Key Management in Hierarchical Access Control Systems

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

KEY MANAGEMENT IN HIERARCHICAL ACCESS CONTROL SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Marina V. Blanton

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2007

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

CA	Central authority
DAG	Directed acyclic graph
GIS	Geographic information system
KA	Key assignment
LBAC	Location-based access control
MAC	Message authentication code
NCA	Nearest common ancestor
NCR	US National Research Council
PPT	Probabilistic polynomial-time
PRF	Pseudo-random function
RBAC	Role-based access control
SMC	Secure multiparty computation
VLSI	Very-large-scale integration

ABSTRACT

Blanton, Marina V. Ph.D., Purdue University, August, 2007. Key Management in Hierarchical Access Control Systems. Major Professor: Mikhail J. Atallah.

In a hierarchical access control system, users are partitioned into a number of classes – called security classes – which are organized in a hierarchy. Hierarchies arise in systems where some users have higher privileges than others and a security class inherits the privileges of its descendant classes. The problem of key assignment in such systems is how to assign cryptographic keys to users and resources to properly enforce access rights. Its crucial goal is efficiency: the number of keys a user obtains, computation a user performs, and amount of resources the server is required to maintain should be minimized.

In this work, we present a fully-dynamic and very efficient solution to the key assignment problem that is also provably secure for a strong notion of security. We then show how the model can be extended to time-based policies where users obtain access rights only for a specific duration of time, and subsequently present our time-based key assignment solution. Finally, we explain how similar techniques can be used to efficiently enforce access control policies in geo-spatial systems and describe our construction for such systems as well.

1 INTRODUCTION

1.1 Access Control for User Hierarchies

The problem of key management in hierarchical access control is important for many applications and has received significant attention in the research literature. In this framework, all users are divided into disjoint access classes – so called *security classes* – and each access class has a set of resources associated with it. Hierarchies of classes arise when a class inherits the privileges of its subordinates, and thus a user at a specific class obtain access to the resources at her own class and all descendant classes in the hierarchy. The model is such that resources are kept encrypted under certain keys, and access to a decryption key implies access to the resources secured with that key.

In such systems, when a user joins a class, she is given secret information that will permit her to access resources at her own and all descendant classes in the hierarchy. For efficiency reasons, access is often based on *key derivation*: users receive one or a small number of keys that allow them to obtain access to the authorized objects without interacting with the server, through a key derivation process. 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, the objective of key management in hierarchical systems is to assign keys to users and resources such that access rights are *efficiently* and *correctly* enforced.

Efficiency in key management schemes is usually measured by a number of criteria, which are (in order of significance):

- The number of secret keys (or the size of private information) each user must store;

- The amount of computation a user needs to perform to obtain access to the desired resource;
- The work needed to perform when the hierarchy or the set of users change and the degree to which users' private keys are affected;
- The size of information the system must maintain.

Security of access control systems comes from their ability to deny access to unauthorized data. For example, in our model a user is not authorized to access resources stored at classes other than its own class and its descendants. Furthermore, it is normally necessary for a key assignment scheme to be *collusion-resilient*. This means that even if a number of users with access to different classes conspire, they cannot get access to more resources than what they can already legitimately access. Even though some key assignment schemes might be intended to be used with tamper-resistant hardware, a number of prior publications (*e.g.*, [1, 2]) suggest that compromising cards is easier than is commonly believed. In addition, the property of collusion-resilience allows us to use the schemes with other devices that do not have tamper-resistance.

There is a wide range of applications where such access hierarchies find its use. They include:

- Role-Based Access Control (RBAC) models, which are useful for many types of organizations with various access constraints. In such systems users are naturally organized in a hierarchy of classes, and a user higher up in the hierarchy inherits privileges of its descendant classes.
- Subscription-based services such as newspapers and magazines, pay TV, etc., where subscription packages are organized in a hierarchy based on the resources included in each package. For instance, a Gold package will include everything available in a Silver package and additional premium services; a user with access to the news and sports sections will be placed in a class which inherits privileges of both news-only and sports-only classes; etc.

- Digital repositories such as music collections, digital libraries, etc., where users are granted access at different levels.

- Project development, where users' views are organized in a hierarchy and each user obtains access to the resources determined by her role in the project. For example, the managerial view will include the views of developers assigned to the project and possibly other data.

- Defense in depth, where at each stage of intrusion defense there is a specific set of resources that can be accessed.

- Cryptographic directories or file systems, where access is similarly based on a hierarchical relationship between users.

and others. Even more broadly, hierarchical access control is used in operating systems (*e.g.*, [3]), databases (*e.g.*, [4]), and networking (*e.g.*, [5,6]).

Normally, an access hierarchy can be modeled as a directed access graph (DAG) G, where each node corresponds to an access class and edges preserve relations between the nodes (*i.e.*, in many cases hierarchies are more general than trees). A naive but simple solution to the key management problem in a hierarchy is then to assign a key to every access class, and to give a user the keys to all access classes that the user is entitled to access. Unfortunately, this solution might require each user to store a prohibitively large set of keys. Thus, key derivation mechanisms are used in the literature to address this issue.

Among the most efficient solutions in prior work are 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.

In this work we present a solution that preserves all of the above characteristics and is the first to additionally achieve the following properties:

- Provably secure even in the presence of collusion;
- Fully dynamic where no changes to the hierarchy affect secret keys of users;
- Simple and slightly more efficient that previous constructions.

In addition to showing security against *key recovery*, we define a stronger notion of security – *key indistinguishability*. Under this stronger definition, all keys a user is not authorized to have access to are pseudo-random and the user, even if she obtains a key of its ancestor class, is unable to verify its correctness.

1.2 Deep Hierarchies

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 hardware, where the hierarchy is based on functional, control, and trust considerations;
- 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 of large complex systems such as aircraft, VLSI circuits, etc.;
- 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 [7, 8] for other examples of deep hierarchies.

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 require the number of operations linear in the depth of the graph in the worst case, which for some graphs is $O(n)$, where n is the number of nodes in the access graph. Thus, a part of this work is concerned with reducing key derivation time by decreasing the distance between any two nodes in the hierarchy. This is achieved by inserting additional edges to the hierarchy. In addition to being useful for reducing key derivation in deep hierarchies, these techniques find their uses in other domains such as adding temporal capabilities to hierarchical systems, which we introduce in the next section.

As was mentioned above, efficiency is achieved by adding additional, so-called *shortcut*, edges to the hierarchy. Such shortcut edges preserve the partial order relationship between the nodes and are in the transitive closure of the graph. In other words, a shortcut is inserted between nodes v and u only if there is already a (directed) path between these nodes in the hierarchy. Carefully partitioning of the graph and insertion of such edges into trees and total orders allows us to reduce key derivation to 3 operations with addition of only $O(n \log\log n)$ extra edges. There is also a tradeoff between the number of extra edges, distance between nodes in the resulting graph, and complexity of the solution.

For more general graphs, efficient key derivation can be achieved by utilizing the notion of the dimension of a DAG. The definition of dimension and the exact bounds of our construction will be given later in this work.

1.3 Time-Based Access Control

Now consider the addition of time-based access control policies to hierarchical systems. That is, as before all users are divided into a set of disjoint classes, but now a user is granted access to a specific class for a period of time specified by its beginning and end. When a user joins the system, she is given a key (or a set of keys) which allows her to derive access keys for all resources she is entitled to have access (*i.e.*, at her own and descendant classes in the hierarchy) only during her time interval. Note that the time interval is user-specific and might be different for each user in the system.

There are many applications that follow this model and which would benefit from automatic enforcement of access policies with temporal constraints through efficient key management. Such applications, among others, include:

- RBAC models, where, in addition to having a hierarchy of user classes, users are normally granted their privileges for a specific period of time depending on their work schedule, which is well captured by our model.

- Subscription-based services such as digital libraries, music collections, digital subscriptions to newspapers and magazines, etc. Here a user may be able to join at any time and/or be able to specify the subscription duration, implying that access only during a specific time interval is allowed. Also, subscription packages are likely to be organized in a hierarchy.

- Content distribution, where users may join at any time and receive content of varying quality or resolution.

- Cable TV where, similarly, users join at arbitrary times and receive different programs based on what is included in their subscription package.

- Project development, where users' views are organized in a hierarchy and users can be assigned to a project only for a specific duration of time.

In all of the above examples, we use the current time to enforce time-based policies. Additionally, instead of being based on the current time, access control policies can be based on the time in the past and permit access to historical data. For example, a user might buy access to data such as historical transactions, prices, legal records, etc. for a specified time interval in the past, *e.g.*, the year of 1920. These different notions of time can be combined, *e.g.*, a user buys access to 1920 data and is entitled to access it for two weeks starting from today.

If we let the lifetime of a system be partitioned into m short time intervals, the existence of time-based access control policies requires the access (or encryption) keys to be changed at each time interval. In this work, we concentrate on applications where the system is setup to support a large number of such time intervals. For example, a video stream might be encrypted with a different key every day (thus, permitting users to subscribe on any given day) or the keys might change even more often. If the system is setup for a few years, this results in m being in thousands. Likewise, if the application of interest is access to historical data, say, for the last century, the number of time intervals will tend to be even higher. Thus, a small number of keys per user and efficient access with large m's is our goal.

The notion of security for time-based hierarchical key assignment (KA) schemes was formalized only recently by Ateniese *et al.* [9]. Thus, in this work we use their security definitions and provide a new efficient solution to the problem of key management in systems with time-based access control policies. Our solution does not impose any requirements or constraints on the mechanisms used to enforce policies in systems where access control is not time-based (*e.g.*, for a hierarchy of user classes). This means that our solution can be built on top of an existing scheme to make it capable of handling time (we refer to a scheme without the support for temporal access control as a *time-invariant* scheme, and we refer to a scheme that supports temporal access control policies as *time-based*). More precisely, we build a solution for a single access class, and additional functional requirements such as key derivation between hierarchically organized access classes can be performed through integration with a time-invariant scheme.

In a setup with m time intervals, the server is likely to maintain information which is linear in m. By building a novel data structure, we only slightly increase the storage space at the server beyond the necessary $O(m)$ and at the same time are able to achieve other attractive characteristics. In more detail, our solution enjoys the following properties:

- To be able to obtain access to an arbitrary contiguous set of time intervals, a user is required to store at most 3 keys.
- Key derivation within the authorized time intervals involves a small constant number of cryptographic operations and thus is independent of the number of time intervals in the systems or the number of time intervals in the user's access rights.
- The increase of the public storage space at the server due to our solution is only by a small asymptotic factor, *e.g.*, $O(\log^* m \log\log m)$ with a small constant.
- All operations are extremely efficient, and no expensive public-key cryptography is used.

We provide several solutions with slightly different characteristics, where the difference is due to the building blocks used in our construction.

While the results given above correspond to a time-based key assignment scheme with a single resource or user class, we can use them to construct a time-based key assignment scheme for a user hierarchy. We show that our construction favorably compares to existing schemes and provides an efficient solution to the problem. Additionally, our scheme is balanced in the sense that all resource consumption such as the client's private storage, computation to derive keys, and the server public storage are minimized with tradeoffs being possible. This allows the scheme to work even with very weak clients and not to burden the server with excessive storage. Furthermore, our scheme is provably secure under standard complexity assumptions.

1.4 Geo-Spatial Access Control

Finally, in this work we also address the problem of key assignment and derivation techniques for geo-spatial access control systems. Consider a system where each user is granted access rights to a specific area (or a set of areas). As this work is a first solution in addressing this problem in the geo-spatial domain, we consider the case where the user has access rights to a rectangular section of a larger grid. If a user's region is not rectangular, then it can be partitioned into a number of rectangles to each of which our techniques are applicable.

We envision many applications of this work, including (but not limited to) the following scenarios:

- Consider a physical facility that houses projects with different degrees of sensitivity/confidentiality, with each project assigned its own area. A specific employee might have access to certain areas of the building, but not to others. In this case, the users could be given a smartcard (or some other device) that can derive the access keys for the areas to which the user has access.

- Consider a geographic information system (GIS) that contains information (*e.g.*, demographic, marketing, etc.) about specific locations. This information may be interesting to researchers, commercial firms, and other entities. Thus key management could be used to provide a subscription-based service where users purchase access rights to the information about a specific geographic area.

- There could be a hybrid access control system based on not only location information, but also on role hierarchies, temporal constraints, or both. As an example, re-consider the first scenario above. It is a reasonable access control policy that a senior researcher might be able to access a specific room all of the time, but a consultant might be able to access the same room during her work hours. As our scheme extends to higher dimensions, it can be used in such hybrid frameworks (with time as an additional dimension).

The key derivation scheme we introduce in this work for geo-spatial grids uses a novel data structure and achieves the following characteristics for a grid composed of $n_1 \times n_2$ cells (where $n_1 \geq n_2$):

- To obtain access to an arbitrary rectangular subsection, a user is required to store a constant number of keys.
- Key derivation within the authorized rectangle involves a constant number of operations (including cryptographic operations).
- The public storage space at the server due to our solution is only $O(n_1 n_2 \log^* n_1 \times (\log\log n_1)^2)$ with a small constant involved in the "$O(\cdot)$" notation.
- All cryptographic operations are very efficient, and no expensive public-key cryptography is required.

1.5 Organization

Chapter 2 describes literature related to all aspects of this work. In Chapter 3 we formally define and present a key assignment scheme for user hierarchies. The solution consists of a basic scheme, which is then extended to support a stronger notion of security and full dynamism. In Chapter 4, we present techniques for lowering key derivation time in deep hierarchies at the expense of increased public storage space at the server. We also show possible space-time tradeoffs.

Chapter 5 extends the time-invariant solution to support temporal constraints. We describe a scheme that can be used with a single access class and then combine it with our time-invariant scheme. Next, Chapter 6 shows how enforcement of access rights in the geo-spatial domain can be achieved. Finally, Chapter 7 summarizes our results and concludes this work.

Most of the material presented here is a joint work with Mikhail Atallah and Keith Frikken. The work presented in Chapter 3 is also a joint work with Nelly Fazio. In particular, the key indistinguishability security model, the scheme modification and the security proof in that model are due to Nelly. Parts of this work appeared in [10–13], and are a copyrighted material under the ACM and Springer copyright agreements.

2 RELATED WORK

2.1 Key Assignment in Hierarchical Systems

The first work that addressed the problem of key management in hierarchical access control was by Akl and Taylor [14]. Since then a large number of publications ([15–43] 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. Also, the work of Crampton at el. [44] provides a more formal categorization of existing schemes.

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 [45–49] or incorrect [50]. Therefore, we do not take these schemes into consideration in our further discussion.

A significant number of schemes, *e.g.*, [14,16,19,24,25,27,28,31–34,37], operate on large numbers computed as a product of up to $O(n)$ coprime numbers or, alternatively, up to $O(n)$ large numbers, where n is the number of nodes in the graph. Such numbers can grow to n bits long and are prohibitively large for large hierarchies. While in many of these approaches key derivation might seem to consist 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. More efficient solutions (including ours) have key derivation bounded by the depth of the access hierarchy and can be implemented using $O(n)$ hash operations in the worst case (*i.e.*, when the depth of the hierarchy is $O(n)$).

