access keys for the node v_j and all of it descendants in the hierarchy to ensure that v_i and its ancestors can no longer have access to these nodes. In the base scheme, this will amount to changing the secret information $S_h = k_h$ for each $v_h \in Desc(v_j)$ and redistributing the new keys to the users at these nodes. Our extended scheme, however, allows for such updates without having to change the secret information users possess. Notice that in the extended scheme each access key k_i is computed as a function of the secret information S_i and the public label ℓ_i. Therefore, by changing the label ℓ_i, we effectively modify the key k_i. This approach will work with other schemes where the access key k_i is computed a function of S_i and ℓ_i, but not with our base scheme.

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 take place: 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 $S_i^{'} \overset{r}{\leftarrow} \{0, 1\}^\rho$.

(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 $v_j \in \mathit{Pred}(v_i)$), 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 $v_l \in \mathit{Succ}(v_i)$), 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, Zhong [2002] considers only a hierarchy of security classes without mentioning individual users, and Lin [2001] 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 re-keying.

7. OTHER ACCESS MODELS

Traditionally, the standard notion of permission inheritance in access control is that permissions are transferred up the access graph G. In other words, any vertex in $\mathit{Anc}(v_i, G)$ has a superset of the permissions held by v_i. Crampton [2003] suggested other access models, including:

(1) Permissions that are transferred down the access graph. For these permissions, any node in $\mathit{Desc}(v_i, G)$ has a superset of the permissions held by v_i.

(2) Permissions that are transferred 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 layer in both graphs. 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 extra, so-called *shortcut*, edges to access graphs so that the distance between any two nodes is small. This is essential for deep hierarchies since the key derivation time in our scheme is the depth of the access graph in the worst case. We start with techniques for one-dimensional graphs (i.e., total orders) and trees and then describe an additional technique to extend the technique to graphs of higher dimensions. Before proceeding with the description of adding shortcut edges, we define the notion of dimension of an access hierarchy.

8.1 Dimension of an Access Hierarchy

An n-vertex access hierarchy G is a partial order, and it is well known that any partial order can be represented as the intersection of t total orders, with the smallest t for which this is possible being the *dimension* of the partial order (see, for example, Dushnik and Miller [1941] and Trotter [1992]). That is, it is possible to associate with every vertex v of G a t-tuple $(x_{v,1}, \ldots, x_{v,t})$ such that:

(1) Every $x_{v,j}$ is an integer between 1 and n.
(2) If $v \neq w$, then $x_{v,j} \neq x_{w,j}$, for every $1 \leq j \leq t$.
(3) Node v is ancestor of node w in G if and only if $x_{v,j} > x_{w,j}$ for every $1 \leq j \leq t$.

We denote the dimension of G by $d(G)$, or by d when G is understood. While computing the dimension of an arbitrary partial order is NP-complete [Yannakakis 1982], and even approximating it to within a constant factor is not known to be in P, the dimension of many access hierarchies is small. For instance, the dimension of a tree is 2. Also, it was shown in Schnyder [1989] that a G whose transitive reduction is planar has dimension at most three (and the three-tuples representing it are computable in linear time). If the transitive reduction of G is 4-colorable, then its dimension is at most four [Schnyder 1989]. Many access hierarchies are four-colorable, especially those for organizational hierarchies.

There are, however, some hierarchies with higher dimension. For example, in the Bell-LaPadula model with k categories (denoted by $s_1, \ldots, s_k$) and ℓ classifications (denoted by $c_1, \ldots, c_\ell$), the dimension of the lattice is $k + 1$. Fortunately, computing the tuple representation for this model is straightforward: The access level c_i with categories in the set S is converted into a tuple $(i, x_1, \ldots, x_k)$ where $x_i = 1$ if and only if $s_i \in S$, and is 0 otherwise. It is not difficult to verify that this conversion correctly implements the access control policy.

We may actually not need to compute the dimension, but rather *any* d'-tuple representation of the graph with a small enough d'. Moreover, some access graphs can naturally be specified in such a tuple representation, when, for instance, the ancestor relationship is the conjunction of a number of total-order conditions such as "v has higher security clearance than w," "v is a higher-priority asset than w," "v is more vulnerable than w," "v is a higher-paying class of subscribers than w," etc. In summary, the techniques of this article generalize the shortcut technique to any access hierarchy where a tuple-based representation (of reasonable dimension) can be found. This significantly extends the results of the previous work that supported only trees.

8.2 The One-Dimensional Case

Shortcut schemes have been considered in prior literature [Yao 1982; Chazelle 1987; Alon and Schieber 1987; Thorup 1992; Bodlaender et al. 1994; Thorup 1995, 1997], and the known bounds (explored in a different domain) for an n-node graph are presented in Table II. These bounds hold for both trees and one-dimensional graphs (chains).

In the remainder of this subsection we present techniques for the one-dimensional case as they apply to the key management domain (in case of trees, a similar approach based on the the notion of a *centroid* (a node removal of which leaves no connected components of size greater than $n/2$ for an n-node tree) and *centroid decomposition* can be used). In what follows, let the nodes

Table II. The Minimum Number of Shortcut Edges Necessary for an n-Node Chain or Tree to Achieve Diameter h

Diameter h	Minimum number of shortcut edges
1	$\Theta(n^2)$
2	$\Theta(n \log n)$
3	$\Theta(n \log \log n)$
4	$\Theta(n \log^* n)$
5	$\Theta(n \log^* n)$
6	$\Theta(n \log^{**} n)$
7	$\Theta(n \log^{**} n)$
...	...
$\log^* n$	$\Theta(n)$

(i.e., access classes) form a one-dimensional graph (i.e., a total order) and be numbered v_1 through v_n. Furthermore, the access rights of node v_i are a superset of the access rights of node v_j if and only if $i \leq j$. We sometimes refer to nodes with lower indices as nodes "on the left" and to nodes with higher indices as nodes "on the right."

Our goal is to compute a small set of shortcut edges such that the distance between any two nodes is minimized preserving the original relationship between the nodes. Given a set of edges E in a graph, we denote the minimum path length between two nodes v_i and v_j by $dist(v_i, v_j)$; this distance is infinity if $i > j$. Then for our graph, the distance between any pair of nodes is bounded by $\max_{v_i, v_j \in V, i<j} dist(v_i, v_j)$. We say that a shortcut scheme is an h-hop solution if no two nodes' distance is more than h, that is, $\max_{v_i, v_j \in V, i<j} dist(v_i, v_j) \leq h$. Our goal is to determine a small set of edges that results in an h-hop solution.

