CERIAS Tech Report 2007-30

EFFICIENT KEY DERIVATION FOR ACCESS HIERARCHIES

by Mikhail Atallah, Marina Blanton, and Keith Frikken

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

Efficient Key Derivation for Access Hierarchies[*]

Mikhail J. Atallah[†]
Department of Computer Science
Purdue University
mja@cs.purdue.edu

Marina Blanton[‡]
Department of Computer Science
Purdue University
mbykova@cs.purdue.edu

Keith B. Frikken
Computer Science and Systems Analysis
Miami University
frikkekb@muohio.edu

Abstract

Access hierarchies are useful in many applications and are modeled as a set of access classes organized by a partial order. A user who obtains access to a class in such a hierarchy is entitled to access objects stored at that class, as well as objects stored at its descendant classes. Efficient schemes for this framework assign only one key to a class and use key derivation to permit access to descendant classes. Ideally, the key derivation uses simple primitives such as cryptographic hash computations. A straightforward key derivation time is then linear in the length of the path between the user's class and the class of the object that the user wants to access.

Recently, work presented in [Atallah et al. 2005] has given a solution that significantly lowers this key derivation time for deep hierarchies, by adding a modest number of extra edges to the hierarchy. While such techniques were given for trees, this work presents efficient key derivation techniques for hierarchies that are not trees using a different mechanism. The construction we give in the present paper is recursive and makes a novel use of the notion of the *dimension* d of an access graph. We provide a solution through which no key derivation requires more than $O(d)$ hash function computations, even for "unbalanced" hierarchies whose depth is linear in their number of access classes n.

The significance of this result is strengthened by the fact that many access graphs have a low d value (e.g., trees correspond to the case $d = 2$). Our scheme inherits the desirable property of the work of [Atallah et al. 2005] that addition and deletion of edges and nodes in the access hierarchy can be "contained" in the node and do not result in modification of keys at other nodes.

1 Introduction

The problem of key management for hierarchical access control is important for many applications and has received significant attention in the research literature. In this framework, all users are

[*]This report is an extended and corrected version of the work that appeared in ACM SACMAT 2006.

[†]Supported in part by Grants IIS-0325345 and CNS-0627488 from the National Science Foundation, and by sponsors of the Center for Education and Research in Information Assurance and Security.

[‡]Supported by Intel Ph.D. fellowship.

divided into disjoint access classes and each access class inherits the privileges of its descendants. Access is based on key derivation where users receive keys that allow them to obtain access to the authorized objects without interaction with the server, through a key derivation process. That is, for any given user, the key issued to that user allows her to access objects stored at her access class as well as objects stored at all descendant classes in the hierarchy.

Applications where such access hierarchies are useful include the Role-Based Access Control (RBAC) models, which are naturally modeled as hierarchies; repositories such as digital libraries, music collections, etc., where users are granted different access levels; subscription-based services, where their subscription packages are organized in a hierarchy; and others. Whereas organizations' role hierarchies tend to be shallow rather than deep, in a number of contexts the access hierarchies have a large size and depth. These include hierarchically organized distributed control structures such as physical plants or power grids (involving thousands of networked devices such as sensors, actuators, etc.); hierarchically organized hardware where the hierarchy is based on functional, control, and trust considerations; hierarchical design structures of large complex systems such as aircrafts and VLSI circuits; subscription services where the set of included programs or media depends on the subscription package type; and task graphs where descendant tasks are known only to their ancestor tasks. Deep access hierarchies can also arise in simple databases where the hierarchical complexity can come from super-imposed classifications on the database that are based on functional or structural features of the database. See also [12, 15] for other examples of deep hierarchies.

Normally, an access hierarchy can be modeled as a directed access graph G, where each node corresponds to an access class and edges preserve relations between the nodes. Recent key management schemes achieve the following properties:

- Each node in the access graph has a single secret key associated with it;
- The amount of public information for the key assignment scheme is asymptotically the same as that needed to represent the graph itself;
- Key derivation involves only the usage of efficient cryptographic primitives such as one-way hash functions;
- Given a key for node v, the key derivation for its descendant node w takes ℓ steps, where ℓ is the length of the path between v and w;
- It is impossible for dishonest colluding users to obtain access to more objects than what they can already legitimately access, ensuring collusion resilience of the scheme.