Work of [29,35,36] is limited to trees and thus is of limited use. Work of [15,38,40] is concerned with a slightly different model having a hierarchy of users and a hierarchy of resources. The scheme of [15], however, is not dynamic; and in [38,40] 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 [23] 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 [51] uses modular exponentiation, and additions/deletions require rekeying of all descendants. A number of schemes [17, 22, 39] are based on interpolating polynomials and give reasonable performance. In [22, 39], however, private storage at a node is up to $O(n)$ and additions/deletions require rekeying of ancestors. As was already mentioned above, we avoid rekeying on additions/deletions and store only one key per node. In [17], 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 and similar schemes), 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) [41, 42] are the only efficient approaches among early schemes. In these schemes, there is only one secret key per class, key derivation is a chain of SIFF function applications which can be implemented using polynomials. However, additions and deletions in [41] require rekeying of all descendants and in [42] all descendants should be rekeyed when a node is deleted.

A number of recent schemes [18, 20, 21, 30, 43] use overall structure similar to ours and have performance comparable to our base scheme. We briefly outline the dif-

Table 2.1
Comparison of our hierarchical scheme with previous work.

Scheme	Private storage	Public storage	Key derivation	Changes I/D/R	Proof of security
Lin [30]	k	$2k\|E\|$	$(3c_{\mathsf{H}} + 4c_{\mathsf{XOR}})\ell$	L/NL/L	No
Zhong [43]	k	$(k + k_1)\|E\|$	$(c_{\mathsf{H}} + 2c_{\mathsf{XOR}})\ell$	L/NL/L	No
Chien and Jan [20]	k	$k\|E\|$	$(c_{\mathsf{H}} + c_{\mathsf{XOR}})\ell$	L/NL/L	No
Chen *et al.* [18]	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

ferences between these solutions and ours. The work of Chou *et al.* [21] does not address dynamic changes, and the scheme additionally uses modular multiplication. The scheme of Chen et al. [18] requires larger public storage, key derivation additionally uses encryption, and the ex-member problem is not addressed (which will require to rekey all descendants on deletions). Compared to the schemes of Lin [30] and Zhong [43], 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. The scheme of Chien and Jan [20] uses only hash functions and achieves performance closest to our base scheme; deletions, however, require rekeying of all descendants. In our scheme dynamic changes to the graph are handled locally (*i.e.*, private information at other nodes is not affected). 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.

Table 2.1 gives a comparison of our basic scheme and other schemes in the literature. 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[1]; c_{XOR} corresponds to computation needed to perform bitwise XOR of two strings; and c_{D} is computation needed for symmetric key decryption. In the table, changes to the hierarchy include insertion (I), deletion (D), and re-keying (R); L stands for "local" and NL for "non-local." In all of the schemes that list "non-local" for deletions, such operations require re-keying of all descendant classes in the hierarchy.

Note that in different schemes, the authors might make assumptions on what information is public and what is stored with the client, which differs from what we present here. For the sake of comparison, however, we unify the schemes and list their capabilities, which may or may not be different from the results reported by the authors. In addition, results of [20, 30] rely on tamper-resistance of the clients.

Most recent developments. As a followup on this work, De Santis *et al.* [52] recently developed new solutions to the problem of key assignment in the hierarchy. Their schemes achieve comparable performance, but rely on a single computational assumption to achieve security with respect to key indistinguishability.

Among other results that appeared after publication of our initial work [10] are results of Zych *et al.* [53], Fuh-gwo and Chun-ming [54], and Vadnala and Mathuria [55]. Here we give a brief description of these schemes.

Construction of Zych *et al.* [53] involves transforming the hierarchy into a special form where each node has two parents (called *cover various V-posets*), assigning to each node a public-private key pair, and letting two parents to derive a child's

[1] Our solution uses a family of pseudo-random functions (PRFs) F to achieve provable security instead of using H directly. PRFs, however, can be implemented using solely a hash function, and for the sake of uniformity we list c_{H} for our scheme as well.

key using Diffie-Hellman key exchange protocol. The core difference of this scheme from previous solutions is that there is no public information associated with edges, but rather only public keys associated with nodes. The authors show that any n-node hierarchy can be transformed into the desired form by using $O(n^{2\frac{\log\log n}{\log n}})$ extra nodes in the worst case, and the average-case performance tends to result in addition of $O(n)$ nodes. Thus, the public storage is lower than schemes that associate a public label with an edge resulting in $O(n^2)$ worst-case public storage. The scheme's disadvantage is in slower key derivation: now deriving a child's key involves modulo exponentiation and the distance between nodes increases due to insertion of additional nodes. Changes to the hierarchy are not addressed.

Work of Fuh-gwo and Chung-ming [54] builds on Rabin public-key encryption scheme [56], where each class has a different public-private key parameters. The solution, however, is unnecessarily complicated, requires each user to store $O(n)$ private keys (basically keys of all descendant classes in the hierarchy), giving no advantage over the straightforward solution where each class maintains keys of its descendant classes. Changes to the hierarchy require users to be re-keyed, plus the security analysis stated in the paper is incorrect.[2]

Finally, work of Vadnala and Mathuria [55] gives a solution based on hash functions. In their scheme the key of class j is computed from its parents' keys as:

$$K_j = H(H(K_{i_1}||i_1||\ldots||i_m||j)||H(K_{i_2}||i_1||\ldots||i_m||j)||\ldots||H(K_{i_m}||i_1||\ldots||i_m||j)||R_j)$$

where $i_1, \ldots, i_m$ are j's parents, R_j is a random number associated with class j, and $||$ denotes concatenation. Key derivation is thus $O(n)$ cryptographic operations regardless of the depth of the hierarchy. Furthermore, changes to the hierarchy require user re-keying (while our solution does not). Finally, the authors state that their

[2] More precisely, a class's access key is stored encrypted using Rabin encryption, and the authors postulate that using 10-digit decimal numbers for prime factors p and q is sufficient to achieve security. Decryption of a message in Rabin scheme, however, uses purely computation modulo p or modulo q and thus simply trying all 10^{10} possibilities for p and q will give access to a class's key.

scheme has lower public storage space requirements compared with our scheme when the hierarchy is a tree, while the public storage of the schemes is equivalent.[3]

2.2 Time-Based Key Assignment in Hierarchical Systems

While the list of publications on time-invariant KA schemes is very large, the number of publications that consider time-based policies and provide schemes for them is rather modest. The time-based setting and the first scheme was introduced by Tzeng [48]. The scheme, however, was later shown to be insecure against collusion of multiple users [57]. Subsequent work of Huang and Chang [49], Chien [58], and Yeh [59] was also shown to be insecure against collusion (in [60], [61, 62], and [9], respectively).

Among very recent publications, Wang and Laih [63] present a time-based hierarchical KA scheme. While their scheme is shown to be collusion-resilient, the notion of security, however, is not formalized and no clear adversarial model is given in that work. Tzeng [64] also describes a time-based hierarchical key assignment scheme, which is used as a part of an anonymous subscription system. The scheme is proven to resist collusion attacks; however, no formal model of adversarial behavior is provided. The work of Ateniese at el. [9] is the first result that provides a formal framework for time-based hierarchical KA schemes and gives provably secure solutions, both secure against key recovery and achieving pseudo-random keys. Also, concurrently with and independently from this work, De Santis *et al.* [65] provide new solutions with pseudo-random keys that exhibit performance similar to that of our scheme.

There is extensive literature on broadcast encryption and multicast security, which might be considered applicable in this setting as well. There are, however, crucial differences in the models, which prevent us from simply using existing solutions from

[3] Public storage associated with our scheme for an n-node tree is n node labels (one per node) and $n-1$ edge labels (one per edge). In the scheme of [55] there are n random numbers (one per node) and n node idenitifiers. Since labels in our solution play the same role as node identifiers in scheme of [55], there is no benefit in public storage in either scheme.

those domains. First of all, literature on broadcast encryption and multicast security provides solutions that permits access to a single resource instead of a hierarchy and they cannot be composed in an obvious way to provide a solution to our problem. More importantly, they assume that each client obtains an updated key for each time interval, which is impossible in our model: no private channels between the server and a client after the initial issuance of the user keys is assumed, the client is allowed to remain off-line, and can access the resources at its own discretion.

The only exception from the above online requirement that we are aware of is the work of Briscoe on multicast key management [66]. That solution builds a binary tree from the time intervals, thus achieving $O(\log n)$ secret keys and $O(\log n)$ key derivation time. Our solution, on the other hand, provides a constant number of secret keys and a constant derivation time, thus resulting in superior performance, which additionally is independent of the number of time intervals in the system.

Time-based key assignment can also be considered related to time-evolving cryptography such as forward-secure and backward-secure encryption and signature schemes. In such schemes, access to a secret key allows the owner to decrypt (respectively, sign) messages corresponding to all past or all future time intervals (i.e, access is unidirectional in time). In our setting, however, unidirectional key derivation is not sufficient, which significantly complicates the problem. Thus, the closest to our setting are the schemes that combine forward and backward security, such as key-insulated cryptography [67–69] and intrusion-resilient models [70, 71]. The mode of operation in key-insulated and intrusion-resilient constructions, however, is such that at each time interval a trusted device transmits a part of the key corresponding to the current time interval, which allows the user to combine it with another part she has and derive the current secret key. In our problem, on the other hand, an off-line mode of operation is assumed, where no interaction with other parties can be used to derive the key for the current time interval. In addition, we do not assume trusted devices or parties once the keys are issued to the user.

Also, key updating schemes in cryptographic storage systems such as those in [72, 73] give various constructions for performing key updates and derivation. Similar to forward-secure solutions, the current key allows users to derive keys for all past time intervals with unidirectional key derivation, which is insufficient in our case.

2.3 Access Control in Geo-Spatial Systems

Using location information for access control, *i.e.*, location-based access control (LBAC), is not a new concept. The major challenges in geo-spatial computing were covered in the summary of the recent NRC's "IT Roadmap to a Geo-Spatial Future" [74]. One of the issues mentioned as a future challenge are "fine-grained access control mechanisms permitting the precise release of location information to just the right parties under the right circumstances." Atluri and Chun [75] propose a new model that supports privilege modes specific to geo-spatial data and includes geometric considerations (such at the region of overlap between an authorization and an access request). Similarly Bertino et al. [76] extend the RBAC model to GEO-RBAC, a model that can deal with geo-spatial information. Other previous work includes efficiently tracking the location of a user [77], models for representing and evaluating LBAC conditions [78], answering database queries based on location [79], the introduction of architectures for supporting location-aware applications [80], and many other important problems. However, we are not aware of any key management schemes that implement geo-spatial access control policies.

3 KEY ASSIGNMENT IN HIERARCHICAL SYSTEMS

This chapter defines the problem of key assignment in hierarchical systems and then presents our solutions.

3.1 Problem Definition

Suppose that a user hierarchy is modeled as a directed access graph $G = (V,\ E,\ O)$, where V is a set of vertices of size $|V| = n$, E is a set of edges, and O is a set of objects associated with the hierarchy. Each vertex $v \in V$ 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 $|\mathcal{O}(v)| \geq 0$ and $\forall v \in V\, \forall w \in V,\ \mathcal{O}(v) \cap \mathcal{O}(w) = \emptyset$ iff $v \neq w$. (For brevity, we use notation $\mathcal{O}_v$ to mean $\mathcal{O}(v)$.) 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 $Anc(v, G)$ which is a set such that $w \in Anc(v, G)$ if there is a path from w to v in G. We also define the set of descendants of node v as $Desc(v, G)$, where $w \in Desc(v, G)$ if there is a path from v to w in G. For a directed graph $G = (V, E)$, we use a function $Pred(v, G)$ to denote the set of immediate predecessors of v in G, *i.e.*, if $w \in Pred(v, G)$ then there is a directed edge from w to v in G. Similarly, we define $Succ(v, G)$ to be the set of immediate successors of v in G. When it is clear what graph we are discussing, we may omit G from the notation and instead use the shorthand notation $Anc(v)$, $Desc(v)$, $Succ(v)$, and $Pred(v)$. 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 to node w (*i.e.*, v is an ancestor of w) means that any subject that can assume access rights at class v is also permitted to access any object $o \in \mathcal{O}_w$ at class w. The function $\mathcal{O}^* : V \rightarrow 2^O$ maps a node $v \in V$ to a set of objects accessible to a subject at class v (we use $\mathcal{O}^*_v$ as a shorthand for $\mathcal{O}^*(v)$). This function is defined as $\mathcal{O}^*_v = \bigcup_{w \in Desc(v)} \mathcal{O}_w$.

Intuitively, a key allocation mechanism aims at implementing such form of access control by assigning a cryptographic key k_v to each class v. Such key k_v is then used to guard access to objects of class v (for example, by encrypting object $o \in \mathcal{O}_v$ under key k_v), and is made available to every user at class v (and at any of its ancestor classes). It follows that each user ought to store (or be able to derive) the cryptographic key k_v associated with the class v to which he belongs, as well as the keys k_w of all classes w which are descendants of v. For the sake of generality, we do not impose any specific structure on the secret information actually stored by users at class v; we denote such information by S_v.

In summary, S_v denotes the secret information that each user at class v stores, while k_v (which is derivable from S_v) is the cryptographic key necessary to gain access to objects at class v. We formalize this intuition with the following definition.

Definition 3.1.1 *A Key Allocation (KA) scheme is a pair of polynomial-time algorithms* $(\mathsf{Set}, \mathsf{Derive})$*, defined as follows:*

- $\mathsf{Set}(1^\kappa, G)$ *is a randomized algorithm that on input a security parameter* 1^κ *and an access graph* G*, outputs two mappings: (i) a public mapping* $\mathsf{Pub} : V \cup E \rightarrow \{0,1\}^*$*, associating a public label* ℓ_v *to each node* v *and a public label* $y_{v,w}$ *to each edge* (v, w) *in the graph; (ii) a secret mapping* $\mathsf{Sec} : V \rightarrow \{0,1\}^\kappa \times \{0,1\}^\kappa$*, associating a secret information* S_v *and a cryptographic key* k_v *to each node* v *in* G*. (No secret information is associated to edges in* G*.)*

- $\mathsf{Derive}(G, \mathsf{Pub}, v, w, S_v)$ *is a deterministic algorithm taking as input the access graph* G*, the public information* Pub *output by* Set*, a source node* v*, a target node* w *and the secret information* S_v *of node* v*. It outputs the cryptographic* *key k_w associated to node w if $w \in Desc(v)$, 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: $\forall v \in V$, $\forall w \in Desc(v)$,*

$$\Pr\left[k_w = \mathsf{Derive}(G, \mathsf{Pub}, v, w, S_v) \;\middle|\; \begin{array}{c} (\mathsf{Pub}, \mathsf{Sec}) \leftarrow \mathsf{Set}(1^\kappa, G), \\ (S_v, k_v) \leftarrow \mathsf{Sec}(v), \\ (S_w, k_w) \leftarrow \mathsf{Sec}(w) \end{array} \right] = 1$$

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

We now formalize two levels of security: *key recovery* and *key indistinguishability.*

Definition 3.1.2 (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 κ) against the challenger in the following game:*

- **Setup***: The challenger runs* $\mathsf{Set}(1^\kappa, G)$*, 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* $\mathsf{Corrupt}(v)$ *queries, which the challenger answers by retrieving (S_v, k_v) = $\mathsf{Sec}(v)$ and giving S_v to $\mathcal{A}$.*
- **Break***: The adversary outputs a node v^*, subject to $v^* \notin Desc(v)$ for any v asked in* **Phase 1***, along with her best guess k'_{v^*} to the cryptographic key k_{v^*} associated with node v^*.*

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

Definition 3.1.3 (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 κ) against the challenger in the following game:*

- **Setup**: *The challenger runs* $\mathsf{Set}(1^\kappa, G)$*, 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* $\mathsf{Corrupt}(v)$ *queries, which the challenger answers by retrieving* $(S_v, k_v) = \mathsf{Sec}(v)$ *and giving* S_v *to* $\mathcal{A}$*.*

- **Challenge**: *Once the adversary decides that* **Phase 1** *is over, it specifies a node* v^**, subject to* $v^* \notin Desc(v)$ *for any* v *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* κ*.*

- **Phase 2**: *The adversary can issue more* $\mathsf{Corrupt}(v)$ *queries, obtaining back the corresponding key* S_v*. Note that* $\mathcal{A}$ *cannot ask* $\mathsf{Corrupt}(v)$ *queries for* $v \in Anc(v^*)$*.*

- **Guess**: *The adversary outputs a bit* $b \in \{0,1\}$ *as her best guess to whether she was given the actual key* k_{v^*} *or a random key.* $\mathcal{A}$ *wins the game if* $b = b^*$*.*

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

Remark. In formalizing the security of a key allocation scheme, $\mathsf{Corrupt}$ queries are answered with respect to the secret info S_v, 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*).