The transitive closure of the directed acyclic graph results in a one-hop solution with $O(n^2)$ edges, and it can easily be shown that this solution is an optimal (in terms of number of edges) one-hop solution. Thus in the remainder of this section we concentrate on solutions with more than a single hop which use much less space.

8.2.1 *Two-hop solutions.* Here we present a shortcut scheme where the distance between any two nodes is at most two edges. This solution requires addition of $O(n \log n)$ edges to the original graph. This bound can be proven to be optimal for any two-hop solution.

AddShortcuts$_2(G)$:

(1) Let n denote the number of nodes in G. If $n \leq 3$, then add edges between consecutive nodes and quit. Otherwise, proceed with the next step.
(2) Find the median node, that is, the node that is dominated by about half of the nodes and that dominates the other half; we denote this median by m. Place the nodes that dominate m in a set L; and place the nodes dominated by m in a set R.
(3) For each node $v_i \in L$, create a shortcut edge (v_i, m).
(4) For each node $v_i \in R$, create a shortcut edge (m, v_i).

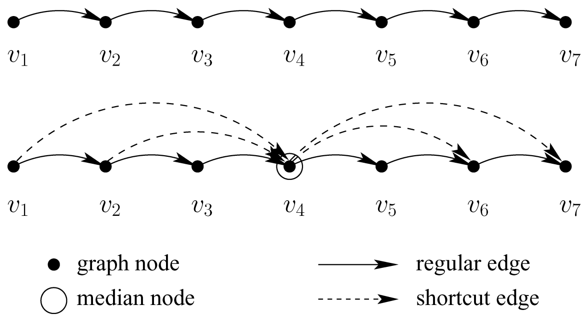

Fig. 3. Addition of shortcut edges for the two-hop one-dimensional solution.

(5) Create a graph G_L from the nodes in L and execute AddShortcuts$_2(G_L)$.

(6) Similarly for R, create a graph G_R and execute AddShortcuts$_2(G_R)$.

Figure 3 depicts the first level of recursion for the above procedure.

It can be easily shown that, in the above-defined structure, the nodes in the graph are at most two hops from each other. That is, suppose nodes x and y are separated by some median m during the above protocol. Then clearly there is a path of length two from x to y (specifically, x to m to y). On the other hand, suppose that x and y are never separated by a median, then from the base case (Step 1) the nodes will have a path of length at most two.

The space required by the solution follows the recurrence $f(n) = O(1)$ for $n \leq 3$, and $f(n) = O(n) + 2f(n/2)$ otherwise. It is straightforward to show that $f(n) = O(n \log n)$.

Creation of a data structure with two-hop paths implies that we also need a constant-time algorithm for finding it. That is, we need a FindPath$_2(x, y, G)$ procedure that, given two nodes x and y, finds a path consisting of two edges from point x to point y. To achieve this, we store the recursion tree (call it RT) for the above AddShortcuts$_2$ algorithm, which takes no more space than storing the shortcut edges. The two-hop path we seek would be easy to find if we could, in constant time, compute the lowest node (call it u) of RT for which x and y are a part of that node's sub-problem: the shortcut edges (x, m) and (m, y) are available at the node u in RT, where m is the median of node u's subproblem. Fortunately, computing u is easy to do in constant time, by making use of Harel and Tarjan [1984] that showed that in any tree it is possible to answer *nearest common ancestor* (NCA) queries in constant time. In more detail, given any two nodes of RT, their common ancestor in RT that is nearest to them can be computed in constant time (in fact, doing so is rather straightforward in our case where RT is a complete binary tree). In our case, the two nodes whose NCA we seek are the leaves of RT that contain x and y, and their NCA is the node u that contains the two shortcut edges that we want.

8.2.2 *Three-hop solutions.* In this section, we describe a shortcut scheme where nodes are separated by at most three hops. Atallah et al. [2005] gave a scheme for trees that introduces $O(n \log \log n)$ edges. While trees have dimension $d = 2$, we cannot use these techniques for the case of $d = 2$, because not all graphs of dimension two are trees. Thus, we adopt that solution to the one-dimensional case and, for completeness, briefly describe the scheme next. We would like to note that this bound is asymptotically optimal for any three-hop solution.

For ease of presentation, the procedure below is given for the case $n = 2^{2^q}$. This allows us to avoid using floor/ceiling functions, but does not narrow the applicability of the solution.

AddShortcuts$_3(G)$:

(1) Let n denote the number of nodes in G. If $n \leq 4$, then add edges between the consecutive nodes. Otherwise, proceed with the next step.

(2) Create a set of special nodes S that consists of every $\sqrt{n}$th node in the graph. That is, initialize S with $\{v_n\}$ and then add nodes $v_{n-j\sqrt{n}}$ for all j such that $j\sqrt{n} < n$ (note that $j < \sqrt{n}$). Let us refer to this set as $S = \{v_{i_1}, \ldots, v_{i_m}\}$, where $i_1 < i_2 < \cdots < i_m$.

(3) Insert new edges between the nodes in S to form the transitive closure of the set (i.e., now the nodes in S are one hop away from each other).

(4) For each node $v_i \notin S$, if a node $v_j \in S$ exists such that $j < i$ and $i < j + \sqrt{n}$, insert an edge (v_j, v_i) if it is not already present.

(5) For each node $v_i \notin S$, find $v_j \in S$ such that $i < j$ and $j < i + \sqrt{n}$, insert an edge (v_i, v_j) if it is not already present.

(6) Form a subgraph G_j from the nodes between v_{i_j} and $v_{i_{j+1}}$ and the edges that preserve their ordering. Also, construct a subgraph G_0 from the nodes before v_{i_1} and the edges connecting them. Execute AddShortcuts$_3$ on graphs $G_0, \ldots, G_{m-1}$ to recursively add shortcut edges to them.

Figure 4 depicts different stages of the above algorithm for the first level of recursion. The top figure gives the original hierarchy, the middle figure shows the hierarchy after selection of special nodes and constructing their transitive closure, and the bottom figures shows the hierarchy after adding shortcut edges to and from the special nodes.

To demonstrate that in the above data structure the nodes are at most three hops from each other, we consider all cases. Clearly, in the base case (Step 1), nodes are at most three hops from each other. Also, if nodes x and y (where x is left of y) are separated by a special node, then x can reach its nearest special node, x', in at most one hop (from Step 5), x' can reach the special node, y', that is rightmost special node before y in at most on hop (from Step 3), and y' can reach y in at most one hop (from Step 4). Finally, the case where x and y are not separated by a special node is solved by the recursive step.