While the above is computation- and space-efficient, for large and deep hierarchies the key derivation process can require up to $O(n)$ steps, where n is the number of nodes in the graph. This operation must also be performed in real time. Thus, if such key derivation is performed by a device with computation and space limitations (such as cheap and possibly disposable smartcards), this key derivation computation will be larger than what the device can handle.

[2] recently provided techniques for reducing the key derivation time for tree hierarchies to $O(\log\log n)$ and $O(1)$ steps using two different schemes. The present paper significantly extends that work by showing how key derivation can be greatly reduced for more general access graphs, and gives techniques for achieving $O(d)$ key derivation steps, where d is the dimension of the graph (and many practical access graphs are of low dimension, typically less than 4). This scheme for

higher dimensions is recursive and utilizes a key derivation scheme for a one-dimensional graph. Specifically, our schemes adds a factor of $O((\log n)^{d-1})$ space complexity to that of the underlying one-dimensional solution, where n is the size of the hierarchy. Furthermore, we also make several contributions in the one-dimensional case providing new schemes and proving their bounds.

Our scheme inherits the desirable property of [2] that the addition and deletion of nodes in the access hierarchy can be "contained" in the node and do not result in the modification of keys at other nodes (similarly for edge additions and deletions); hence no wholesale re-keying is done as changes are made to the access hierarchy.

The rest of this paper is organized as follows: Section 2 provides a review of related literature. Section 3 gives a brief overview of the new technique. In Section 4 we present background information and definitions. Section 5 gives several solutions to the one-dimensional key derivation problem. Section 6 gives the details of how to build and use our approach, while Section 7 supplements it with comments on how dynamic changes to hierarchies can be addressed. Finally, Section 8 concludes this work.

2 Related Work

Prior research literature on hierarchical access control is very extensive, and its overview is beyond the scope of this paper (see, e.g., [2] and [8] for an overview). In this section, we thus focus on the publications most closely related to this work.

A number of recent schemes [2, 6, 7, 11, 22] give efficient solutions to the key assignment and management in access hierarchies. All of these schemes have similar overall structures with the following properties: (i) each node in the access graph has a single secret key; (ii) public information associated with the scheme is asymptotically the same as that needed to represent the access graph itself; (iii) key derivation involves only efficient operations such as one-way hash functions and bitwise XOR[1]; and (iv) the number of steps in key derivation is the same as the length of the path between the user's node and the target node (the node whose key is being derived). These constructions differ, resulting in some of them being more efficient and simpler than others, but the general structure remains the same[2].

Dynamic changes to the hierarchy are handled differently in these schemes: in [6, 7, 11, 22] insertions and re-keying are local, but deletions affect secret keys of all descendant nodes; in [2], however, all changes to the graph are local. In addition, only the scheme of [2] has a formal proof of security (in the presence of an active adversary who can adaptively corrupt nodes), while other schemes provide only informal discussion of attacks. Results of [11, 7] also assume tamper-resistance of clients.

Since the inspiration of the results reported in this work comes from the shortcut techniques of [2] (which allow to reduce the key derivation time for trees), here we give a brief description of the ideas underlying the shortcut techniques of [2]. In what follows, a shortcut edge is a new edge, which is being added to the graph G for efficiency reasons; while a shortcut edge is not in the original graph G, it is in the transitive closure of G. As was mentioned earlier, for trees (whose dimension is $d = 2$), [2] reports two results: (i) addition of $O(n)$ shortcut edges results in key derivation time being no more than $O(\log \log n)$ for n-node hierarchies, and (ii) addition of

[1]The scheme of [6] additionally utilizes symmetric key encryption.

[2]Not all properties of these approaches were stated in the above form, but for the sake of comparison, here we unify the notation to describe their capabilities.

$O(n \log \log n)$ shortcut edges results in key derivation time being no more than 3 steps. Here an edge from node v to node w means that w's key can be computed from v's key (and thus anyone with access to v can obtain access to w). We provide more details of the key derivation technique in Section 4.3.