3.2 Base Scheme

This section describes our scheme in which every node has one key associated with it, the public information is linear in the size of the access graph G, and computation by node v of a key that is ℓ levels below it can be done in ℓ evaluations of a pseudo-random function. Note that pseudo-random functions can be efficiently implemented as, *e.g.*, HMAC [81] built using only a cryptographic hash function or CBCMAC that utilizes symmetric key encryption. Here we focus on key allocations for a static access hierarchy. An extension of this scheme is given in Section 3.3, and its support for dynamic access hierarchies is discussed in Section 3.4.

Our construction is based on the use of pseudo-random functions, which we define next.

Definition 3.2.1 (Pseudo-Random Function (PRF) Family) *Let $\{F^\kappa\}_{\kappa\in\mathbb{N}}$ be a family of functions where $F^\kappa : K^\kappa \times D^\kappa \rightarrow R^\kappa$. For $k \in K^\kappa$, denote by $F^\kappa_k : D^\kappa \rightarrow R^\kappa$ the function defined by $F^\kappa_k(x) = F^\kappa(k, x)$. Let $Rand^\kappa$ denote the family of all functions from D^κ to R^κ,* i.e., *$Rand^\kappa = \{g \mid g : D^\kappa \rightarrow R^\kappa\}$.*

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

Experiment $\boldsymbol{Exp}^{PRF-1}_{F,\mathcal{A}}(\kappa)$	*Experiment* $\boldsymbol{Exp}^{PRF-0}_{F,\mathcal{A}}(\kappa)$
$k \overset{R}{\leftarrow} K^\kappa$	$g \overset{R}{\leftarrow} Rand^\kappa$
$d \leftarrow \mathcal{A}^{F^\kappa_k}(1^\kappa)$	$d \leftarrow \mathcal{A}^{g}(1^\kappa)$
Return d	*Return d*

The PRF-advantage of $\mathcal{A}$ is then defined as:

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

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

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

- For each vertex $v \in V$, pick a unique label $\ell_v \in \{0,1\}^\kappa$ and a random value $S_v \in \{0,1\}^\kappa$, then set $k_v = S_v$.
- For each edge $(v, w) \in E$, compute $y_{v,w} = k_w \oplus F(k_v, \ell_w)$, where $\oplus$ denotes bitwise XOR.

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

$$\mathsf{Pub} : v \mapsto \ell_v \qquad \mathsf{Pub} : (v, w) \mapsto y_{v,w}$$
$$\mathsf{Sec} : v \mapsto (S_v, k_v)$$

When a user joins an access class v, she is given secret value S_v for her class. If a user is assigned access to several classes $V' \subseteq V$, she is given keys for each of her access classes $w \in V'$.

We now describe the Derive algorithm. Suppose a user has access to class v and is in possession of the corresponding key k_v. Then to obtain the cryptographic key k_w of a descendant node w, the user sequentially processes every edge (v_i, v_j) on the path between v and w. Given an edge (v_i, v_j) for which both v_i's private key k_{v_i} and the stored public information ℓ_{v_i} and y_{v_i,v_j} are known, we can compute v_j's private information k_{v_j} as $k_{v_j} = F(k_{v_i}, \ell_{v_j}) \oplus y_{v_i,v_j}$, using the fact that $y_{v_i,v_j} = k_{v_j} \oplus F(k_{v_i}, \ell_{v_j})$.

Due to the sequential nature of key generation on the path between v and w, the user will be able to derive keys of all necessary nodes and produce key k_w.

Example. Figure 3.1 shows key allocation for a graph more complicated than a tree. First, it is possible for the node with key 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

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

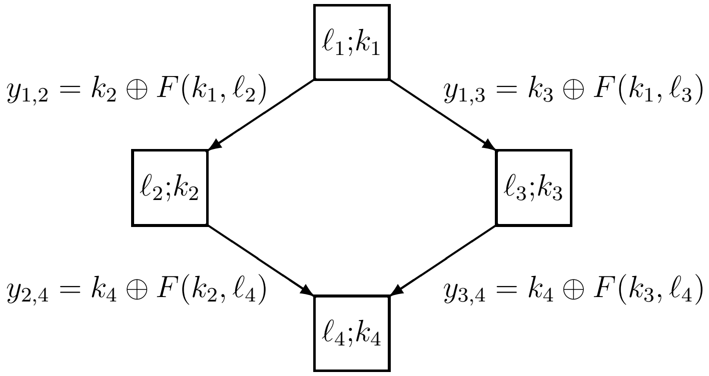

$\ell_1;k_1$
$y_{1,2} = k_2 \oplus F(k_1, \ell_2)$
$y_{1,3} = k_3 \oplus F(k_1, \ell_3)$
$\ell_2;k_2$
$\ell_3;k_3$
$y_{2,4} = k_4 \oplus F(k_2, \ell_4)$
$y_{3,4} = k_4 \oplus F(k_3, \ell_4)$
$\ell_4;k_4$

Figure 3.1. Key allocation in the base hierarchical scheme for an example access graph.

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 3.2.1 *The above scheme is secure against key-recovery (per Definition 3.1.2) for a DAG G, assuming the security of the pseudo-random function family $\{F^\kappa\}_{\kappa \in \mathbb{N}}$ as given in Definition 3.2.1.*

Proof In the security proof, we will follow the same structural approach used in [82], first advocated in [83]. 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 is by breaking the security of the pseudo-random function F. To accomplish this, we need to show how to turn an adversary $\mathcal{A}$ attacking the scheme into an adversary $\mathcal{B}_F$ attacking F.

One difficulty with this approach is that whereas $\mathcal{A}$ can choose which part of the public info to attack (via the challenge query), the adversaries $\mathcal{B}_F$ does not have such flexibility. The standard way to solve this technical problem is to "guess" the node v^* for which adversary $\mathcal{A}$ will ask the challenge query and construct adversaries $\mathcal{B}_F$ based on the assumption that this guess is correct. In the rest of the proof, we will assume that we correctly guessed the challenge node v^*. Since such a priori guess is correct with $1/n$ probability, 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. Also, let $v_1, \ldots, v_h$ 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 3.1.2), 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 then follow that in the original game the adversary's advantage is also 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 $\mathbf{G}_j$. We then define the games as follows:

Game $\mathbf{G}_0$. Define $\mathbf{G}_0$ to be the original game as described in Definition 3.1.2.

Game $\mathbf{G}_1$. This game is identical to game **Game $\mathbf{G}_0$**, except that in $\mathbf{G}_1$ the $\mathsf{Set}(1^\kappa, G)$ 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 $\{0,1\}^\kappa$, *i.e.*, $y_{v_1,v_j} \overset{R}{\leftarrow} \{0,1\}^\kappa$.

Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F(k_{v_1}, \cdot)$ in $\mathbf{G}_0$ with a truly random function. Since k_{v_1}

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

$$\Big|\Pr[T_1] - \Pr[T_0]\Big| \leq \epsilon_{PRF} \tag{3.1}$$

where ϵ_{PRF} is the (negligible) advantage $\mathsf{Adv}^{PRF}_{F,\mathcal{B}_F}(\kappa)$ of any PPT adversary $\mathcal{B}_F$ against the security of the pseudo-random function F. We now generalize the description of game $\mathbf{G}_1$ to any game in the sequence $\mathbf{G}_1$, ..., $\mathbf{G}_h$.

Game $\mathbf{G}_i$ $(1 \leq i \leq h)$. This game is identical to game $\mathbf{G}_{i-1}$, except that the $\mathsf{Set}(1^\kappa, G)$ algorithm is modified in such a way that the secret key k_{v_i} of node v_i is never used in the creation of the public information. Observe that the cryptographic key k_{v_i} occurs in game $\mathbf{G}_{i-1}$ only as the key to the pseudo-random function $F(\cdot,\cdot)$. In particular, no information about k_{v_i} is present in the public information associated with the edges going into node v_i due to the modifications carried out in games $\mathbf{G}_0$, ..., $\mathbf{G}_{i-1}$ 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_{v_i,v_j} at random from $\{0,1\}^\kappa$ (rather than computing it as prescribed). Such modification amounts to substituting all occurrences of $F(k_{v_i}, \cdot)$ in $\mathbf{G}_{i-1}$ with a truly random function. Since k_{v_i} does not occur anywhere else in $\mathbf{G}_{i-1}$, we can conclude (as above) that such modification is warranted by the security of the PRF family $\{F^\kappa\}_{\kappa\in\mathbb{N}}$, *i.e.*:

$$\Big|\Pr[T_i] - \Pr[T_{i-1}]\Big| \leq \epsilon_{PRF} \tag{3.2}$$

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 $1/2^\kappa$, *i.e.*:

$$\Pr[T_h] = \frac{1}{2^\kappa} \tag{3.3}$$

Combining Equation 3.3 with the intermediate results in Equation 3.2, we can conclude that:

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

□

3.3 The Extended Scheme

We now present an extension of the scheme described in Section 3.2 and prove it secure w.r.t. key indistinguishability (according to Definition 3.1.3), without random oracles. An important feature of this extended scheme is that it permits arbitrary changes to the hierarchy without affecting secret information stored by the users.

The scheme maintains essentially the same parameters as the one in Section 3.2: every node has only one random κ-bit key associated with it; 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, computation consists of ℓ efficient operations. In addition to using PRFs, the extended scheme makes use of a secure[2] encryption scheme $\mathcal{E}$: we denote with Enc and Dec the corresponding encryption and decryption algorithms.

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

- For each vertex $v \in V$, first pick a unique label $\ell_v \in \{0,1\}^\kappa$ and a random value $S_v \in \{0,1\}^\kappa$; then compute $t_v = F_{S_v}(0||\ell_v)$ and $k_v = F_{S_v}(1||\ell_v)$.
- For each edge $(v, w) \in E$, compute $r_{v,w} = F_{t_v}(\ell_v)$ and $y_{v,w} = \mathsf{Enc}_{r_{v,w}}(t_w||k_w)$.

[2] We require the encryption scheme to be chosen ciphertext secure; see the definition in [84].

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

$$\begin{aligned} &\mathsf{Pub} : v \mapsto \ell_v \qquad \mathsf{Pub} : (v,w) \mapsto y_{v,w} \\ &\mathsf{Sec} : v \mapsto (S_v, k_v) \end{aligned}$$

When a user joins a class v, she is then given secret value S_v associated with that class.

We now describe the Derive algorithm. Given G, the corresponding public information Pub, a source node v, a target node w and the secret information S_v of node v, the cryptographic key k_w of node w is derived by considering each edge on the path[3] from v down to w in turn, and repeatedly decrypting the public info associated to such edge. More precisely, $\mathsf{Derive}(G, \mathsf{Pub}, v, w, S_v)$ proceeds as follows:

- If there is no path from v to w in G, return $\perp$;
- If $v = w$, retrieve ℓ_v from Pub and return $k_w \leftarrow F_{S_v}(1||\ell_v)$;
- Else compute $t_v \leftarrow F_{S_v}(0||\ell_v)$ and let $v_i = v$ and $t_{v_i} = t_v$; then

```
repeat
    let $v_j$ be the successor of $v_i$ in the path from $v$ to $w$;
    retrieve $\ell_{v_i}$ and $y_{v_i,v_j}$ from $\mathsf{Pub}$;
    $r_{v_i,v_j} \leftarrow F_{t_{v_i}}(\ell_{v_j})$;
    $t_{v_j}||k_{v_j} \leftarrow \mathsf{Dec}_{r_{v_i,v_j}}(y_{v_i,v_j})$;
    $v_i \leftarrow v_j$;   $t_{v_i} = t_{v_j}$;
until $v_j = w$;
return $k_w$.
```

Figure 3.2 shows how the key derivation mechanism works for the same toy example given in Figure 3.1.

[3] If there is more than one path, pick one arbitrarily, *e.g.*, the shortest path from v to w.

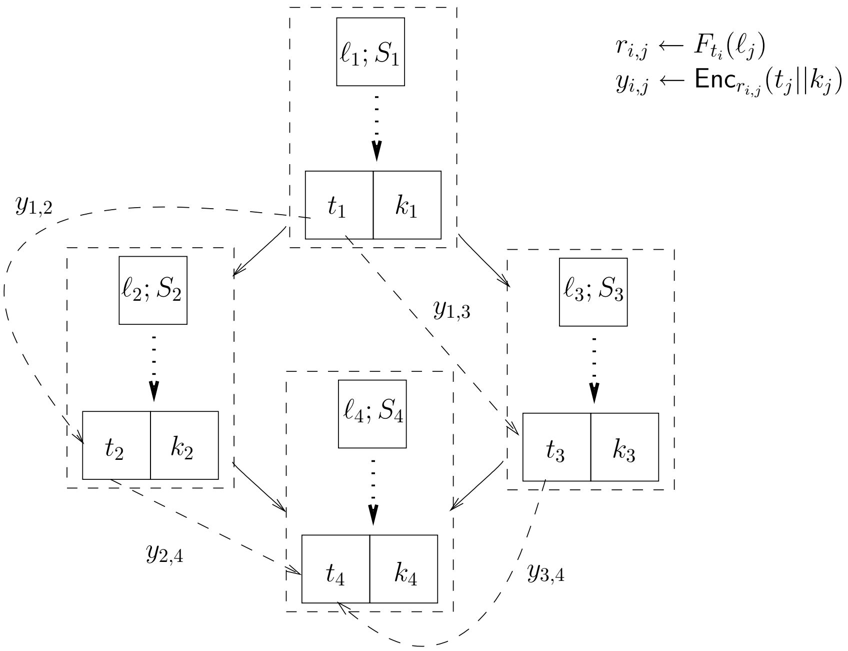


Figure 3.2. Key allocation in the extended hierarchical scheme for an example access graph.