The space required by the solution easily follows the recurrence $f(n) = O(1)$ for $n \leq 4$, and $f(n) = O(n) + \sqrt{n} f(\sqrt{n})$ otherwise. It is straightforward to show using induction that $f(n) = O(n \log \log n)$.

Similar to the case of two-hop solution, the existence of a three-hop path is not enough: we also need a constant-time algorithm for finding it. The FindPath$_3(x, y, G)$ procedure for doing this is very similar to the FindPath$_2(x, y, G)$ that we gave for the two-hop case. In more detail, we find

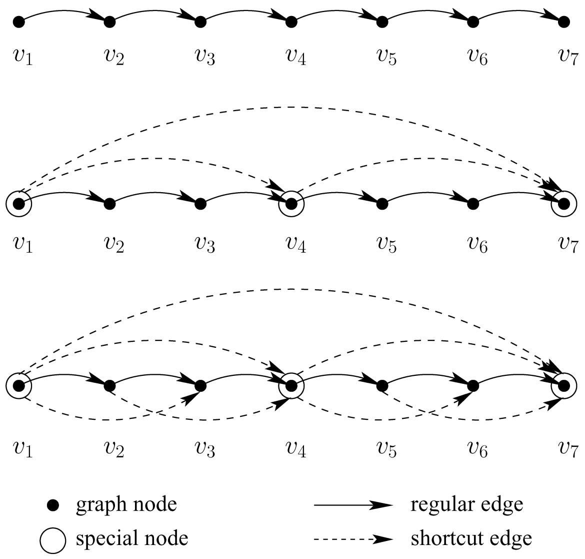

Fig. 4. Addition of shortcut edges for the three-hop one-dimensional solution.

the NCA (call it u) of the two leaves of RT that contain x and y, and the nodes x' and y', such that edges (x, x'), (x', y'), (y', y) are in G and are available at u (from the shortcut edges added in each of the steps 3 to 5 of the above AddShortcuts$_3(G)$ procedure).

8.2.3 *Four or more hop solutions.* The three-hop solution presented in the previous section gives us a template for designing schemes with three or more hops. Suppose that an h-hop solution ($h > 2$) is desired, then we can use the following algorithm for adding shortcuts to G:

AddShortcuts$_h(G)$:

(1) Let n denote the number of nodes in G. If $n \leq h+1$, then add edges between the consecutive nodes. Otherwise, proceed with the next step.
(2) Partition the nodes into n/m cells of size m each. Declare the last node in every cell to be a special node, and add all of the special nodes to a set S.
(3) Use the scheme that provides an $(h-2)$-hop solution to connect the nodes in S. That is, execute AddShortcuts$_{h-2}(G_S)$, where G_S consists of the nodes of S and the edges that define the relationship between such nodes.
(4) For each nonspecial node $v \notin S$, add an edge from it to its nearest special node after v.
(5) For each nonspecial node $v \notin S$, add an edge from the nearest special node before v (if one exists) to it.
(6) Recursively add shortcut edges to each cell (ignoring the special nodes) by executing this algorithm on each of them.

Table III. Performance of Shortcut Schemes for One-Dimensional Graphs

Scheme	Private Storage	Key Derivation	Public Storage
2HS	1	2 op.	$O(n \log n)$
3HS	1	3 op.	$O(n \log \log n)$
4HS	1	4 op.	$O(n \log^* n)$
$\log^*$HS	1	$O(\log^* n)$ op.	$O(n)$

The number of edges in the above algorithm follows the recurrence $f(n, h) = O(n) + f(n/m, h-2) + (n/m) f(m, h)$. For $h = 4$ and $m = \log n$, this leads to $f(n, 4) \leq O(n) + (n/\log n) f(\log n, 4)$ (recall that $f(n, 2) = O(n \log n)$). Now it is straightforward to show that $f(n, 4) = O(n\log^* n)$.

The constant-time procedure for computing the four-hop path between any two nodes is very similar to the one given for the two-hop case: The whole recursion tree RT is stored, and a constant-time NCA computation is used to get to the node of RT at which the nodes x' and y' of the four-hop path x, x', m', y', y can simply be read. The node m is retrieved from the recursion tree of G_S using NCA computation. For any general h, $\mathsf{FindPath}_h(x, y, G)$ will have $O(h)$ complexity.

8.2.4 *$O(\log^* n)$-hop solutions.* We briefly point out here that any $O(1)$-hop scheme of edge complexity $O(n\log^* n)$ (such as the scheme given in the previous section) can be used to build an $O(\log^* n)$-hop scheme of $O(n)$ edge complexity, as follows. Let S consist of every $(j\log^* n)$th node of the input total order, $1 \leq j \leq m = n/\log^* n$. This S induces a partition of the n-node chain into (at most) $m + 1$ chunks $C_1, C_2, \ldots, C_{m+1}$ of size $\leq \log^* n$ each. We build a linear chain of size m on S and use the constant-hop solution on that chain. This allows us to achieve the distance of 4 edges between nodes of S at an edge complexity of $O(m\log^* m)$, which is $O(n)$. The key derivation between two nodes in the same chunk is done in $\leq \log^* n$ hops by marching along the edges within that chunk. A derivation from a node x in chunk C_i to a node y in chunk C_j, $i < j$, is done by first (1) marching within C_i from v to the vertex $x' \in S$ that is at the boundary between C_i and $C_i + 1$; then (2) using a four-hop derivation within S to go from x' to the vertex $y' \in S$ that is at the boundary between C_{j-1} and C_j; and finally (3) marching within C_j from y' to y. The total number of hops in that case is therefore $\leq 2\log^* n + 4$.

8.2.5 *Summary of one-dimensional solutions.* Table III shows a summary of one-dimensional schemes described here. In the table, we denote by sHS a solution where the distance between any two nodes is at most s, that is, a so-called s-Hop Scheme.