One idea in [2] is to use the notion of a *centroid*: a centroid of an n-node tree T is a node whose removal from T leaves no connected component of size greater than $n/2$. In addition, a modified variant of *centroid decomposition* is used: to compute a centroid decomposition, compute the centroid of the tree, remove it, and then recursively repeat this process with the remaining trees. The modified variant used in that paper is the so-called "prematurely terminated centroid decomposition," which is similar to the above centroid decomposition, except that the recursion stops not when the tree becomes a single node, but rather when the tree size becomes $\leq \sqrt{n}$. The centroids used in this type of tree decomposition and the root of the tree are called the "special nodes." These special nodes are caused to become "well connected" with the addition of shortcuts, and the nodes of each small residual subtree of size $\sqrt{n}$ are also caused to become well connected with the addition of shortcuts. Then to reach any node from another node, we first need to reach the special node of the current residual tree, then jump to the special node of the target residual tree, and finally reach the target node within that small tree.

As non-tree hierarchies have no centroids, the above scheme does not readily extend to non-tree access hierarchies. The extension turns out to require a new, different technique. We briefly sketch it (at a high level) next.

3 Overview of the Techniques

The basis of our solution for the d-dimensional case is a reduction to the $d-1$-dimensional case. Thus, in this work we provide solutions to the one-dimensional case (Section 5) and then give a dimension reduction technique (Section 6).

For the one-dimensional case, we describe solutions where any two nodes are at most 2 edges away, 3 edges away, etc. For higher dimensions, the essence of our technique consists of three main components:

1. 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 [2], where shortcut edges were *in addition* to the original edges of the hierarchy, in this work 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.

2. As our technique could be cumbersome to apply to (and later use on) the original graph, we operate on a different representation of the graph. Namely, we need to "transform" the graph into a d-tuple-representation of the vertices where d is the dimension of the partial order represented by the graph. Such transformation step is not needed if the graph is already specified in the d-tuple form, e.g., policies of the form "node v is ancestor of node w iff v has both a higher value than w and is also less vulnerable than w." We believe this representation

of the access graph will have uses other than the present framework of key assignment and derivation.

3. With the above representation, it becomes possible to carry out the desired computation of dummy vertices and shortcut edges with very efficient performance. We provide an algorithm for achieving this and prove precise bounds for it, both in terms of its consumption of resources (time and space) and in terms of the key-derivation performance made possible by the data structure that it produces.

4 Background

4.1 User hierarchy

In our model, each user belongs to one of n disjoint access classes. The classes are organized into a hierarchy described by a directed acyclic graph G, i.e., by a partial order relationship. We denote the vertices of this graph, each of which corresponds to a specific access class, as $v_1, \ldots, v_n$. The users at access class v_i can access information at access class v_j if and only if there is a path in G from node v_i to node v_j.

The above-mentioned directed access graph $G = (V, E, O)$ is such that V is a set of vertices $V = \{v_1, \ldots, v_n\}$, E is a set of edges, and O is a set of objects. Each class v_i in the access hierarchy has a set of objects $O(v_i) \subset O$ associated with it, and thus a user who has privileges to access v_i obtains access to $\bigcup O(v_j)$ for each class v_j that is descendant of v_i (including v_i itself).

4.2 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, e.g., [9, 19]). 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 [21], 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 [16] that a G whose transitive reduction is planar has dimension at most 3 (and the 3-tuples representing it are computable in linear time). If the transitive reduction of G is 4-colorable, then its dimension is at most 4 [16]. Many access hierarchies are 4-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 paper 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.

4.3 Key derivation

The key derivation technique given in [2] consists of two algorithms: an algorithm to setup the system, Set, and an algorithm to derive a key, Derive. In what follows, $F : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$ denotes a family of pseudo-random functions (PRFs) that, on input a κ-bit key and a string, outputs a κ-bit string that is indistinguishable from random. Note that a PRF can be efficiently implemented using HMAC [3] or CBC MAC constructions.