Next, we prove that the extended scheme described above is key indistinguishable (per Definition 3.1.3), following the same approach as in the proof of Theorem 3.2.1.

Theorem 3.3.1 *The above extended scheme is key indistinguishable for any DAG G, assuming the security of the pseudo-random function family $\{F^\kappa\}_{\kappa\in\mathbb{N}}$ 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 property of the extended scheme is by either breaking the pseudo-random function F or the encryption scheme $\mathcal{E}$. To accomplish this, we need to show how to turn an adversary $\mathcal{A}$ attacking the scheme into either an adversary $\mathcal{B}_F$ attacking F or an adversary $\mathcal{B}_\mathcal{E}$ attacking $\mathcal{E}$.

One difficulty with this approach is that whereas $\mathcal{A}$ can choose which part of the public info to attack (via the challenge query), the adversaries $\mathcal{B}_F$ and $\mathcal{B}_\mathcal{E}$ do not have such flexibility. As noted in Theorem 3.2.1, the standard way to solve this is to "guess" the node v^* for which adversary $\mathcal{A}$ will ask the challenge query and construct adversaries $\mathcal{B}_F$ (or $\mathcal{B}_\mathcal{E}$) based on the assumption that this guess is correct. Thus 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 $1/n$ probability, 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 3.1.3), 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 3.1.3. 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 the general case is addressed.

First special case. v^* *is one of the roots*[4] *in* G.

Game $\mathbf{G}_0$. Define $\mathbf{G}_0$ to be the original game as described in Definition 3.1.3.

Game $\mathbf{G}_1$. This game is identical to game $\mathbf{G}_0$, except that in $\mathbf{G}_1$ the $\mathsf{Set}(1^\kappa, G)$ 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 achieve this, we compute

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

where $R_1 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function. Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{S_{v^*}}(\cdot)$ with a truly random function $R_1(\cdot)$. Since S_{v^*} does not occur anywhere else in the attack game, such modification is warranted by the security of a pseudo-random function. In other words, using a standard reduction argument, any non-negligible difference in behavior between game $\mathbf{G}_0$ and $\mathbf{G}_1$ can be used to construct a PPT

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

algorithm $\mathcal{B}_F$ that is able to break the pseudo-random function F with non-negligible advantage. Hence,

$$\big|\Pr[T_1] - \Pr[T_0]\big| \leq \epsilon_{PRF} \tag{3.4}$$

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

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

$$\Pr[T_1] = 1/2 \tag{3.5}$$

Combining Equations 3.4 and 3.5, the thesis follows.

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

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

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

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

where $R_1 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function. Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{S_p}(\cdot)$ with a truly random function $R_1(\cdot)$. Since S_p does not occur anywhere else in the attack game, such modification is warranted by the security of a pseudo-random function; hence,

$$\big|\Pr[T_2^{(a)}] - \Pr[T_1]\big| \leq \epsilon_{PRF} \tag{3.6}$$

.

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

$$r_{p,s} \leftarrow R_2(\ell_s)$$

where $R_2 : \{0,1\}^* \rightarrow \{0,1\}^*$ is a truly random function. Note that such modification essentially amounts to substituting any occurrences of the pseudo-random function $F_{t_p}(\cdot)$ with a truly random function $R_2(\cdot)$, which is safe since p is a root of G, and thus t_p does not occur anywhere else in the adversarial view (in particular, it is not encrypted within any label in Pub). Therefore, using a standard reduction argument, any non-negligible difference in behavior between game $\mathbf{G}_1$ and $\mathbf{G}_2^{(b)}$ can be used to construct a PPT algorithm $\mathcal{B}_F$ that is able to break the pseudo-random function F with non-negligible advantage. Hence,

$$\left| \Pr[T_2^{(b)}] - \Pr[T_2^{(a)}] \right| \leq \epsilon_{PRF} \tag{3.7}$$

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

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

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 (because of the changes in game $\mathbf{G}_2^{(b)}$). Therefore, using a standard reduction argument, any non-negligible difference in behavior between games $\mathbf{G}_2^{(b)}$ and $\mathbf{G}_2^{(c)}$ can be used to construct a PPT algorithm $\mathcal{B}_{\mathcal{E}}$ that is able to break the security of the encryption scheme $\mathcal{E}$ with non-negligible advantage. Hence,

$$\left| \Pr[T_2^{(c)}] - \Pr[T_2^{(b)}] \right| \leq \epsilon_{Enc} \tag{3.8}$$

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

It remains to notice that in game $\mathbf{G}_2^{(c)}$, the challenge no longer contains any information about b^*. This is because the label y_{p,v^*} of the only incoming edge

$(p, v^*) \in E$ no longer contains any information about k_{v^*} (after the modification in this game), which is therefore independent from the adversary view. Thus,

$$\Pr[T_2^{(c)}] = 1/2 \tag{3.9}$$

Combining Equations 3.4, 3.6, 3.7, 3.8 and 3.9, the thesis follows.

The general case. *There are no restrictions on the position of v^* in G.*
The second special case demonstrated how to remove from the adversary view's the information on k_{v^*} (which could be leaked by the label y_{p,v^*} 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 to v^*.

To accomplish this, we start the sequence of games with **Game $\mathbf{G}_0$** and **Game $\mathbf{G}_1$** defined as in the first special case. Then for each 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 of the ancestors of v^*, until we reach v^* itself, according to the following rules:

1. A node can be pebbled only after all its ancestors have already been pebbled.

2. To pebble a node u, we introduce the games $\mathbf{G}_u^{(a)}$, $\mathbf{G}_u^{(b)}$ and $\mathbf{G}_u^{(c)}$ in the sequence, following the same approach employed in the second special case. In particular, first we define a game $\mathbf{G}_u^{(a)}$ in which the secret information t_u is computed as:

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

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

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

   $$r_{u,s} \leftarrow R_u^{(b)}(\ell_s)$$

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

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

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

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

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

$$\Pr[T_{v^*}^{(c)}] = 1/2 \tag{3.10}$$

Combining all of the intermediate equations, we can conclude that

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

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

3.4 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 3.3.

Insertion of an edge. Suppose the edge (v, w) is to be inserted into G. First, compute $r_{v,w} = F_{t_v}(\ell_w)$ and $y_{v,w} = \mathsf{Enc}_{r_{v,w}}(t_w||k_w)$. Then, augment Pub to contain the mapping $(v, w) \mapsto y_{v,w}$.

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

1. Change the label of u, call it ℓ'_u. Note that S_u remains unchanged, but the keys t_u and k_u need to be recomputed as $t'_u = F_{S_u}(0||\ell'_u)$ and $k'_u \doteq F_{S_u}(1||\ell'_u)$.

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

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

1. Create the node v without any incoming or outgoing edges; this requires just generating a unique public label $\ell_v \in \{0,1\}^\kappa$ and a random secret value $S_v \in \{0,1\}^\kappa$, computing $k_v = F_{S_v}(1||\ell_v)$ and augmenting Pub with the mapping $v \mapsto \ell_v$ and Sec with the mapping $v \mapsto (S_v, k_v)$.

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

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

1. Deletion of all of the edges coming into and out of v, using the above procedure for edge deletion.

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

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

1. Update the secret information S_v with a new random value $S'_v \overset{R}{\leftarrow} \{0,1\}^\kappa$.

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

3. Update Sec to map $v \mapsto (S'_v, k'_v)$.

4. For each edge (w, v) (*i.e.*, where $w \in \mathit{Pred}(v)$), compute $y'_{w,v}$ according to the new keys t'_v and k'_v and updates Pub to map $(w, v) \mapsto y'_{w,v}$.

5. For each edge (v, u) (*i.e.*, where $u \in Succ(v)$), compute $y'_{v,u}$ according to the new key t'_v and update Pub to map $(v, u) \mapsto y'_{v,u}$.

No node other than v 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, [43] considers only a hierarchy of security classes without mentioning individual users, and [30] considers a hierarchy of users without grouping them into classes. However, it is important to group users with the same privileges together and on the other hand permit revocation of individual users. In our scheme, revoking a single user can be done with two approaches:

1. Recard every user at that user's access class(es), and for all descendants of this access class(es) perform the operation described for edge deletion (*i.e.*, change all keys by changing the labels and then update the public information). Note that the descendants do not have to be rekeyed.

2. Make the access graph such that each user is represented by a single node in the graph with edges from this node to each of that user's access classes. By creating such a graph, removing a user is as easy as removing his node, and thus does not require rekeying of any other user in the system.

3.5 Other Access Models

Traditionally, the standard notion of permission inheritance in access control is that permissions are transfered "up" the access graph G. In other words, any vertex in $Anc(v, G)$ has a superset of the permissions held by v. Crampton [85] suggested other access models, including:

1. Permissions that are transfered down the access graph. For these permissions, any node in $Desc(v, G)$ has a superset of the permissions held by v.

2. Permissions that are transfered either up or down the graph but only to a limited depth.

In this section, we discuss how to extend our scheme to allow such permissions. We can achieve upward and downward inheritance with only two keys per node. Also, we can achieve all of these permissions with four keys at each node for a special class of access graphs that are "layered" DAGs (defined later) when there is no collusion.

3.5.1 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 $(v, w) \in E$ there is an edge $(w, v) \in E'$. 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.

3.5.2 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, w) in the access graph it holds that if $v \in S_m$ then $w \in S_{m+1}$. 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. Clearly, with such a technique we can support access rights 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 that are lower than a specific layer L” (an analogous technique can be used for permission sets of the form “all descendants of v above some specific layer”). Suppose the key for the permission requirement to access “all ancestors of node v” is k_v 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 $F(k_v, k_L)$. Clearly, only nodes that are an ancestor of v can generate k_v 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 v AND below level L, assuming that there is no collusion.

4 IMPROVING EFFICIENCY

The main focus of this chapter is to improve key derivation time in graphs where the distance between two nodes can be large. Recall that the key derivation time in the schemes of the previous chapter require the number of operations proportional to the length of the paths between two nodes. Therefore, if the distance is large (*i.e.*, the graph is deep), we might want to reduce the distance in order to decrease computation a user must perform to access resources at a descendant class. Also, the techniques given in this chapter apply to any graphs not necessarily in the context of access graphs. Thus, for instance, we are able to utilize these techniques in our solution to time-based access control given in Chapter 5.

The main idea used here is to insert extra edges, so-called *shortcut edges*, to the hierarchy to decrease the distance between nodes. Such edges do not affect the relationship between the nodes, and thus are in the transitive closure of G. In other words, we may insert a shortcut edge between nodes v and w only if there is already a path between the nodes in the original graph G.

In this chapter, we first, in Section 4.1, present an efficient solution for hierarchies that are trees (which also extends to more general graphs), and then, in Section 4.2, provide another solution for more general graphs, which is based on the notion of dimension of a graph (defined later). Note that for trees, the first construction results in a more efficient solution, while the latter is designed to support graphs more general than trees.

4.1 A Solution for Tree Hierarchies

Throughout this section we assume that the access structure is a tree with n nodes, unless mentioned otherwise. We first describe a construction that reduces the path between any two nodes (from the worst-case $O(n)$) to $O(\log\log n)$ with $O(n)$ public space. This construction is given in Sections 4.1.1–4.1.3, where, for clarity of presentation, with start with a simple construction and gradually improve to result in the above performance. Then Section 4.1.4 describes an alternative construction that allows us to achieve the distance of 3 edges between any two nodes with public storage space (*e.g.*, the number of edges) of $O(n \log\log n)$. Finally, in Section 4.1.5 we show how the technique can be extended to more general hierarchies.

4.1.1 A Preliminary Scheme

Our solution uses the notion of a centroid of a tree: A *centroid* of an n-node tree T is a node whose removal from T leaves no connected component of size greater than $n/2$ [86]. The tree T does not need to be binary or even have constant-degree nodes. It can be shown that in any tree there are at most two centroids, and if there are two centroids, then they must be adjacent. If the tree is rooted and has two centroids, we break the tie by arbitrarily selecting the parent among the two centroids. Thus, we refer to “the” centroid of a rooted tree.

Our preliminary algorithm AddShortcuts0 for adding shortcuts edges to T is described next. It outputs a set of $O(n \log\log n)$ edges that reduce the distance between any two nodes to less than $\log n$ edges.

AddShortcuts0(T): For every node v of T, do the following:

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

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

Now we show that the shortcut edges make it possible for every ancestor v to reach any of its descendants w in a path of no more than $\log n$ edges. When we trace the path from v to w, we distinguish two cases, depending on whether w is in the subtree of the centroid c_v of T_v. The tracing algorithm which we call FindPath0 is as follows:

FindPath0(v, w, T):

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

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

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

4.1.2 Improving the Time Complexity

Before describing an improvement to the above scheme, we need to review the concept of *centroid decomposition* of a tree. If we compute the centroid of a tree, then remove it, and recursively repeat this process with the remaining trees (of size no more than $n/2$ each), we obtain a decomposition of the tree into what is called a "centroid decomposition." Such a decomposition can be easily computed in linear time (see, for example, [87]).

Our improved scheme is based on doing a pre-processing step of T that consists of performing what we call a "prematurely terminated centroid decomposition." This is similar to standard centroid decomposition, except that we stop the recursion not when the tree becomes a single node, but when the tree size becomes at most $\sqrt{n}$. This means that there are at most $\sqrt{n}$ successive centroids that are affected by this prematurely terminated decomposition (as opposed to n of them for the standard decomposition). We call these centroids, as well as the root of T, *special* nodes. Note that by our construction removing the special nodes from T leaves no connected components of size larger than $\sqrt{n}$ each; let us call these connected components (which are trees) the "residual" trees and denote them by $T_1, \ldots, T_k$.

We also use the notion of a "reduced tree.". The reduced tree $\hat{T}$ consists of the special nodes and of edges (v, w) that satisfy the following conditions: (i) v is an ancestor of w in T, and (ii) there is no other node of $\hat{T}$ on the path from v to w in T.

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

AddShortcuts1(T):

1. If $|T| \leq 4$, then return an empty set of shortcuts. Otherwise continue with the next step.

2. Compute the special nodes of T. Initialize the set of shortcuts S to be empty.

3. Create from T the reduced tree $\hat{T}$, and add to S a shortcut edge between every ancestor-descendant pair in $\hat{T}$ (unless the ancestor is a parent of the descendant, in which case there is already such an edge in T).

4. For every residual tree T_i in turn ($i = 1, \ldots, k$), add to S a shortcut edge from the root of T_i to every node in T_i that is not a child of that root.

5. For every residual tree T_i in turn ($i = 1, \ldots, k$), recursively call $\mathsf{AddShortcuts1}(T_i)$. If we let S_i denote the set of shortcuts returned by that recursive call, then update S by setting $S = S \cup S_i$. Return S.

The number of shortcut edges added in Steps 3 is $O(|T|)$ (since there are $O(\sqrt{|T|})$ special nodes) and in Step 4 is $\sum_{i=1}^{k} |T_i|$, which is $\leq |T|$. The overall recursive procedure then obeys the following recurrence, where $f(|T|)$ is the number of shortcut edges we are introducing:

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

Here every T_i has size $\leq \sqrt{n}$ and c_1 is a constant. A straightforward induction proves that this recurrence implies that $f(n) = O(n \log\log n)$, which is the size of the public information due to the creation of shortcut edges.