To make the numbers more concrete, we performed simulation experiments to determine the minimum number of shortcut edges that are required to reduce the distance between nodes in an n-node one-dimensional graph to no more than h hops. In such experiments, we used the transitive closure and the scheme of Section 8.2.1 to achieve one-hop and two-hop graphs, respectively. For the simulations of schemes with more than two hops, we used the

Table IV. Number of Edges for h-Hop Solutions

No. of Nodes	Number of Hops									
	1	2	3	4	5	6	7	8	9	10
10	45	19	17	15	14	13	13	13	9	9
25	300	74	61	49	46	43	43	42	40	40
50	1225	193	146	119	110	98	95	92	92	91
100	4950	480	342	264	245	218	209	197	194	191
250	31125	1503	997	724	685	587	562	527	512	498
500	124750	3498	2173	1538	1427	1223	1184	1086	1061	1026
750	280875	5737	3408	2375	2186	1870	1804	1651	1620	1553
1000	499500	7987	4666	3241	2941	2537	2426	2222	2183	2085
2500	3123750	23417	12912	8652	7542	6618	6198	5704	5556	5298
5000	12497500	51822	27379	18144	15334	13651	12541	11617	11197	10703
10000	49995000	113631	57978	37950	31192	28143	25333	23650	22540	21616

generic scheme of Section 8.2.3. In the case of the generic scheme, to choose the number of groups to use, we performed an exhaustive search to find the number that minimized the number of edges. Table IV shows the number of edges from our simulation results for schemes with 1 to 10 hops.

8.3 Higher Dimensions

The idea behind building a solution for higher dimension is the addition of new dummy vertices that make it possible to add a small number of shortcuts to achieve the desired fast-key-derivation performance. Note that the dummy vertices and their associated keys are internal to the system (used purely for performance reasons) and that no access classes correspond to them. Unlike the solution given above, where shortcut edges were *in addition* to the original edges of the hierarchy, here the only explicit edges that remain are the shortcut edges (some of them may of course coincidentally correspond to edges in the original graph, but this is not required). The addition of dummy vertices and shortcut edges is a novel technique in this area, and we believe it has much promise beyond enabling the specific performance bounds that we achieve in this work.

We give a solution that achieves key derivation in no more than (and typically less than) $2(d-1)+h_1(n)$ steps (each of which corresponds to following one shortcut edge), where $h_1(n)$ denotes the number of hops between any two nodes in the underlying one-dimensional scheme (i.e., any of the above) for a graph with n nodes. The public space used in this scheme is $O(f_1(n)(\log n)^{d-1})$, where $f_1(n)$ denotes the space complexity (i.e., the number of edges) of the underlying one-dimensional scheme.

Rather than immediately giving the solution for arbitrary d, for expository reasons we choose to first present the solution for $d = 2$, because the two-dimensional case is easier to grasp intuitively than the higher-dimensional one. Once the basic idea has been presented (with good intuition) for $d = 2$, we give the general construction for arbitrary d.

8.3.1 *The case* $d = 2$. The fact that the graph G has dimension 2 implies that every vertex v can be replaced by a pair of numbers $(x(v), y(v))$, such that w is an ancestor of v in G if and only if w *dominates* v, that is, $x(w) \geq x(v)$ and

$y(w) \geq y(v)$. From now on, for convenience, we refer to "points" rather than "vertices." A *shortcut* is then an ordered pair of points w, v describing an extra "key-derivation edge" that will be added from point w to point v.

The input is a set V of n points in two-dimensional space, and the desired output includes a set S of shortcuts between pairs of points (some of which may not belong to V) such that (1) $|S| = O(f_1(n)\log n)$, and (2) given any pair of points $v, w \in V$ such that w dominates v, there is a path of at most $h_1(n) + 2$ shortcut edges from w to v. The output also includes the set P that contains V as well as the additional dummy points (i.e., points not in V but that are touched by edges in S).

The solution steps are as follows.

(1) Initialize $P = V$, and initialize S to be empty.

(2) If $|V| = 1$, then return P and S; otherwise continue with the next steps.

(3) If $|V| > 1$, then compute a median line M that is perpendicular to the y axis and partitions V into two equal sets V_1 and V_2, where V_1 (V_2) is left (resp., right) of M. Let V_1' (V_2') be the projection of V_1 (resp., V_2) on line M.

(4) Add to S the following shortcut edges:
–a shortcut edge from every point of V_1' to its corresponding point of V_1;
–a shortcut edge from every point of V_2 to its corresponding point of V_2'.

(5) Recursively build the shortcut edges and dummy points for the set V_1. Let that recursive call return P_1 as the set of points (including dummies) and S_1 as the set of shortcut edges within P_1. Update S and P as follows: $S = S \cup S_1$ and $P = P \cup P_1$.

(6) Recursively build the shortcut edges and dummy points for the set V_2. Let that recursive call return P_2 as the set of points (including dummies) and S_2 as the set of shortcut edges within P_2. Update S and P as follows: $S = S \cup S_2$, and $P = P \cup P_2$.

(7) Solve the one-dimensional problem consisting of $V_1' \cup V_2'$ using one of the schemes of Section 8.2. Let this return a set of edges S_3 (note that it returns only a set of edges, i.e., it does not add any dummy points). Update (i.e., augment) S as follows: $S = S \cup S_3$. (P stays the same.)

The space complexity (i.e., the number of shortcut edges and dummy points) of the above-described scheme obeys a recurrence of the form $f(n) \leq 2f(n/2) + cf_1(n)$ for some constant c if $n > 1$; and $f(n) = O(1)$ if $n = 1$. The resulting solution is $O(f_1(n)\log n)$. Note that this recurrence follows from step 4 (which recursively solves the problem for $(n/2)$ points), step 5 (which recursively solves the problem for $(n/2)$ points), step 7 which adds $f_1(n)$ edges, and step 1 which uses $O(n)$ points.

That any w-to-v number of shortcut edges is at most $h_1(n) + 2$ is proved by induction on n (the base case being trivial): If $w \in V_2$ and $v \in V_1$, then the path of length at most $h_1(n) + 2$ consists of following one edge from $w \in V_2$ to its projection $w' \in V_2'$, at most $h_1(n)$ edges from w' to the point $v' \in V_1'$ that is the projection of v on M, and one edge from v' to v. When both points v and w are in V_1 or both are in V_2, the claim follows from the induction hypothesis.

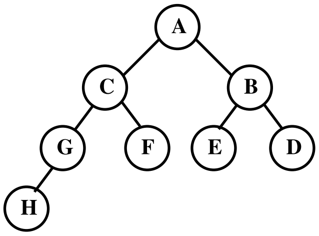

Fig. 5. Two dimensional access hierarchy (original).