The Set and Derive algorithms are then as follows:

$\mathsf{Set}(1^\kappa, G)$: For each node $v \in V$, select a random secret key $k_v \in \{0,1\}^\kappa$. For each node $v \in V$, select a unique label $\ell_v \in \{0,1\}^\kappa$ and make it publicly available. For each edge $(v,w) \in E$, compute $y_{v,w} = k_w \oplus F_{k_v}(\ell_w)$, where $\oplus$ denotes bitwise XOR, and make it publicly available.

$\mathsf{Derive}(v, w, k_v)$: Let $(v,w) \in E$. Then given k_v and public information, derivation of the key k_w can be performed as $k_w = F_{k_v}(\ell_w) \oplus y_{v,w}$, where ℓ_w and $y_{v,w}$ are publicly available. More generally, if there is a directed path between nodes v and u in G, then u's key can be derived from v's key by considering each edge on the path.

In other words, there is a public labels associated with every node in the graph, and there is a public information associated with every edge in the graph. This public information is what allows users to derive appropriate keys.

To avoid changing user keys when the hierarchy needs to be changed or a user class needs to be re-keyed, a slightly different version of the scheme should be used. We refer the reader to [2] for more detail.

The above key derivation mechanism is provably secure against key recovery. Usage of a different scheme with the same properties or a scheme with stronger security properties (e.g., security in the key indistinguishability setting) will imply different, normally higher, computational requirements.

5 The One-Dimensional Case

In this section we present our techniques for the one-dimensional case. In what follows, let the nodes (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."

Following the results of [2], we add extra, so-called shortcut, edges to the graph associated with the user hierarchy to lower key derivation time. Our goal is to compute a small set of shortcut edges such that the distance between any two nodes is minimized. Such shortcut edges should preserve the original relationship between the nodes in the graph and thus must satisfy the following constraints:

Reachability Given nodes v_i and v_j where $i < j$, there is a path from v_i to v_j.

Security There is a path from v_i to v_j only if $i < j$.

Given a set of edges E that satisfy the above constraints, 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, i.e., $\max_{v_i, v_j \in V, i<j} dist(v_i, v_j) \leq h$. 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.

5.1 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.

$\mathsf{AddShortcuts}_2(G)$:

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, i.e., 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).
5. Create a graph G_L from the nodes in L and execute $\mathsf{AddShortcuts}_2(G_L)$.
6. Similarly for R, create a graph G_R and execute $\mathsf{AddShortcuts}_2(G_R)$.

Figure 1 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 $f(n) = O(n \log n)$.

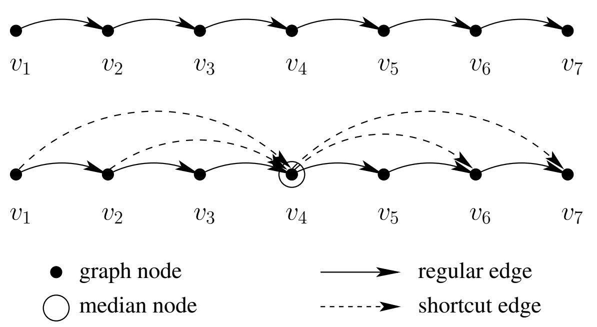

Figure 1: Addition of shortcut edges for the two-hop one-dimensional solution.

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 $\mathsf{FindPath}_2(x, y, G)$ 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 $\mathsf{AddShortcuts}_2$ 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 [10] 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.

5.2 Three-hop solutions

In this section, we describe a shortcut scheme where nodes are separated by at most three hops. [2] 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.

$\mathsf{AddShortcuts}_3(G)$:

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 $i_1 < i_2 < \cdots < i_m$.