Next, we show that for every ancestor-descendant pair $(v,\ w)$ in T, there is now a path of length $O(\log\log n)$. Our algorithm for finding such a path mimics the recursion of $\mathsf{AddShortcuts1}$. In what follows, we use $\mathsf{Dist}(n)$ to denote the distance between any ancestor and descendant in the graph after generating shortcuts using $\mathsf{AddShortcuts1}(T)$.

$\mathsf{FindPath1}(v, w, T)$:

1. If $|T| \leq 4$, then trace a path from v to w along T and return that path. If $|T| > 4$, continue with the next step.

2. If v and w are both special in T, then return the edge (v, w). If v and/or w is not special, then proceed to the next step.

3. Let T_i be the residual tree containing v, and let T_j be the residual tree containing w. If $i = j$, then we recursively call $\mathsf{FindPath1}(v, w, T_i)$ and return the path the call returns.If $i \neq j$, then we proceed as follows:

(a) We recursively call $\mathsf{FindPath1}(v, u, T_i)$, where u is the node of T_i nearest to w in T (hence u is a leaf of T_i, and one of its children u' in T is a special node that is an ancestor of w in T). This path is the initial portion of the path $\mathcal{P}$ from v to w that will be returned by this algorithm ($\mathcal{P}$ will be further built in the steps that follow).

(b) Follow the edge in T from u to the special node u' that is ancestor of w in T, and append that edge (u, u') to $\mathcal{P}$.

(c) Follow (and append to $\mathcal{P}$) the edge in $\hat{T}$ from special node u' to the special node (call it x) that is the special ancestor of w nearest to it (*i.e.*, x is parent of the root of the residual tree T_j that contains w). (Note that such an edge exists because of Step 3 of $\mathsf{AddShortcuts1}(T)$.) If $x = w$, then return $\mathcal{P}$, otherwise continue with the next step.

(d) Follow (and append to $\mathcal{P}$) the edge in T from x to the root of T_j.

(e) Follow (and append to $\mathcal{P}$) the edge from the root of T_j to w; such an edge exists because of Step 4 of $\mathsf{AddShortcuts1}(T)$. Return $\mathcal{P}$.

Note that in the above algorithm, the path returned in Step 3 when $i = j$ has length $\leq \mathsf{Dist}(|T_i|) \leq \mathsf{Dist}(\sqrt{|T|})$ and, likewise, the path returned in Step 3(a) has length $\leq \mathsf{Dist}(|T_i|)$. Then the recurrence for Dist implied by the above algorithm is:

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

where c_i's are constants. A straightforward induction proves that this recurrence implies that $\mathsf{Dist}(n) = O(\log\log n)$, which is the worst-case key derivation time.

4.1.3 Improving the Space Complexity

In this section we further lower performance of the key assignment scheme by reducing the public information (*i.e.*, the number of edges in the graph) to $O(n)$.

Similar to the solution of the previous section, we begin with a pre-processing step that consists of performing "prematurely terminated centroid decomposition" of T, except that now we stop the recursion not when the tree becomes of size $\leq \sqrt{n}$, but when the tree size becomes $\leq \log\log n$. This means that there are at most $O(n/\log\log n)$ successive centroids that are affected by this new form of prematurely terminated decomposition, which we call *distinguished* nodes altogether with the root of T. Note that removing these distinguished nodes from T leaves no connected components of size larger than $\log\log n$ each; we call these connected components (which are trees) "tiny trees."

We also use the notion of a "reduced tree" T' that is conceptually similar to the $\hat{T}$ of the previous section: The nodes of T' are the distinguished nodes, hence there are $O(n/\log\log n)$ of them. The edges of T' satisfy the following condition: there is an edge from node v to node w in T' if and only if (i) $v \in Anc(w, T)$, and (ii) there is no other node of T' on the path from v to w in T.

Our final AddShortcuts algorithm is then as the following:

AddShortcuts(T):

1. Compute the distinguished nodes of T. Create T'.
2. Use the method of Section 4.1.2 on the tree T'. Any edge of T' that was not in T must be considered a new (*i.e.*, a shortcut) edge.

Note that this algorithm uses $O(n)$ public storage, because $|T'| = O(n/\log\log n)$. To trace a path between any ancestor and descendant nodes, we use the following algorithm:

FindPath(v, w, T):

1. If both v and w are distinguished nodes, call FindPath1(v, w, T'). Otherwise, proceed with the next step.
2. Trace a path in T from v to the nearest distinguished node (call it u) that is ancestor of w (the length of this path is at most $\log\log n$ because the tiny trees

have size $\leq \log\log n$). If there does not exist such a distinguished node u that is both a descendant of v and ancestor of w, then v and w must be in the same tiny tree. In this case we can directly follow edges of T from v to w and stop.

3. Trace a path in T' from u to the nearest distinguished node (call it x) that is an ancestor of w. If $x = w$ then stop, otherwise continue with the next step.

4. Trace a path in T from x to w (within the same tiny tree).

Since the distance between any two distinguished nodes is at most $\log\log n - \log\log\log n$, and the size of each tiny tree is at most $\log\log n$, paths produced in each of the above steps (and the total paths from v to w) have length $O(\log\log n)$.

4.1.4 A Time/Space Tradeoff

In this section we introduce solutions with constant key derivation time. The idea is that by increasing the space complexity to $O(n \log\log n)$ we can reduce the distance between any ancestor and descendant nodes to at most 3 edges. Like the scheme described in Section 4.1.2, we start with prematurely terminated centroid decomposition that stops when the tree size is $\leq \sqrt{n}$. We also use the reduced tree $\hat{T}$. The approach is as follows.

AddShortcuts(T):

1–4. The same as in the AddShortcuts1(T) algorithm of Section 4.1.2.

5. For every residual tree T_i in turn ($i = 1, \ldots, k$), add to S a shortcut edge from each node v in T_i (other than the root) to all nodes in $\hat{T}$ that are both: (i) descendants of v and (ii) children of the root of T_i in $\hat{T}$.

6. For every residual tree T_i in turn ($i = 1, \ldots, k$), recursively call AddShortcuts(T_i) and, if we let S_i be the set of shortcuts returned by that call, update S by setting $S = S \cup S_i$. Return S.

Step 5 of the above algorithm adds at most $O(|T|)$ edges to the shortcut set: all new edges that point to a single node w in $\hat{T}$ come from at most one tree (as w has at most one parent in $\hat{T}$). And since each tree has at most $O(\sqrt{|T|})$ nodes, there are at most $O(\sqrt{|T|})$ new edges pointing to w. Finally, the overall $O(|T|)$ bound comes from the fact that there are $O(\sqrt{|T|})$ nodes in $\hat{T}$.

The total number of edges added to the shortcut set in the above algorithm follows a recurrence similar to the scheme in Section 4.1.2; thus, this scheme adds $O(n \log\log n)$ edges. Furthermore, the algorithm $\mathsf{FindPath}(v, w, T)$ is very similar to the Section 4.1.2 algorithm. To avoid unnecessarily repeating the techniques, we describe only the case of FindPath that differs from its previous version. It corresponds to the situations where v and w are in different residual trees and neither of them is a special node. In this case, it takes at most one hop to get to a special node (call it u_1) that is an ancestor of w (by Step 5 of $\mathsf{AddShortcuts}(T)$). Then we can get to the special node of the residual tree containing w following a single edge (call this node u_2) by Step 3. Finally, we can reach w from u_2 following another edge by Step 4. The path from v to w is thus v, u_1, u_2, w.

The above scheme requires three edges to reach a specific node. It is trivial to show that a one-hop solution must add $O(n^2)$ edges, but a two-hop solution exists with $O(n \log n)$ public space. Section 4.2 provides more details on shortcut schemes for total orders with the distance between nodes being different small constants. Such results apply to trees as well (due to the possibility of using centroid decomposition), and we do not further describe them here.

4.1.5 Extending the Techniques to Other Graphs

In this section, we extend the shortcut techniques beyond tree hierarchies and introduce an algorithm for adding shortcut edges to general access graphs. This algorithm can be applied to any hierarchy and addition of shortcuts results in key derivation being at most $O(\log\log n)$ steps. The algorithm, however, guarantees efficient storage only for certain hierarchies, specifically, if the number of nodes with multiple parents is relatively small (for instance, $\sqrt{n}$). We believe that this limited notion of an access graph captures many real life access hierarchies, and the results of Section 4.2 can be applied to hierarchies where this is not the case.

Suppose we are given a transitively-reduced[1] access graph $G = (V, E)$. Let M denote the set of nodes in G with multiple parents. Define the set of edges E_M to be the set of edges in E that are incident on one or more nodes in M. The AddShortcuts algorithm then works as follows:

AddShortcuts(G):

1. Invoke the algorithm of Section 4.1.3 on the graph $G' = (V - M, E - E_M)$. Note that this graph is a forest of trees (since the nodes in M have been removed). Let E_1 denote the set of new edges returned by the algorithm.

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

By the correctness of the AddShortcuts algorithm of Section 4.1.3, any ancestry relation in the graph formed in Step 1 is captured by a path of length $O(\log \log n)$ and $|E_1| = O(|E - E_M|) = O(n)$. The shortcut edges added to G are $E_1 \cup E_2$, and the new graph is $G_S = (V, E \cup E_1 \cup E_2)$.

To show the correctness of this algorithm (*i.e.*, to show that if there is path from node v to node w in G_S, then there is a path from v to w in G), we must illustrate that every added edge is part of the transitive closure of G. Observe that by the correctness of the previous scheme, this is true for every edge in E_1. Furthermore, it is clearly true for edges in E_2, since the edges are added to E_2 only when the source is an ancestor of the destination.

[1] The transitive reduction of a graph G is the smallest graph $R(G)$ such that $C(G) = C(R(G))$, where $C(G)$ is the transitive closure of G.

We also need analyze the space complexity of the algorithm. We have $|E_1| = O(n)$ and $|E_2| = O(|M|^2)$. Thus, there will be at most $O(n + |M|^2)$ new edges introduced. And if $|M|$ is relatively small, *e.g.*, $O(\sqrt{n})$, then the space complexity is $O(n)$.

Now we show how the FindPath algorithm works in this case, resulting in paths of length $O(\log\log n)$ between any pair of an ancestor and descendant in G.

FindPath(v, w, G_S): There are 3 cases to consider:

1. There is no node from M on the path from v to w (including v and w themselves). Then there will be a path from v to w in $G' = (V - M, E - E_M)$, and we can use previous FindPath(v, w, G'_S) procedure to produce a path of length $O(\log\log n)$ from v to w.

2. All paths from v to w contain at least one node from M, but there is a path from v to w with a single node from M on it. Let us for now assume that neither v nor w is in M. Then there will be a path $v, \ldots, u_1, m, u_2, \ldots, w$ such that $m \in M$. In this case v can reach u_1 with $O(\log\log n)$ hops by executing the previous FindPath(v, u_1, G'_S) algorithm, u_1 can reach m following a single edge, m can reach u_2 following a single edge, and u_2 can reach w with $O(\log\log n)$ hops by executing previous FindPath(u_2, w, G'_S). The case where v or w is in M easily follows from the above.

3. All paths from v to w contain at least two nodes in M. Similar the previous case, we describe what needs to be done when neither v nor w is in M; cases when v and/or w is in M directly follow from this description. Let $v, \ldots, u_1, m_1, \ldots, m_2, u_2, \ldots, w$ be such a path where $m_1, m_2 \in M$ and no node between (v and u_1) and (u_2 and w) is in M. Note that v can reach u_1 by calling previous FindPath procedure in $O(\log\log n)$ hops, u_1 is within one edge from m_1, m_1 is within one edge from m_2 (because of the edges in E_2), m_2 is within one edge from u_2, and u_2 can reach w in $O(\log\log n)$ hops by calling FindPath of Section 5.3.

It is clear that this algorithm returns paths of length $O(\log\log n)$.

4.2 A Solution for More General Hierarchies

Our construction for more general graphs is based on the notion of the dimension of a graph, which we define shortly. For d-dimensional graphs, the basis of our solution includes a reduction to the $(d-1)$-dimensional case. Thus, in this section we provide solutions to one-dimensional case (Section 4.2.2) and then give a dimension reduction technique (Section 4.2.3).

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 the previous section 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, and we believe it has much promise beyond enabling the specific performance bounds that we achieve in this work.

2. As our techniques 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 its dimension. 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.

4.2.1 Background

An n-vertex access hierarchy G is a partial order; and 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.*, [88, 89]). 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 [90], 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 [91] 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 [91]. 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 section 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 section that supports trees.

4.2.2 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 may 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 approach of the previous section, we add shortcut edges to the graph corresponding to the user hierarchy to lower key derivation time. As before, such shortcut edges should preserve the original relationship between the nodes in the graph and thus must satisfy the constraint of (i) *reachability*: given two nodes v_i and v_j where $i < j$, there is a path from v_i to v_j; and (ii) *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.*, $\max_{v_i,v_j \in V, i<j} dist(v_i, v_j) \leq h$. In other words, an h-hop solution produces graphs of diameter 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.

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 also 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; denote this median node 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 $\mathsf{AddShortcuts}_2(G_L)$.

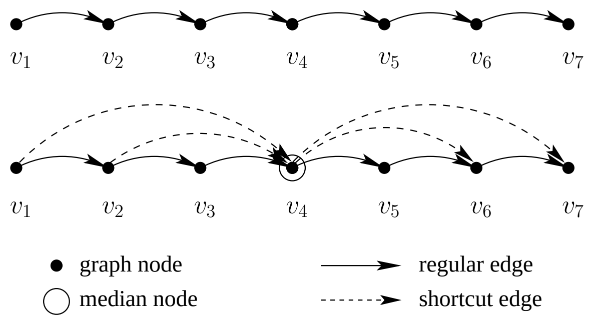


Figure 4.1. Addition of shortcut edges for the two-hop one-dimensional solution.

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

Figure 4.1 depicts the first level of recursion for the above procedure.

It can easily be shown that, in the above-defined structure, the nodes in the graph are at most two hops from each other. That is, suppose nodes v and w are separated by some median m during the above protocol. Then clearly there is a path of length two from v to w (specifically, v to m to w). On the other hand, if v and w 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 a simple 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 $\mathsf{FindPath}_2(v, w, G)$ procedure that, given two nodes v and w, finds a path consisting of two edges from point v to point w. 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 v and w are a part of that node's sub-problem: the shortcut edges (v, m) and (m, w) 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 the results of [92] 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 for our construction where RT is a complete binary tree). In our case, the two nodes whose NCA we seek are the leaves of RT that contain v and w, and their NCA is the node u that contains the two shortcut edges that we want.

Three-Hop Solutions

In this section, we describe a shortcut scheme where nodes are separated by at most three hops. In Section 4.1 we gave a scheme for trees that introduced $O(n \log \log n)$ shortcut 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 $i_1 < i_2 < \cdots < i_m$.