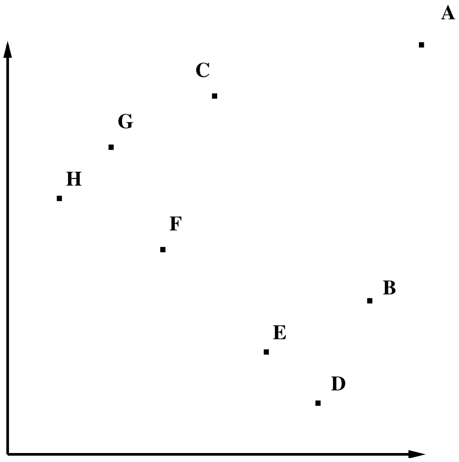

Fig. 6. Two dimensional access hierarchy (converted to tuple form).

Example. To help clarify our shortcut technique, we give an example of the recursive step in the previous section. Figure 5 shows a tree access hierarchy that will be used for this example.

Figure 6 contains a set of points in two dimensions that represents a tree's access structure. Note that if a point dominates another point in this figure, then the dominating point must have a path to the dominated point in the final structure.

Figure 7 shows the shadow points (added in step 3 and denoted by open circles) for the previous figure. Note that the shadow points are on a one dimensional plane (i.e., a line). This figure also shows the transitions from normal points to shadow points and vice versa (as described in step 4). Also note that the shadow points will be linked in step 7.

8.3.2 *The case $d \geq 3$.* The fact that the graph G has dimension d implies that every vertex v can be replaced by a d-tuple of numbers $(x_1(v), \ldots, x_d(v))$, such that w is an ancestor of v in G if and only if w *dominates* v, i.e., $x_i(w) \geq x_i(v)$ for all $i \in \{1, \ldots, d\}$.

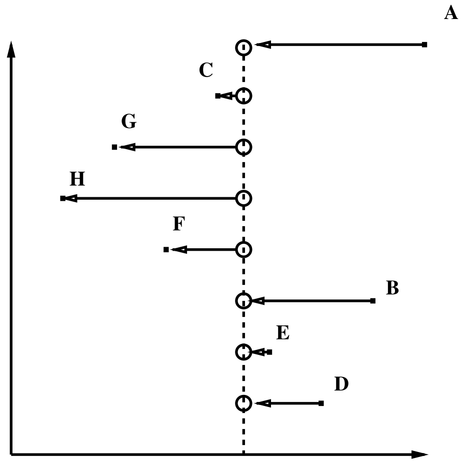

Fig. 7. Two dimensional hierarchy with shadow points.

The input is a set V of n d-dimensional points, and the desired output includes a set S of shortcuts between pairs of points (some of which may not belong to V) such that (1) $|S| = O(f_1(n)(\log n)^{d-1})$, and (2) given any pair of points $v, w \in V$ such that w dominates v, there is a path of $O(d+h_1(n))$ shortcut edges from w to v. The output also includes the set P that contains V as well as the additional dummy points (i.e., points not in V but that are touched by edges in S).

As we did for the two-dimensional case, the construction we use is recursive. Specifically, we inductively assume that the $d-1$ dimensional problem can be solved with $O(f_1(n)(\log n)^{d-2})$ edges and with a key derivation path of $2(d-1)+h_1(n)$ (note that this holds for $d = 1$ and for $d = 2$ by the previous subsections).

The solution steps are as follows:

(1) Initialize $P = V$, and initialize S to be empty.

(2) If $|V| = 1$, then return P and S, otherwise continue with the next steps.

(3) If $d = 1$, then solve using one of the one-dimensional schemes of the previous section, otherwise continue with the next steps.

(4) If $|V| > 1$, then compute a $d-1$ dimensional hyperplane M, perpendicular to the dth dimension, that partitions V into two equal sets V_1 and V_2, where V_1 is the set of points that are on the smaller side of the hyperplane (according to their dth coordinate). Let V_1' be the projection along dimension d of V_1 on hyperplane M. Let V_2' be the projection of V_2, along dimension d, on hyperplane M.

(5) Add to S the following shortcut edges:
–a shortcut edge from every point of V_1' to its corresponding point of V_1;
–a shortcut edge from every point of V_2 to its corresponding point of V_2'.

(6) Recursively build the shortcut edges and dummy points for the set V_1. Let that recursive call return P_1 as the set of points (including dummies) and

S_1 as the set of shortcut edges within P_1. Update S and P as follows: $S = S \cup S_1$ and $P = P \cup P_1$.

(7) Recursively build the shortcut edges and dummy points for the set V_2. Let that recursive call return P_2 as the set of points (including dummies) and S_2 as the set of shortcut edges within P_2. Update S and P as follows: $S = S \cup S_2$ and $P = P \cup P_2$.

(8) Solve the $d-1$ dimensional problem consisting of $V'_1 \cup V'_2$, using the solution for dimension $d-1$, and update P and S according to what this solution returns: If it returns S_3 and P_3, then the updates are $S = S \cup S_3$ and $P = P \cup P_3$.

The space complexity (i.e., the number of shortcut edges and dummy points) obeys the following recurrence. If $n > 1$, then:

$$f(n, 2) \leq c_1 f_1(n) \log n$$

and, if $d > 2$, then

$$f(n, d) \leq 2 f(n/2, d) + f(n, d-1) + c_2 dn$$

Note that this recurrence follows from steps 5 and 6 (which each recursively solve the problem for $n/2$ points in d dimensions), step 7 (which recursively solves a problem for n points in $d-1$ dimension), and the other steps add at most $O(n)$ points and edges.

Now, if $n = 1$, then $f(1, d) = c_3 d$. Thus, the solution to the above recurrence is

$$f(n, d) = O(d f_1(n)(\log n)^{d-1}).$$

The w-to-v number of shortcut edges obeys the following recurrence. If $n > 1$, then:

$$h(n, 2) \leq h_1(n) + 2$$

and, if $d > 2$, then

$$h(n, d) \leq 2 + h(n, d-1).$$

Note that the above recurrence follows from the following number of edges: one hop from V_2 to a shadow point, $h(n, d-1)$ hops on the $d-1$ dimensional hyperplane in step 7, and one hop from the shadow point to the destination point.

Now, if $n = 1$ then $h(1, d) = 1$. Thus, the solution to the above recurrence is

$$h(n, d) \leq 2(d-1) + h_1(n).$$

Table V summarizes the performance of our solution, when instantiated with different one-dimensional schemes. In the table, $h_1(n)$ and $h_d(n)$ denote the maximum distance between two nodes for one-dimensional and d-dimensional n-node graphs, respectively; and $f_1(n)$ and $f_d(n)$ denote the space complexity (i.e., the number of edges) for one-dimensional and d-dimensional graphs, respectively.