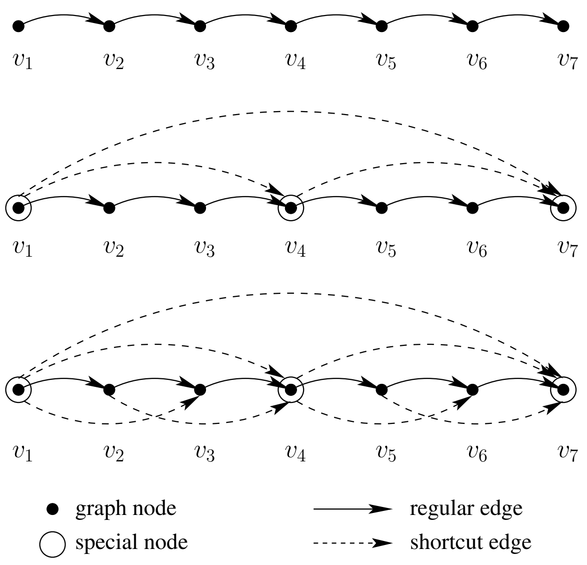

Figure 2: Addition of shortcut edges for the three-hop one-dimensional solution.

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 $\mathsf{AddShortcuts}_3$ on graphs $G_0, \ldots, G_{m-1}$ to recursively add shortcut edges to them.

Figure 2 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 $\mathsf{FindPath}_3(x, y, G)$ procedure for doing this is very similar to the $\mathsf{FindPath}_2(x, y, G)$ that we gave for the two-hop case. In more detail, we find 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 step 3 to 5 of the above $\mathsf{AddShortcuts}_3(G)$ procedure).

5.3 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:

$\mathsf{AddShortcuts}_h(G)$:

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 $\mathsf{AddShortcuts}_{h-2}(G_S)$, where G_S consists of the nodes of S and the edges that define the relationship between such nodes.
4. For each non-special node $v \notin S$, add an edge from it to its nearest special node after v.
5. For each non-special 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.

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.

5.4 $O(\log^* n)$-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

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.	$O(n)$

Table 1: Performance of shortcut schemes for one-dimensional graphs.

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 (i) 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 (ii) using a 4-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 (iii) marching within C_j from y' to y. The total number of hops in that case is therefore $\leq 2\log^* n + 4$.

5.5 Summary of one-dimensional solutions

Table 1 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, i.e., a so-called s-Hop Scheme.

Shortcut schemes have also been considered in prior literature ([5, 1, 4, 17, 18]), and the following results (explored in a different domain) are available for trees which also coincide with solutions for one-dimensional graphs. Lowering the diameter to 4 or 5 edges requires addition of $\Theta(n \log^* n)$ edges, lowering the diameter to 6 or 7 edges requires addition of $\Theta(n \log^{**} n)$ edges, etc.

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 5.1 to achieve 1-hop and 2-hop graphs, respectively. For the simulations of schemes with more than two hops, we used the generic scheme of Section 5.3. 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 2 shows the number of edges from our simulation results for schemes with 1 to 10 hops.

6 Higher Dimensions

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

No. of	Number of hops									
nodes	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

Table 2: Number of edges for h-hop solutions.

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.

6.1 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, i.e., $x(w) \geq x(v)$ 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 2-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 (i) $|S| = O(f_1(n) \log n)$, and (ii) 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 5. 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.

6.1.1 An example

To help clarify our shortcut technique, we give an example of the recursive step in the previous section. Figure 3 shows a tree access hierarchy that will be used for this example.

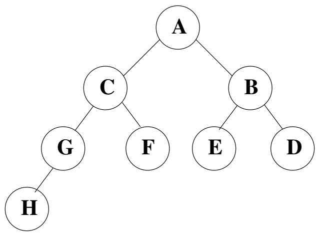

Figure 3: Two dimensional access hierarchy (original).

Figure 4 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 5 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.

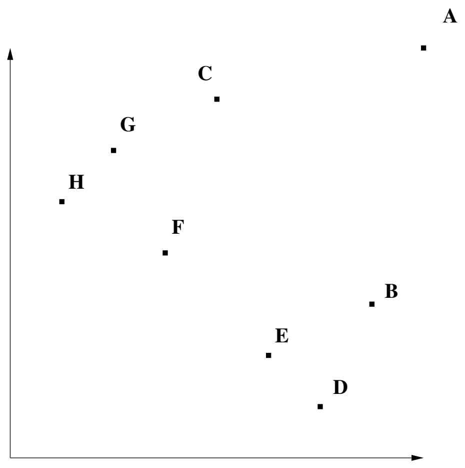

Figure 4: Two dimensional access hierarchy (converted to tuple form).