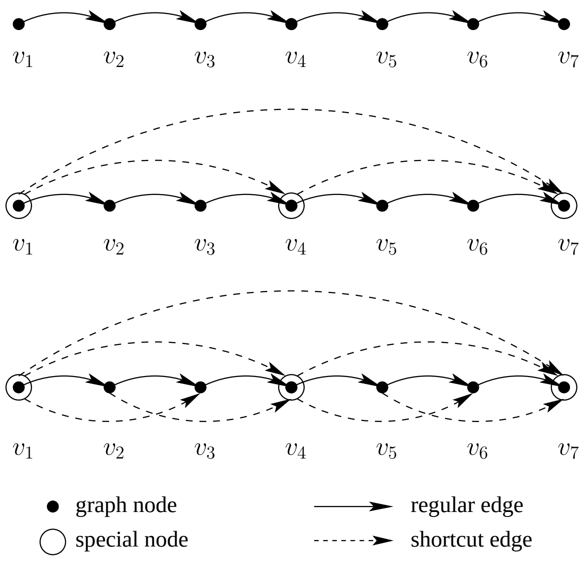


Figure 4.2. Addition of shortcut edges for the three-hop one-dimensional solution.

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 4.2 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 figure 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 v and w (where v is left of w) are separated by a special node, then v can reach its nearest special node u_1 in at most one hop (from Step 5), u_1 can reach the special node u_2 that is rightmost special node before w in at most on hop (from Step 3), and u_2 can reach w in at most one hop (from Step 4). Finally, if v and w do not fall in the base case of Step 1, they will always be separated by a special node due to the recursive nature of the algorithm.

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 previous case of the two-hop solution, the existence of a three-hop path is not enough: we also need a constant-time algorithm for finding it. The $\mathsf{FindPath}_3(v, w, G)$ procedure for doing this is very similar to the $\mathsf{FindPath}_2(v, w, G)$ 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 v and w and the nodes u_1 and u_2, such that edges (v, u_1), (u_1, u_2), (u_2, w) are in G and are available at u (from the shortcut edges added the $\mathsf{AddShortcuts}_3(G)$ procedure).

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 $\frac{n}{m}$ 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 the nodes.

4. For each non-special node $v \notin S$, add an edge from it to its nearest special node on the right.

5. For each non-special node $v \notin S$, add an edge from the nearest special node on the left (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 scheme 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 can be shown 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 u_1 and u_2 of the four-hop path x, u_1, m, u_2, y can simply be read. The node m is retrieved from the recursion tree of G_S using NCA computation. For any general h, $\mathsf{FindPath}_h(v,w,G)$ will have $O(h)$ complexity.

$O(\log^* n)$-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 above) 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) $m+1$ blocks $G_1, G_2, \ldots, G_{m+1}$ 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

Table 4.1
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.	$O(n)$

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 block is done in $\leq \log^* n$ hops by following each edge within that block. Given node v in G_i, derivation of the key of node w in G_j, $i < j$, is done by first (i) following edges in G_i from v to the vertex $u_1 \in S$ that is at the boundary of G_i and G_{i+1}; then (ii) using a 4-hop derivation within S to go from u_1 to the vertex $u_2 \in S$ that is at the boundary of G_{j-1} and G_j; and finally (iii) following edges in G_j from u_2 to w. The total number of hops in that case is therefore $\leq 2\log^* n + 4$.

Summary of One-Dimensional Solutions

Table 4.1 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 (*i.e.*, the diameter of the graph) is at most s, *i.e.*, a so-called s-Hop Scheme.

Shortcut schemes have also been considered in prior literature ([93–97]), 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.

Table 4.2
The number of edges in one-dimensional h-hop solutions.

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

Throughout the rest of this work we may use $\mathcal{S}1(n)$ to denote any shortcut scheme for graphs of dimension 1 applied to a total order of size n. We also use $space(\mathcal{S}1(n))$ and $time(\mathcal{S}1(n))$ to denote its public storage and key derivation complexity, respectively.

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 4.2.2 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 4.2.2. 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 4.2 shows the number of edges from our simulation results for schemes with the maximum of 1 to 9 hops.

4.2.3 Higher Dimensions

We give a solution for graphs of dimension d that achieves key derivation of no more than (and typically less than) $2(d-1)+h_1(n)$ steps, where $h_1(n)$ denotes the distance between two nodes in the underlying one-dimensional scheme for an n-node graph. As before, each step in key derivation corresponds to following one shortcut edge. 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 an arbitrary dimension d, for clarity of exposition, we first present the solution for $d=2$. Once the intuition and the basic construction have been presented for $d=2$, we give the general construction for arbitrary d.

The Case of Dimension 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:

AddShortcuts(G):

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 $S = S \cup S_2$ 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 4.2.2. 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 S as $S = S \cup S_3$. (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 $f(n) \leq 2f(n/2) + cf_1(n)$

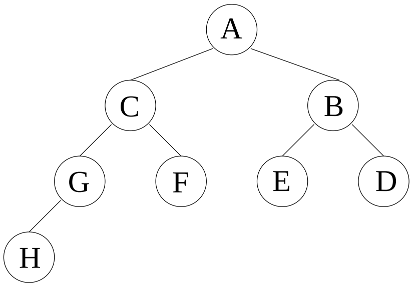


Figure 4.3. Example two-dimensional access hierarchy (original).

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.

The length of the path between any w and v can be proved to be at most $h_1(n)+2$ by induction on n (the base case is trivial). That is, if $w \in V_2$ and $v \in V_1$, then the path of length at most $h_1(n) + 2$ consists of 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.

Example. To help clarify our technique, we give an example of the recursive step of the above AddShortcuts algorithm. Figure 4.3 shows a tree access hierarchy that will be used for this example.

Figure 4.4 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 4.5 shows the shadow points (added in Step 3 of the algorithm 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.

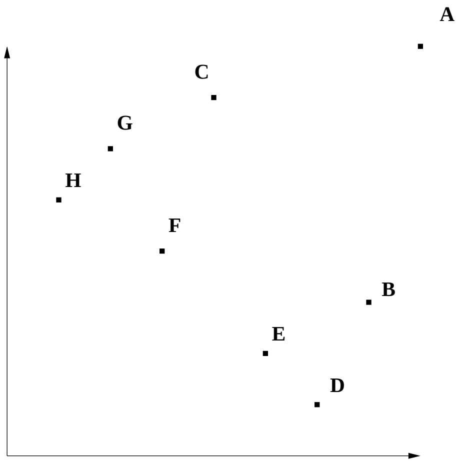


Figure 4.4. Example two-dimensional access hierarchy converted to tuple form.

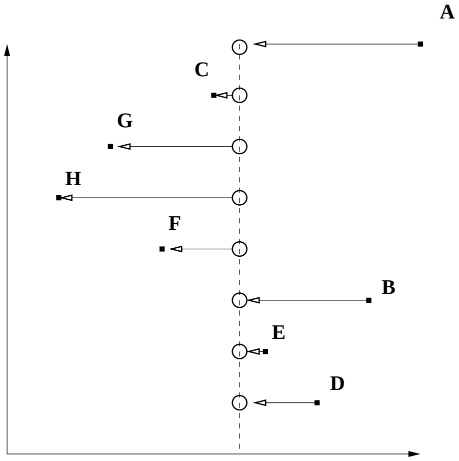


Figure 4.5. Example two-dimensional access hierarchy with shadow points.

The Case of Dimension 3 and Higher

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 was the case with the 2-dimensional solution, we use a recursive construction. Specifically, we inductively assume that the $(d-1)$-dimensional problem can be solved with addition of $O(f_1(n)(\log n)^{d-2})$ edges achieving the distance of $2(d-1)+h_1(n)$ between the nodes (note that this holds for $d=1$ and for $d=2$ by the previous subsections). The solution steps are as follows:

AddShortcuts(G):

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

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

3. If $d = 1$, then solve using any of the one-dimensional schemes; otherwise continue with the next step.

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, and 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 $S = S \cup S_1$ 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 $S = S \cup S_2$ 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 $d-1$, then 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 $f(n,2) \leq c_1 f_1(n) \log n$; if $d > 2$, then:

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

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 $d-1$ dimensions), 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:

$$f(n,d) = O(df_1(n)(\log n)^{d-1}).$$

The w-to-v number of edges obeys the following recurrence: If $n > 1$, then $h(n,2) \leq h_1(n) + 2$; if $d > 2$, then

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

The above recurrence follows from the following number of edges: one edge from V_2 to a shadow point, $h(n,d-1)$ edges on the $(d-1)$-dimensional hyperplane in Step 7, and one edge 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 4.3 summarizes performance of our solution when instantiated with different one-dimensional schemes. In the table, $h_1(n)$ and $h_d(n)$ denote the 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 4.3
Performance of the d-dimensional scheme using various 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	$O(n)$	$2(d-1)+O(\log^* n)$	$O(n(\log n)^{d-1})$

Using the Data Structure

To permit efficient key derivation, we also need a corresponding $\mathsf{FindPath}(v, w, G)$ procedure that, given two points v and w in V, finds a shortest path of shortcut edges from v to w. Here we give such a procedure, which can be viewed as a simple generalization of the path-finding procedures of the one-dimensional case.

As before, we use RT to denote the recursion tree corresponding to the algorithm that adds shortcuts to the graph; 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 augment every node u in RT with an array Π_u that, for every point v of V_u, gives its projection $\Pi_u(x)$ on the hyperplane M_u that was used in the recursive call for u. This takes the same space as 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 $v \in V$ occurs in h sets like V_u once at each depth i in RT, $1 \leq i \leq h$ (the root being at the depth of 1). In what follows, for every point $v \in V$ and $1 \leq i \leq h$, we use $N(v, i)$ to denote the node u of RT at depth i and whose V_u contains v. Note that $N(v, 1)$ is the root of RT, and that $N(v, h)$ is the leaf of RT that contains v.

Now we turn our attention to how RT is used to trace a short path between two points. As we did for the one-dimensional case, here we make use of the fact that in a tree it is easy to answer nearest common ancestor queries in constant time (*i.e.*, 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 $v, w \in V$ and, if v dominates w, returns a shortest path from v to w. 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 v dominates w. If not, then output "no path exists" and stop; otherwise continue with the next step.

2. If the dimension of (v, w, G) is 1, then use a one-dimensional procedure for finding the path. Otherwise, continue with the next step.

3. Let $x = N(v, h)$ and $y = N(w, h)$, *i.e.*, x (resp., y) is the leaf of RT whose corresponding set contains v (resp., w).

4. Compute in constant time the NCA in RT of x and y, call it u. Note that x and y are both in V_u and they are on different sides of the hyperplane M_u. Let $v' = \Pi_u(v)$ and $w' = \Pi_u(w)$. The first edge on the path we seek is (v, v'), the last edge on it is (w', w), and the portion of it from v' to w' is of dimension $d-1$ and can be computed as FindPath(v', w', G'_u).

The time taken by this 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 technique we used in this FindPath algorithm is widely applicable to other recursive solutions built by addition of shortcut edges: Its essence is that a nearest common ancestor computation 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).

5 TIME-BASED KEY ASSIGNMENT IN HIERARCHICAL SYSTEMS

5.1 Problem Description

While the motivation for this chapter comes from the need to support access control policies with temporal constraints in user hierarchies, the problem does not need to be limited to this particular setting. That is, an efficient solution to the key management problem in temporal access control can find use in other domains. Therefore, we provide a very general formulation of the problem, without any assumptions on the environment in which it is used. Of course, access control in user hierarchies remains the most immediate and important application of our techniques. Thus, in Section 5.4 we will show how our solution can be used to realize temporal access control for user hierarchies.

Now let us assume that we are given a resource, and the owner of this resource would like to control user access to that resource using time-based policies. For that purpose, the lifetime of the system is partitioned into short time intervals (normally, of a length of a day or shorter), and the access key for that resource changes every time interval. Let m denote the number of time intervals in the system, $T = \{t_1, \ldots, t_m\}$ denote the intervals, and $K = \{k_{t_1}, \ldots, k_{t_m}\}$ denote the corresponding access keys.

Now assume that a user $\mathcal{U}$ is authorized to access that resource during a contiguous set of time intervals $T_{\mathcal{U}} \subseteq T$, where $T_{\mathcal{U}} = \{t_{start}, \ldots, t_{end}\}$. Following the notation of [9], we use the *interval-set* over T, denoted by $\mathcal{P}$, which is the set of all non-empty contiguous subsequences of T, *i.e.*, $T_{\mathcal{U}} \in \mathcal{P}$ for any $T_{\mathcal{U}}$. With such access rights, $\mathcal{U}$ should receive or should be able to compute the keys $K_{T_{\mathcal{U}}} \subseteq K$, where for each $t \in T_{\mathcal{U}}$ the key $k_t \in K_{T_{\mathcal{U}}}$. We denote the private information that $\mathcal{U}$ receives by $S_{T_{\mathcal{U}}}$. Obviously, storing $|T_{\mathcal{U}}|$ keys at the user end is not always practical (especially if this number is large), and significantly more efficient solutions are possible. Then a *time-based key assignment scheme* assigns keys to the time intervals and users, so that time-based access control is enforced in a correct and efficient manner. Such key generation is assumed to be performed by a central authority, but once a user is issued the keys, there is no interaction with other entities. More formally, we define a time-based KA scheme as follows:

Definition 5.1.1 *Let T be a set of distinct time intervals and $\mathcal{P}$ be the interval-set over T. A time-based key assignment scheme consists of algorithms* (Set_T, Assign_T, Derive_T) *such that:*

Set_T *is a probabilistic algorithm, which, on input a security parameter 1^κ and the set of time intervals T, outputs (i) a key k_t for any $t \in T$; (ii) secret information* Sec *associated with the system; and (iii) public information* Pub. *Let $(K, \mathsf{Sec}, \mathsf{Pub})$ denote the output of this algorithm, where K is the set of all keys.*

Assign_T *is a deterministic algorithm, which, on input a time sequence $T_\mathcal{U} \in \mathcal{P}$ and secret information* Sec, *outputs private information $S_{T_\mathcal{U}}$ for $T_\mathcal{U}$.*

Derive_T *is a deterministic algorithm, which, on input a time sequence $T_\mathcal{U}$, time interval $t \in T_\mathcal{U}$, private information $S_{T_\mathcal{U}}$, and public information* Pub, *outputs the key k_t for time interval t.*

The correctness requirement is such that, for each time sequence $T_\mathcal{U} \in \mathcal{P}$, each time interval $t \in T_\mathcal{U}$, each private information $S_{T_\mathcal{U}}$, each key $k_t \in K$, and each public information Pub *that* $\mathsf{Set}_T(1^\kappa, T)$ *and* $\mathsf{Assign}_T(T_\mathcal{U}, \mathsf{Sec})$ *can output,* $\Pr[\mathsf{Derive}_T(T_\mathcal{U}, t, S_{T_\mathcal{U}}, \mathsf{Pub}) = k_t] = 1$.

Note that in many cases the Assign_T algorithm can be a part of the Set_T algorithm, *i.e.*, private values $S_{T_\mathcal{U}}$ for every $T_\mathcal{U} \in \mathcal{P}$ are generated at the system initialization time. We, however, separate these algorithms to account for cases where retrieving $S_{T_\mathcal{U}}$ from Sec is not straightforward (which is the case in our scheme). In such cases, merging these two algorithms together will needlessly complicate Set_T resulting in unnecessary overheads.