Table V. Performance of our Solution with Different One-Dimensional Schemes

One Dimensional Scheme		d-Dimensional Scheme	
$h_1(n)$	$f_1(n)$	$h_d(n)$	$f_d(n)$
1 edge	$O(n^2)$	$2d - 1$	$O(n^2(\log n)^{d-1})$
2 edges	$O(n \log n)$	$2d$	$O(n(\log n)^d)$
3 edges	$O(n \log\log n)$	$2d + 1$	$O(n(\log n)^{d-1} \log\log n)$
4 edges	$O(n \log^* n)$	$2d + 2$	$O(n(\log n)^{d-1} \log^* n)$
$O(\log^* n)$ edges	$O(n)$	$2(d - 1) + O(\log^* n)$	$O(n(\log n)^{d-1})$

8.3.3 *Using the data structure.* We also need a corresponding FindPath (x, y, G) procedure that, given two points x and y in V, finds a shortest path of shortcut edges from point x to point y. This subsection gives such a procedure (it is a simple generalization of the path-finding procedures we gave earlier for the one-dimensional case).

As before, we use RT to denote the recursion tree corresponding to the recursive calls of the procedure that adds shortcuts (given in the previous subsection); that is, in RT, the root corresponds to V, and the root's children correspond to the respective sets V_1 and V_2 that are separated by the hyperplane M. We henceforth use V_u to denote the set of points that correspond to a node u of RT, and V'_u to denote the projection of V_u on the hyperplane M_u that was used in the recursive call for u (of course $|V'_u| = |V_u|$, but V'_u has one dimension less than V_u). The height of RT is $h = \log n$ and its leaves correspond to sets of size 1 (as they correspond to the bottom of the recursion). We shall augment every node u in RT with an array Π_u that, for every point x of V_u, gives its projection $\Pi_u(x)$ on the hyperplane M_u that was used in the recursive call for u; we use V'_u to denote the projection of V on M_u. This takes space similar to the number of shortcut edges that were added at that particular node of RT, and provides a constant-time mechanism for following each such edge.

Note that a point $x \in V$ occurs in h sets like V_u, once at each depth i in RT, $1 \leq i \leq h$ (the root being at a depth of 1). In what follows, for every point $x \in V$ and $1 \leq i \leq h$, we use $N(x, i)$ to denote the node u of RT at depth i and whose V_u contains p. Note that $N(x, 1)$ is the root of RT, and that $N(x, h)$ is the leaf of RT that contains x.

The overall space complexity of RT is the same as the space complexity of the data structure created in the previous subsection (as the size of the array Π_u is equal to the number of shortcut edges that were added at node u of RT). We now turn our attention to how RT is used to trace a path of shortcut edges between two points.

As we did for the one-dimensional case, we shall make use of the fact that in a tree it is easy to answer *nearest common ancestor* (NCA) queries in constant time: given any two nodes of RT, their common ancestor in RT that is nearest to them, can be computed in constant time.

The following procedure takes as inputs two d-dimensional points $x, y \in V$ and, if x dominates y, returns a shortest path from x to y. In what follows, G'_u denotes the graph formed from the nodes of V'_u (preserving the partial order relationship between the nodes).

FindPath(x, y, G):

(1) Check in constant time whether x dominates y: If not then output "no path exists" and stop, otherwise continue with the next steps.
(2) If the dimension of (x, y, G) is 1, then use a one-dimensional path-finding procedure. Otherwise continue with the next steps.
(3) Let $v = N(x, h)$ and $w = N(y, h)$, that is, v (resp., w) is the leaf of RT whose corresponding set contains x (resp., y).
(4) Compute in constant time the nearest common ancestor (NCA) in RT of v and w, call it u. Note that p and q are both in V_u, and they are on different sides of the hyperplane M_u. Let $p' = \Pi_u(x)$, $q' = \Pi_u(y)$. The first edge on the path we seek is (x, x'), the last edge on it is (y', y), and the portion of it from x' to y' is of dimension $d - 1$ and can be computed as FindPath(x', y', G'_u).

The time taken by the above path-finding procedure is $h_1(n)$ for the base case, and constant per dimension-reduction round, hence a total of $O(d + h_1(n))$.

The FindPath technique we used in the above is widely applicable in other recursive solutions to shortcut-edge-adding procedures: Its essence is that a nearest common ancestor computation [Harel and Tarjan 1984] provides a constant-time jump to the relevant spot in the recursion tree, after which the problem becomes easy (we thereby avoid paying a price proportional to the height of the recursion tree).

8.4 Extension to Dynamic Hierarchies

Dynamic changes to the hierarchy (such as addition and deletion of nodes and edges, as well as a node's key replacement) do not require wholesale rekeying, rather, only the nodes directly affected by the change need rekeying. As described in Section 6, by changing the label, one can change a node's key. However, while individual nodes do not need to be rekeyed, the public information (dummy nodes and shortcut edges) does need recomputing after the access graph is modified. Because of the divide and conquer recursive nature of the algorithm, this recomputation looks amenable (at least for the case $d = 2$) to the techniques of dynamization of van Leeuwen and Overmars (see, e.g., van Leeuwen and Overmars [1981]; and Overmars and van Leeuwen [1981a; 1981b]). This looks like a promising direction for future work.

8.5 Connection to Graph Spanners

Recall that a k-spanner of a graph G (directed or undirected) is a subgraph G' of G such that the distance between every two vertices in G' is at most k times the distance between them in G [Peleg and Schaeffer 1989]. Taking G^* to be the transitive closure of our access graph, and k to be the number of hops we achieve in our key-derivation, our results can be interpreted as providing the construction of a sparse k-spanner for G^*, for the class of access graphs G that we consider. Most existing work on graph spanners is for undirected graphs, and the existing work on directed graphs does not provide bounds that compete with ours (but it is for more general graphs than ours). It was shown in Peleg and Schaeffer [1989] that for certain dense directed graphs *any*

k-spanner requires $\Omega(n^2)$ edges. The problem of computing the sparsest k-spanner of a graph (directed or undirected) is known to be NP-hard, and is $\Omega(\log n)$ inapproximable for $k = 2$ and $2^{\log^{1-\epsilon} n}$ inapproximable for $k > 2$. But for graphs with $O(n^{1+\beta})$ edges an approximation within a factor of $O(n^{(\beta+1)/3} \log^{\alpha})$ exists, with $0 < \beta \leq 1$ and $\alpha = O(1)$ [Elkin and Peleg 2005].