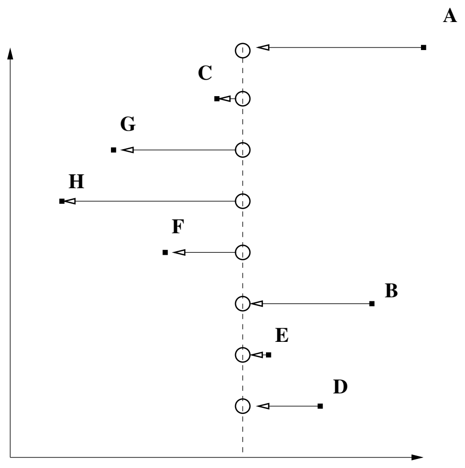

Figure 5: Two dimensional hierarchy with shadow points.

6.2 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 $i \in \{1, \ldots, d\}$.

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 (i) $|S| = O(f_1(n)(\log n)^{d-1})$, and (ii) 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 2-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

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

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:

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

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	$O(n)$	$2(d-1) + O(\log^* n)$	$O(n(\log n)^{d-1})$

Table 3: Performance of our solution with different one-dimensional schemes.

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 3 summarized 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.

6.3 Using the data structure

We also need a corresponding $\mathsf{FindPath}(x, y, G)$ procedure that, given two points x and y in V, finds a shortest path of shortcut edges from point x to point y. This sub-section 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 sub-section); 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 sub-section (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 $w = N(y, h)$, i.e., 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 [10] 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).

7 Extensions: 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 re-keying, rather, only the nodes directly affected by the change need re-keying. To achieve this, the ideas of [2] can be applied.

At a high level, we use two ideas from [2]. First, each subject is given their own key (and a separate new node), and the public storage is modified to include an edge from that subject's node to its access class v_i. The other idea is that the actual key of a node is computed as a function of a label and a secret key. And it follows that, by changing the label, one can change a node's key. For more information, we refer the reader to [2].

However, while individual nodes do not need to be rekeyed using the above techniques, 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 re-computation looks amenable (at least for the case $d = 2$) to the techniques of dynamization of van Leeuwen and Overmars (see, e.g., [20, 13, 14]). This looks like a promising direction for future work.

8 Conclusions and Future Research

We gave the first scheme for key derivation in a non-tree hierarchy that performs any derivation in a small number of efficient pseudo-random function computations. This performance bound holds even for key derivations between nodes separated by a path of linear length and comes at the cost of adding a modest number of extra edges to the hierarchy.

Promising directions for future work include reducing the space complexity and improving the performance of dynamic changes to the graph.

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, Institute of Computer Science, Tel-Aviv University, 1987.

[2] M. Atallah, K. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *ACM Conference on Computer and Communications Security (CCS'05)*, pages 190–201, 2005.

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

[4] H. Bodlaender, G. Tel, and N. Santoro. Trade-offs in non-reversing diameter. *Nordic Journal of Computing*, (1):111–134, 1994.

[5] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, (2):337–361, 1987.

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

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

[8] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006.

[9] B. Dushnik and E.W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63:600–610, 1941.

[10] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

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

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

[13] M.H. Overmars and J. van Leeuwen. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2(3):245–260, 1981.

[14] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and Systems Science*, 23(2):166–204, 1981.

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

[16] W. Schnyder. Planar graphs and poset dimension. *Order*, 5:323–343, 1989.

[17] M. Thorup. On shortcutting digraphs. *Combinatorics, Probability & Computing*, (4):287–315, 1995.

[18] M. Thorup. Parallel shortcutting of rooted trees. *Jounal of Algorithms*, (23):139–159, 1997.

[19] W.T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory.* Johns Hopkins University Press, Baltimore, MD, 1992.

[20] J. van Leeuwen and M.H. Overmars. The art of dynamizing. *Mathematical Foundations of Computer Science*, pages 121–131, 1981.

[21] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3:351–358, 1982.

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