As in hierarchical access control, we distinguish between two different notions of security for a time-based KA scheme: security against *key recovery* and security with respect to *key indistinguishability*. A time-based KA scheme can also be secure against static or adaptive adversaries. In [9], however, it was shown that the security of a time-based hierarchical KA scheme against a static adversary is polynomial-time equivalent to the security of that scheme against an adaptive adversary for both security goals (key recovery and key indistinguishability). While in the current discussion we are not concerned with hierarchical schemes, our setting can be considered to be a special case of a hierarchy with a single class. Thus, in this work we only provide definitions of a time-based KA scheme secure against a static adversary; and a proof of security under such definitions will imply security against an adaptive adversary.

In our definition of a scheme secure against static adversary, let adversary $\mathcal{A}_{st}$ attack the security of the scheme at time $t \in T$. $\mathcal{A}_{st}$ is then allowed to corrupt all users who are not authorized to have access to k_t and, when finished, is asked to guess k_t. We consider a scheme to be secure only if $\mathcal{A}_{st}$ has at most negligible probability in outputting the correct key. More formally, we capture the adversary's corrupt queries using algorithm $\mathsf{Corrupt}_t(\cdot)$ that takes the secret information Sec as input and outputs a sequence of private information denoted by $corr$. The adversary then uses $corr$ to try to compute the key k_t.

Definition 5.1.2 *Let T be a set of distinct time intervals, $\mathcal{P}$ be the interval-set over T, and $\mathsf{KA} = (\mathsf{Set}_T, \mathsf{Assign}_T, \mathsf{Derive}_T)$ be a time-based KA scheme for $\mathcal{P}$ and a security parameter κ. Then KA is secure against key recovery in the presence of a static adversary if it satisfies the following properties:*

- Completeness: *A user, who is given private information $S_{T_{\mathcal{U}}}$ for a sequence of time intervals $T_{\mathcal{U}} \in \mathcal{P}$, is able to compute the access key k_t for each $t \in T_{\mathcal{U}}$ using only her knowledge of $S_{T_{\mathcal{U}}}$ and public information Pub with probability 1.*

```
Experiment $\mathbf{Exp}^{\mathrm{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}) \leftarrow \mathsf{Set}_T(1^\kappa, T)$
    $corr \leftarrow \mathsf{Corrupt}_t(\mathsf{Sec})$
    $k \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr)$
    if $k = k_t$ then return 1
    else return 0
```

```
Experiment $\mathbf{Exp}^{\mathrm{key-ind}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}) \leftarrow \mathsf{Set}_T(1^\kappa, T)$
    $corr \leftarrow \mathsf{Corrupt}_t(\mathsf{Sec})$
    $b \overset{R}{\leftarrow} \{0, 1\}$
    if $b = 0$ then $r \overset{R}{\leftarrow} \{0, 1\}^{|k_t|}$
                $b' \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr, r)$
    else $b' \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr, k_t)$
    if $b = b'$ then return 1
    else return 0
```

Figure 5.1. Experiments in which a static adversary attacking a time-based scheme participates.

- Soundness: *Let $\mathcal{A}_{st}$ be a static adversary who attacks the scheme* KA *at time interval $t \in T$. If we let the experiment $\mathbf{Exp}^{\mathrm{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}$ be specified as in Figure 5.1, the advantage of $\mathcal{A}_{st}$ is defined as:*

  $$\mathsf{Adv}^{\mathrm{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = \Pr[\mathbf{Exp}^{\mathrm{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = 1]$$

  We say that KA *is sound with respect to key recovery if for each $t \in T$, for all sufficiently large κ, and every positive polynomial $p(\cdot)$, $\mathsf{Adv}^{\mathrm{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) < 1/p(\kappa)$ for each polynomial-time adversary $\mathcal{A}_{st}$.*

If now we consider schemes where keys are pseudo-random, the task of an adversary is simply to distinguish between a real key and a random value instead of having to recover the key. In this case, the definition of a secure time-based KA scheme is similar to that of Definition 5.1.2, except the adversary is asked to participate in a different experiment:

Definition 5.1.3 *Let T be a set of distinct time intervals, $\mathcal{P}$ be the interval-set over T, and $\mathsf{KA} = (\mathsf{Set}_T, \mathsf{Assign}_T, \mathsf{Derive}_T)$ be a time-based KA scheme for $\mathcal{P}$ and a security parameter κ. Then* KA *is secure with respect to key indistinguishability against a static adversary if it satisfies the following properties:*

- Completeness: *A user, who is given private information $S_{T_{\mathcal{U}}}$ for a sequence of time intervals $T_{\mathcal{U}} \in \mathcal{P}$, is able to compute the access key k_t for each $t \in T_{\mathcal{U}}$ using only her knowledge of $S_{T_{\mathcal{U}}}$ and public information* Pub *with probability 1.*

- Soundness: *Let $\mathcal{A}_{st}$ be a static adversary who attacks the scheme* KA *at a time interval $t \in T$. If we let the experiment $\mathbf{Exp}^{\mathrm{key-ind}}_{\mathsf{KA},\mathcal{A}_{st}}$ be specified as in Figure 5.1, then the advantage of $\mathcal{A}_{st}$ is defined as:*

  $$\mathsf{Adv}^{\mathrm{key-ind}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = \Pr[\mathbf{Exp}^{\mathrm{key-ind}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = 1]$$

  We say that KA *is sound with respect to key indistinguishability if for each $t \in T$, for all sufficiently large κ, and every positive polynomial $p(\cdot)$, $\mathsf{Adv}^{\mathrm{key-ind}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) < 1/p(\kappa)$ for each adversary $\mathcal{A}_{st}$ that runs in polynomial time.*

In addition to the security requirements, an efficient KA scheme is evaluated by the following criteria:

- The size of the private information a user must store;

- The amount of computation necessary to generate an access key for the target time interval;

- The amount of information the service provider must maintain for public access.

5.2 Building the Initial Scheme

All of our constructions are based on the notion of key derivation in a graph, and, when we say that there is a directed edge from v to w in G, it implies that v is capable of deriving w's key using its own key. This means that, for the data structures that we build (all of which are DAGs), there will be a public and secret information associated with each node, and there will be public information corresponding to each edge.

Our preliminary data structure is rather simple and consists of two main steps: building a grid of size $m \times m$ (where m is the number of time intervals in the sys-

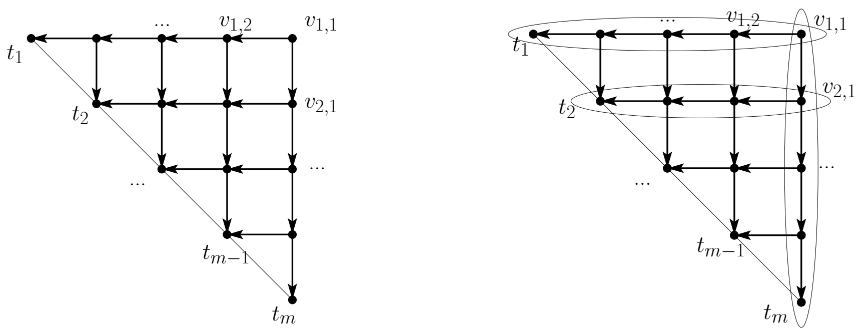


Figure 5.2. Building the data structure for the initial time-based scheme.

tem) and applying one-dimensional shortcut techniques to parts of the grid. A more detailed description follows.

$\mathsf{Set}_T(1^\kappa, T)$:

1. Build half of a grid of dimension $m \times m$ as shown in the left diagram of Figure 5.2. We denote by $v_{1,1}$ the root node; node $v_{i,j}$ is located at the row i and column j. There is a directed edge from each $v_{i,j}$ to $v_{i+1,j}$, and from each $v_{i,j}$ to $v_{i,j+1}$. Call such data structure G. The time interval t_i corresponds to the node $v_{i,m-i}$ in G.

2. Apply the hierarchical solution to G by executing $\mathsf{Set}(1^\kappa, G)$. It should be clear that in G, given a key for $v_{i,j}$, all keys for time intervals in the range $t_i, \ldots, t_{m-j+1}$ can be derived from it (in the worst-case $O(m)$ time).

3. Apply a one-dimensional shortcut scheme $\mathcal{S}1$ to each row and column of the grid (see the right diagram in Figure 5.2). In more detail, add shortcuts to the data structure to permit fast derivation of $v_{i,x}$'s key from $v_{i,y}$'s key for any $x > y$ (and similarly $v_{x,j}$'s key from $v_{y,j}$'s key for any $x > y$).

The above algorithm generates a data structure with $O(\mathit{space}(\mathcal{S}1(m)))$ shortcuts per row or column and therefore $O(m \cdot \mathit{space}(\mathcal{S}1(m)))$ total space.

Having this, the key assignment algorithm for a user entitled to have access to the resource at time intervals $T_{\mathcal{U}} = \{t_x, \ldots, t_y\} \in \mathcal{P}$ is straightforward:

Table 5.1
Performance of the initial time-based scheme.

Underlying scheme	Private storage	Key derivation	Public storage
2HS	1	≤ 4 op.	$O(m^2 \log m)$
3HS	1	≤ 6 op.	$O(m^2 \log\log m)$
4HS	1	≤ 8 op.	$O(m^2 \log^* m)$
$\log^*$HS	1	$O(\log^* m)$ op.	$O(m^2)$

$\mathsf{Assign}_T(T_{\mathcal{U}}, \mathsf{Sec})$: Return $S_{v_{x,n-y+1}}$.

Key derivation of the key corresponding to the current time interval $t_i \in T_{\mathcal{U}}$ now consists of at most $2 \cdot \mathit{time}(\mathcal{S}1(m))$ steps: at most $\mathit{time}(\mathcal{S}1(m))$ steps are needed to derive $v_{i,m-y+1}$'s key from that of $v_{x,m-y+1}$, and then at most $\mathit{time}(\mathcal{S}1(m))$ steps are needed to derive $v_{i,m-i+1}$'s key (which corresponds to t_i) from that of $v_{i,m-y+1}$. Table 5.1 summarizes the performance of this basic scheme, when used with various one-dimensional solutions.

5.3 An Improved Scheme

Next, we describe a solution that exhibits improved performance compared to the previous scheme. We start by presenting a new data structure and then provide other algorithms and usage cases to result in a full-fledged time-based KA scheme.

At a high level, to build a new data structure, we partition all time intervals into coarse blocks of $\sqrt{m}$ time intervals each. We then apply the initial scheme to these blocks treating each of them as a single unit. If user access is to be granted to a large sequence of time intervals that spans across boundaries of these blocks, we can use this level of granularity to assign keys. If, on the other hand, the user sequence of time intervals is contained within a block, we recursively apply this procedure to the time intervals within each block to support time-based access control of finer granularity.

If the user sequence spans across different blocks, but contains partial blocks at the beginning and at the end of it, then we utilize the coarse blocks' keys to cover the whole blocks along with two new types of keys that will be introduced later.

5.3.1 Lowering the Size of the Data Structure

We now present our improved data structure. For the purposes of the current presentation, we let $m = 2^{2^q}$ for some integer q. This allows us to avoid using rounding notation $\lfloor x \rfloor$ and $\lceil x \rceil$ throughout the algorithms and results in a cleaner presentation, but does not limit the applicability of our result. Our procedure for building the data structure takes as inputs a node u and the set T, and then recursively builds a tree for the set rooted at u. Due to the recursive nature of this function, we use $T^w = \{t_a, \ldots, t_b\}$ to denote the working set of the current function invocation and $|T^w|$ to denote the size of T^w. Then the data structure is constructed as follows:

DataStructBuild(u, T^w):

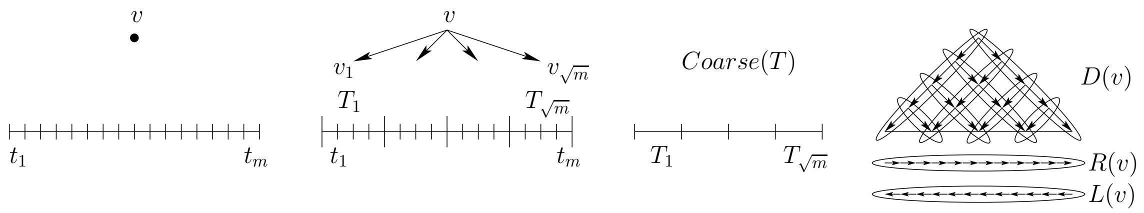
1. If $|T^w| = 2$ (*i.e.*, $q = 0$), then return. Otherwise, continue with the next step.

2. Partition T^w into $\sqrt{|T^w|}$ sets of $\sqrt{|T^w|}$ contiguous time intervals each and call them T^w_1, …, $T^w_{\sqrt{|T^w|}}$. That is, if $T^w = \{t_1, \ldots, t_{|T^w|}\}$, then $T^w_i = \{t_{i\sqrt{|T^w|}+1}$, …, $t_{i\sqrt{|T^w|}+\sqrt{|T^w|}}\}$. Create a node u_i for each T^w_i, and make u_i a child of u.

3. Generate a problem $Coarse(T^w)$, derived from T^w by treating each T^w_i as a single unit (*i.e.*, "merging" the internals of T^w_i into a single item). Note that the size of set $Coarse(T^w)$ is $\sqrt{|T^w|}$.

4. Store at node u an instance of the initial data structure for $Coarse(T^w)$, denoted by $D(u)$. $D(v)$ can process time sequences that are the union of a contiguous subset of blocks in $Coarse(T^w)$, but not time sequences with at least one endpoint inside the T^w_i's.

5. Build a one-dimensional structure by creating an edge from t_i to t_{i+1} for $i = a, \ldots, b-1$ and the reverse of this structure by creating an edge from t_{i+1} to t_i.



(a) Initial state. (b) State after Step 2. (c) State after Step 3. (d) State after Step 5.

Figure 5.3. Construction of the data structure for the improved time-based scheme (first level of recursion).

Apply the shortcutting technique to the first structure and the call the resulting graph $R(u)$. Similarly, apply the shortcutting technique to the second structure and call the resulting graph $L(u)$. The former will be used to support time sequences that start at the right boundary of T^w and end inside it (we call this the *right-anchored* problem); and the latter will be used to support time sequences that start at the left boundary of T^w and end inside it (we call this the *left-anchored* problem).

6. Recursively apply the algorithm to each child of T^w; *i.e.*, call $\mathsf{DataStructBuild}(u_i, T_i^w)$ in turn for each $i = 1, \ldots, \sqrt{|T^w|}$. Return the data structure rooted at u.

Figure 5.3 gives an illustration of how the data structure is built. $D(u)$ supports performance of 1 key, $O(time(\mathcal{S}1(|T^w|)))$ key derivation, and $O(space(\mathcal{S}1(|T^w|)))$ space. Performance of $R(u)$ and $L(u)$ is 1 key, $O(time(\mathcal{S}1(|T^w|)))$ steps per key derivation, and $O(space(\mathcal{S}1(|T^w|)))$ total space. The total space $S(m)$ of the above data structure satisfies the recurrence $S(m) \leq \sqrt{m}S(\sqrt{m}) + c_1 \cdot space(\mathcal{S}1(m))$ if $m > 2$ and $S(2) = c_2$, where c_1 and c_2 are constants. Thus, $S(m) = O(space(\mathcal{S}1(m)) \log\log m)$.