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, which can be significantly decreased for many types of graphs by increasing public information.

(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; and

(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 [Crampton 2003]. Additionally, we provided techniques for reducing the distance between nodes for the purposes of faster key derivation. For one-dimensional graphs and trees, lowering of key derivation time consists of inserting extra edges to the hierarchy, and for more general graphs of dimension d it consists of adding dummy vertices that permit dimension reduction. The fast performance achieved by our scheme is summarized in Table V.

Our work was extended in De Santis et al. [2007], and the primary improvements to our scheme were twofold: (1) the authors introduced a provably secure key derivation that relies only on one complexity assumption (secure symmetric encryption scheme) and (2) they introduced a new short-cutting technique for higher dimensional partial orders, there the number of shortcut edges is $O(3^{d-1} \cdot n \cdot d \cdot \log^{d-1} n \log\log n)$ and the distance between any two points is three.

REFERENCES

AKL, S. AND TAYLOR, P. 1983. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst. 1*, 3 (Sept.), 239–248.

ALON, N. AND SCHIEBER, B. 1987. Optimal preprocessing for answering on-line product queries. Tech. rep. TR 71/87, Institute of Computer Science, Tel-Aviv University.

ANDERSON, R. AND KUHN, M. 1996. Tamper resistance – a cautionary note. In *Proceedings of the USENIX Workshop on Electronic Commerce (EC'96)*. 1–11.

ANDERSON, R. AND KUHN, M. 1997. Low cost attacks on tamper resistant devices. In *Proceedings of the Security Protocols Workshop*. Lecture Notes on Computer Science, vol. 1361. 125–136.

ATALLAH, M., FRIKKEN, K., AND BLANTON, M. 2005. Dynamic and efficient key management for access hierarchies. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'05)*. 190–201.

BELL, D. AND LAPADULA, L. 1973. Secure computer systems: Mathematical foundations. Tech. rep. MTR–2547, MITRE Corporation.

BELLARE, M., CANETTI, R., AND KRAWCZYK, H. 1996. Keying hash functions for message authentication. In *Proceedings of the Annual International Cryptology Conference (CRYPTO'96)*, vol. 1109. 1–15.

BIRGET, J., ZOU, X., NOUBIR, G., AND RAMAMURTHY, B. 2001. Hierarchy-based access control in distributed environments. In *Proceedings of the IEEE International Conference on Communications (ICC'01)*. 229–233.

BODLAENDER, H., TEL, G., AND SANTORO, N. 1994. Trade-offs in non-reversing diameter. *Nordic J. Comput. 1*, 111–134.

CHANG, C. AND BUEHRER, D. 1993. Access control in a hierarchy using a one-way trapdoor function. *Comput. Math. Appl. 26*, 5, 71–76.

CHANG, C., LIN, I., TSAI, H., WANG, H., AND TAICHUNG, T. 2004. A key assignment scheme for controlling access in partially ordered user hierarchies. In *Proceedings of the International Conference on Advanced Information Networking and Application (AINA'04)*. 376–378.

CHAZELLE, B. 1987. Computing on a free tree via complexity-preserving mappings. *Algorithmica 2*, 337–361.

CHEN, T. AND CHUNG, Y. 2002. Hierarchical access control based on Chinese remainder theorem and symmetric algorithm. *Comput. Secur.* 565–570.

CHEN, T., CHUNG, Y., AND TIAN, C. 2004. A novel key management scheme for dynamic access control in a user hierarchy. In *Proceedings of the IEEE Annual International Computer Software and Applications Conference (COMPSAC'04)*. 396–401.

CHICK, G. AND TAVARES, S. 1990. Flexible access control with master keys. In *Proceedings of the Proceedings of the Annual International Cryptology Conference (CRYPTO'96)*. Lecture Notes on Computer Science, vol. 435. 316–322.

CHIEN, H. AND JAN, J. 2003. New hierarchical assignment without public key cryptography. *Comput. Secur. 22*, 6, 523–526.

CHOU, J., LIN, C., AND LEE, T. 2004. A novel hierarchical key management scheme based on quadratic residues. In *Proceedings of the International Symposium on Parallel and Distributed Processing and Applications (ISPA'04)*. Vol. 3358. 858–865.

CRAMER, R. AND SHOUP, V. 2003. Design and analysis of practical public-key encryption scheme secure against adaptive chosen ciphertext attack. *SIAM J. Comput. 33*, 1, 167–226.

CRAMPTON, J. 2003. On permissions, inheritance and role hierarchies. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'03)*. 85–92.

DAS, M., SAXENA, A., GULATI, V., AND PHATAK, D. 2005. Hierarchical key management scheme using polynomial interpolation. *SIGOPS Oper. Syst. Rev. 39*, 1, 40–47.

DE SANTIS, A., FERRARA, A., AND MASUCCI, B. 2007. Efficient provably-secure hierarchical key assignment schemes. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*. Lecture Notes on Computer Science, vol. 4708. 371–382.

DENNING, D., AKL, S., MORGENSTERN, M., AND NEUMANN, P. 1986. Views for multilevel database security. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'86)*. 156–172.

DODIS, Y., FAZIO, N., KIAYIAS, A., AND YUNG, M. 2005. Scalable public-key tracing and revoking. *J. Dist. Comput. 17*, 4, 323–347.

DUSHNIK, B. AND MILLER, E. 1941. Partially ordered sets. *American Journal of Mathematics 63*, 600–610.

ELKIN, M. AND PELEG, D. 2005. Approximating k-spanner problems for $k > 2$. *Theor. Comput. Sci. 337*, 1–3, 249–277.

FERRAIOLO, D. AND KUHN, D. 1992. Role based access control. In *Proceedings of the National Computer Security Conference (NISSC'92)*. 554–563.

FERRARA, A. AND MASUCCI, B. 2003. An information-theoretic approach to the access control problem. In *Proceedings of the Italian Conference on Theoretical Computer Science (ICTCS'03)*. vol. 2841. 342–354.

FRAIM, L. 1983. Scomp: A solution to multilevel security problem. *IEEE Comput. 16*, 7, 126–143.

GOLDREICH, O. 2004. *Foundations of Cryptography*. Vol. 2. Basic Applications.

HAREL, D. AND TARJAN, R. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13*, 2, 338–355.

HARN, L. AND LIN, H. 1990. A cryptographic key generation scheme for multilevel data security. *Comput. Secur. 9*, 6, 539–546.

HE, M., FAN, P., KADERALI, F., AND YUAN, D. 2003. Access key distribution scheme for level-based hierarchy. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'03)*. 942–945.

HUANG, H. AND CHANG, C. 2004. A new cryptographic key assignment scheme with time-constraint access control in a hierarchy. *Comput. Stand. Interfaces 26*, 159–166.

HWANG, M. 1999a. An improvement of novel cryptographic key assignment scheme for dynamic access control in a hierarchy. *IEICE Trans. Fundam. E82–A*, 2 (Mar.), 548–550.

HWANG, M. 1999b. A new dynamic key generation scheme for access control in a hierarchy. *Nordic J. Comput. 6*, 4, 363–371.

HWANG, M. AND YANG, W. 2003. Controlling access in large partially ordered hierarchies using cryptographic keys. *J. Syst. Softw. 67*, 2 (Aug.), 99–107.

LIAW, H., WANG, S., AND LEI, C. 1993. A dynamic cryptographic key assignment scheme in a tree structure. *Comput. Math. Appl. 25*, 6, 109–114.

LIN, C. 2001. Hierarchical key assignment without public-key cryptography. *Comput. Secur. 20*, 7, 612–619.

LIN, I., HWANG, M., AND CHANG, C. 2003. A new key assignment scheme for enforcing complicated access control policies in hierarchy. *Future Gen. Comput. Syst. 19*, 4, 457–462.

LU, W. AND SUNDARESHAN, M. 1988. A moredle for multilevel security in computer networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'88)*. 1095–1104.

MACKINNON, S., TAYLOR, P., MEIJER, H., AND AKL, S. 1985. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Trans. Comput. 34*, 9, 797–802.

MAHESHWARI, P. 2003. Enterprise application integration using a component-based architecture. In *Proceedings of the IEEE Annual International Computer Software and Applications Conference (COM-SAC'03)*. 557–563.

MCHUGH, J. AND MOORE, A. 1986. A security policy and formal top level specification for a multi-level secure local area network. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'86)*. 34–49.

OHTA, K., OKAMOTO, T., AND KOYAMA, K. 1991. Membership authentication for hierarchical multigroups using the extended fiat-shamir scheme. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EUROCRYPT'91)*. 446–457.

OVERMARS, M. AND VAN LEEUWEN, J. 1981a. Dynamization of order decomposable set problems. *J. Algorithms 2*, 3, 245–260.

OVERMARS, M. AND VAN LEEUWEN, J. 1981b. Maintenance of configurations in the plane. *J. Comput. Syst. Sci. 23*, 2, 166–204.

PELEG, D. AND SCHAEFFER, A. 1989. Graph spanners. *Theor. Comput. Sci. 13*, 99–116.

RAY, I., RAY, I., AND NARASIMHAMURTHI, N. 2002. A cryptographic solution to implement access control in a hierarchy and more. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT'02)*. 65–73.

ROSE, J. AND GASTEIGER, J. 1994. Hierarchical classification as an aid to database and hit-list browsing. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'94)*. 408–414.

SANDHU, R. 1987. On some cryptographic solutions for access control in a tree hierarchy. In *Proceedings of the Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow (CSC-ER'87)*. 405–410.

SANDHU, R. 1988. Cryptographic implementation of a tree hierarchy for access control. *Inform. Process. Lett. 27*, 2 (Jan.), 95–98.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control models. *IEEE Comput. 29*, 2, 38–47.

SANTIS, A. D., FERRARA, A., AND MASUCCI, B. 2004. Cryptographic key assignment schemes for any access control policy. *Inform. Process. Lett. 92*, 4 (Nov.), 199–205.

SCHNYDER, W. 1989. Planar graphs and poset dimension. *Order 5*, 323–343.

SHEN, V. AND CHEN, T. 2002. A novel key management scheme based on discrete logarithms and polynomial interpolations. *Comput. Secur. 21*, 2, 164–171.

SUN, Y. AND LIU, K. 2004. Scalable hierarchical access control in secure group communications. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*. 1296–1306.

THORUP, M. 1992. On shortcutting digraphs. In *Proceedings of the International Workshop on Graph=Theoretic Concepts in Computer Science (WG'92)*. 205–211.

THORUP, M. 1995. Shortcutting planar digraphs. *Comb. Probab. Comput. 4*, 287–315.

THORUP, M. 1997. Parallel shortcutting of rooted trees. *J. Algorithms 23*, 1, 139–159.

TROTTER, W. 1992. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins University Press, Baltimore, MD.

TSAI, H. AND CHANG, C. 1995. A cryptographic implementation for dynamic access control in a user hierarchy. *Comput. Secur. 14*, 2, 159–166.

TZENG, W. 2002. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *IEEE Trans. Knowl. Data Eng. 14*, 1, 182–188.

VAN LEEUWEN, J. AND OVERMARS, M. 1981. The art of dynamizing. *Math. Found. Comp. Sci.* 121–131.

WU, J. AND WEI, R. 2004. An access control scheme for partially ordered set hierarchy with provable security. Cryptology ePrint Archive, Report 2004/295. http://eprint.iacr.org/.

WU, T. AND CHANG, C. 2001. Cryptograpic key assignment scheme for hierarchical access control. *Int. J. Comput. Syst. Sci. Eng. 1*, 1, 25–28.

YANNAKAKIS, M. 1982. The complexity of the partial order dimension problem. *SIAM J. Algebraic Discrete Methods 3*, 351–358.

YAO, A. 1982. Space-time tradeoff for answering range queries. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'82)*. 128–136.

YEH, J., CHOW, R., AND NEWMAN, R. 1998. A key assignment for enforcing access control policy exceptions. In *Proceedings of the International Symposium on Internet Technology*. 54–59.

ZHANG, Q. AND WANG, Y. 2004. A centralized key management scheme for hierarchical access control. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'04)*. 2067–2071.

ZHENG, Y., HARDJONO, T., AND PIEPRZYK, J. 1992. Sibling intractable function families and their applications. In *Proceedings of Advances in Cryptology (ASIACRYPT'91)*. Lecture Notes on Computer Science, vol. 739. 124–138.

ZHENG, Y., HARDJONO, T., AND SEBERRY, J. 1993. New solutions to the problem of access control in a hierarchy. Tech. rep. Department of Computer Science, University of Wollongong.

ZHONG, S. 2002. A practical key management scheme for access control in a user hierarchy. *Comput. Secur. 21*, 8, 750–759.

Received April 2006; revised February 2008, June 2008; accepted September 2008