Now we can use the algorithm for building the data structure to setup a time-based KA scheme:

$\mathsf{Set}_T(1^\kappa, T)$:

1. Execute $\mathsf{DataStructBuild}(root, T)$; call the data structure returned G.

2. Execute $\mathsf{Set}(1^\kappa, G)$ using a hierarchical key assignment scheme.

5.3.2 Key Assignment

We now turn our attention to showing which keys are given to a user who has access to an arbitrary sequence of time intervals $T_{\mathcal{U}} \in \mathcal{P}$. In what follows, u is a node of the above tree data structure and T^w is the set of time intervals associated with u. The recursive procedure below, when invoked on any $T_{\mathcal{U}}$ and our data structure, returns a set of (at most 3) keys associated with $T_{\mathcal{U}}$.

$\mathsf{AssignKeys}(T_{\mathcal{U}}, G, u, T^w)$:

1. If u is a leaf, then return a key for each of the (at most two) time intervals in $T_{\mathcal{U}}$. Otherwise, continue with the next step.

2. Let $u_1, \ldots, u_{\sqrt{|T^w|}}$ be the children of u, and let $T_1^w, \ldots, T^w_{\sqrt{|T^w|}}$ be the respective sets of times associated with these children. We distinguish two cases:

   (a) $T_{\mathcal{U}}$ overlaps with only one set T_i^w. Then we return the keys from the recursive call $\mathsf{AssignKeys}(T_{\mathcal{U}}, G, u_i, T_i^w)$.

   (b) $T_{\mathcal{U}}$ overlaps with all of $T_k^w, \ldots, T_{k+\ell}^w$, where $\ell \geq 1$. These $\ell + 1$ intervals are handled in 3 different ways: Those completely contained in $T_{\mathcal{U}}$ are collectively processed using the $D(u)$ structure, resulting in one key. If T_k^w overlaps with $T_{\mathcal{U}}$, but is not contained in $T_{\mathcal{U}}$, then it is right-anchored and is processed using $R(u_k)$, resulting in one key. If $T_{k+\ell}^w$ overlaps with $T_{\mathcal{U}}$, but is not contained in $T_{\mathcal{U}}$, then it is left-anchored and is processed using $L(u_{k+\ell})$, resulting in one key. Those (at most) 3 keys are returned.

Then the overall procedure for assigning a key to user simply calls the above recursive algorithm:

$\mathsf{Assign}_T(T_{\mathcal{U}})$: Return $\mathsf{AssignKeys}(T_{\mathcal{U}}, G, root, T)$.

All keys given to users must be labeled with the level at which they were retrieved in the data structure, *i.e.*, the distance from the root node in the tree T. This is necessary for achieving constant-time computation of access keys, which will be explained in the next section. To make key derivation simpler, we also label user keys with their type; namely: D, R, or L. In addition, if a user receives more than a single key for her time sequence $T_{\mathcal{U}}$, each key is labeled with a range of time intervals to which it permits access.

To summarize, we assume that a key given to a user will be labeled with four values $(lev, type, t_a, t_b)$, where $0 \leq lev \leq \log\log n$, $type \in \{R, L, D\}$, and $t_a, t_b \in T$ such that $t_a < t_b$. For example, if a user with access rights to $T_{\mathcal{U}} = \{t_{start}, \ldots, t_{end}\}$ is given private information consisting of three keys $S_{T_{\mathcal{U}}} = \{k_1, k_2, k_3\}$, then k_1 could be labeled with (l, R, t_{start}, t_a), k_2 with $(l-1, D, t_{a+1}, t_b)$, and k_3 with (l, L, t_{b+1}, t_{end}).

5.3.3 Content Distribution

At time $t \in T$, the service provider wants to make certain content (possibly very voluminous) available to the users with access rights to time interval t. To do so, the content is encrypted with the access key k_t using a symmetric encryption scheme and is made available to all users in the encrypted form (by placing it in a public location, broadcasting it to the users, or by other means). In our scheme the server also needs to ensure that the keys that users derive for t allow them to obtain k_t. There are $O(\log\log n)$ such keys for t in the data structure, access to which should allow access to k_t. Since the data structure has $(\log\log n + 1)$ levels, such keys are:

- Keys from $R(v)$, for some v in the data structure T, one from each level.
- Keys from $L(v)$, similarly, for a single v per level.
- Keys corresponding to $D(v)$, one from each level l, where $0 \leq l \leq \log\log n - 1$.

We refer to these keys as *enabling keys*. The server places in the public domain information that permits derivation of k_t from any of the enabling keys above (by computing a public information corresponding to an edge between an enabling key and k_t). Additionally, the server labels the public derivation information associated with each of the enabling keys with the level and the type (*i.e.*, R, L, or D) of the corresponding enabling key. This is needed to permit fast constant-time derivation of the access key k_t.

5.3.4 Key Derivation

A user $\mathcal{U}$ with access to the sequence of time intervals $T_{\mathcal{U}} = \{t_{start}, \ldots, t_{end}\} \in \mathcal{P}$ receives private information $S_{T_{\mathcal{U}}}$ consisting of 1, 2, or 3 keys that permit her to derive enabling keys for each $t \in T_{\mathcal{U}}$. In the most general (and common) case, such private information consists of 3 keys (denoted by k_1, k_2, and k_3), which are labeled as (l, R, t_{start}, t_a), $(l-1, D, t_{a+1}, t_b)$, and (l, L, t_{b+1}, t_{end}), respectively, for some l, a, and b. Let us assume, without loss of generality, that if the number of keys is less than 3, then the missing keys are set to empty strings with key k_1 remaining to be of type R, key k_2 of type D, and key k_3 of type L. Then to obtain the enabling key for a time interval $t_i \in T_{\mathcal{U}}$, $\mathcal{U}$ executes a derivation algorithm which we sketch here:

$\mathsf{DeriveKey}(T_{\mathcal{U}}, t_i, S_{T_{\mathcal{U}}}, \mathsf{Pub})$:

1. Parse $S_{T_{\mathcal{U}}}$ as $k_1(l,\ R,\ t_{start},\ t_a)$, $k_2(l-1,\ D,\ t_{a+1},\ t_b)$, $k_3(l, L, t_{b+1}, t_{end})$.

2. If $t_i \in \{t_{start}, \ldots, t_a\}$, find the node v at level l such that $R(v)$ permits access to t_i (note that such node v can be computed in constant time using index i of the time interval t_i). Use k_1 and the public information from Pub to derive the key corresponding to t_i and return that enabling key.

3. Similarly, if $t_i \in \{t_{b+1}, \ldots, t_{end}\}$, locate the node v at level l such that $L(v)$ permits access to t_i. Use k_3 and Pub to derive an enabling key for t_i and return that key.

4. Finally, if $t_i \in \{t_{a+1}, t_b\}$, locate v at level $l-1$ such that $D(v)$ permits access to t_i; use k_2 and Pub to derive an enabling key for t_i and return it.

Key derivation complexity in all of the above cases is $O(time(\mathcal{S}1(n)))$.

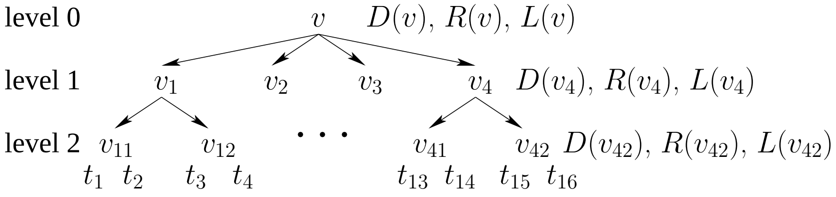

level 0
v $D(v)$, $R(v)$, $L(v)$
level 1
v_1
v_2
v_3
v_4 $D(v_4)$, $R(v_4)$, $L(v_4)$
level 2
v_{11}
v_{12}
v_{41}
v_{42} $D(v_{42})$, $R(v_{42})$, $L(v_{42})$
t_1 t_2 t_3 t_4 t_{13} t_{14} t_{15} t_{16}

Figure 5.4. Example illustration of the temporal data structure.

5.3.5 Example

To better illustrate how the above algorithms for building the data structure and assigning and deriving keys work, we give a toy example. Let $n = 16$. Then the DataStructBuild$(root, T)$ procedure will result in a tree of depth three. Let us denote the root of the tree by v, ith child of the root by v_i, and jth child of node v_i by v_{ij}. Also, let T_i and T_{ij} denote the set of time intervals that v_i and v_{ij} cover, respectively. For $n = 16$, such a tree is given in Figure 5.4. In the figure, each node w has data structures $D(w)$, $R(w)$, and $L(w)$ associated with it, which we omit for conciseness.

Now consider users $\mathcal{U}_1, \mathcal{U}_2$, and $\mathcal{U}_3$ with the following access rights: $T_{\mathcal{U}_1} = \{t_1, \ldots, t_6\}$, $T_{\mathcal{U}_2} = \{t_2, \ldots, t_4\}$, and $T_{\mathcal{U}_3} = \{t_4, \ldots, t_{14}\}$. According to the key assignment algorithm AssignKeys, they are assigned keys in the following way: Since $\mathcal{U}_1$'s sequence of time intervals starts at the beginning of the system's lifetime, $\mathcal{U}_1$'s credentials are left-anchored at the level of v, and $\mathcal{U}_1$ obtains a single key from $L(v)$ corresponding to t_6. Such a key permits derivation of the remaining enabling keys for t_1 through t_5. User $\mathcal{U}_2$'s access rights are contained within the time interval covered by v_1. Thus, $\mathcal{U}_2$ obtains from $D(v_1)$ a key corresponding to T_{12} (covers t_3 and t_4). The remaining part of $T_{\mathcal{U}_2}$ is obtained from $R(v_{11})$ (covers t_2). Finally, for user $\mathcal{U}_3$, the access rights cross the boundaries of nodes at the first level, so we start at v. $\mathcal{U}_3$ obtains from $D(v)$ a key that permits generation of keys for T_2 and T_3 (their parent) and thus covers t_5 through t_{12}. To cover the remaining intervals, $\mathcal{U}_3$ is given the key corresponding to t_4 from $R(v_1)$ and a key from $L(v_4)$ corresponding to t_{14} (which permits derivation of the key for t_{13} as well).

Table 5.2
Performance of the improved time-based scheme.

Underlying scheme	Private storage	Key derivation	Public storage
2HS	≤ 3	≤ 5 op.	$O(n \log n \log \log n)$
3HS	≤ 3	≤ 7 op.	$O(n(\log \log n)^2)$
4HS	≤ 3	≤ 9 op.	$O(n \log^* n \log \log n)$
$\log^*$HS	≤ 3	$O(\log^* n)$ op.	$O(n \log \log n)$

To illustrate content distribution and key derivation, let t_4 be the current time interval. Our data structure contains 8 enabling keys for t_4 of level-type $(0, R)$, $(1, R)$, $(2, R)$, $(0, L)$, $(1, L)$, $(2, L)$, $(0, D)$, and $(1, D)$. The service provider places in the public domain derivation information that, given any of the keys above, permits computation of the access key k_{t_4}. $\mathcal{U}_1$ then uses its only key and $L(v)$ to derive the enabling key for t_4 and derives k_{t_4} by using public information marked with $(0, L)$. $\mathcal{U}_2$ uses its key for T_{12} compute its enabling key and obtain k_{t_4} using public information marked with $(1, D)$. Finally, $\mathcal{U}_3$ uses its key for t_4 and public information with label $(1, R)$ to obtain k_{t_4}.

5.3.6 Putting Everything Together

In this section we summarize our construction and show its performance. Figure 5.5 gives a complete description of our time-based KA scheme using the basic scheme of Chapter 3 as the key derivation mechanism. Table 5.2 summarizes performance of our solution. The security of our solution comes from the way key assignment and derivation are performed in a DAG and is not due to the details of the data structures built.

```
Algorithm $\mathsf{Set}_T(1^\kappa, T)$:
  1. Create a root node $root$ for the data structure and run $\mathsf{DataStuctBuild}(root, T)$. Let $G = (V,\ E)$ denote the tree structure returned.
  2. For each $v \in V$, randomly choose a secret key $k_w \in \{0,1\}^\kappa$ and a unique public label $\ell_w \in \{0,1\}^\kappa$ associated with each node $w$ in $D(v)$, $R(v)$, and $L(v)$.
  3. For each $v \in V$, construct public information about each edge in $D(v)$, $R(v)$, and $L(v)$ using the key derivation method. That is, for each edge $(w, u)$, its public value is $y_{w,u} \in \{0,1\}^\kappa$.
  4. For each $t \in T$, randomly choose a secret key $k_t \in \{0,1\}^\kappa$ and a unique public label $\ell_t \in \{0,1\}^\kappa$.
  5. For each $t \in T$, let $V_t \subset V$ denote the set of nodes in $G$ access to which implies access to $t$. Then for each $V_t$, for each $v \in V_t$:
     (a) find in $D(v)$ the node corresponding to the time interval $t$; call it $w$.
     (b) create an edge from $w$ to $t$ by computing public information using enabling key $k_w$, $t$'s secret key $k_t$, public label $\ell_t$, and the key derivation method. Mark such an edge with the level of $v$ and type $D$.
     (c) repeat (a) and (b) for $R(v)$ and $L(v)$, using types $R$ and $L$, respectively.
  6. Let $K$ consist of the secret keys $k_t$ for each $t \in T$ and $\mathsf{Sec}$ consist of the remaining secret keys $k_w$. Also let $\mathsf{Pub}$ consist of $G$, all public labels (of the form $\ell_w$ and $\ell_t$), and public information about all edges generated above.
Algorithm $\mathsf{Assign}_T(T_\mathcal{U}, \mathsf{Sec})$:
  1. Execute $\mathsf{AssignKeys}(T_\mathcal{U}, root, T)$, where $root$ is the root node of $G$.
  2. Set $S_{T_\mathcal{U}}$ to the keys computed and return $S_{T_\mathcal{U}}$.
Algorithm $\mathsf{Derive}_T(T_\mathcal{U}, t, S_{T_\mathcal{U}}, \mathsf{Pub})$:
  1. If $t \notin T_\mathcal{U}$, return a special rejection symbol $\perp$.
  2. Execute $\mathsf{DeriveKey}(T_\mathcal{U}, t, S_{T_\mathcal{U}}, \mathsf{Pub})$ to compute an enabling key for $t$; call it $k'_t$.
  3. Use $k'_t$ along with its (level-type) label and $\mathsf{Pub}$ to derive key $k_t$.
```

Figure 5.5. Description of proposed time-based key assignment scheme.

Theorem 5.3.1 *Assuming the security of the family of PRFs F^κ, the time-based key assignment scheme given in Figure 5.5 is both complete and sound with respect to key recovery in the presence of a static adversary.*

Proof Our proof uses a standard hybrid argument. Per Definition 5.1.2, we are dealing with adversary $\mathcal{A}_{st}$ who participates in the experiment $\mathbf{Exp}^{\text{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}$ for time interval $t \in T$. We construct a sequence of experiments $\mathbf{Exp}^{0}_{\mathsf{KA},\mathcal{A}_{st}}, \ldots, \mathbf{Exp}^{q}_{\mathsf{KA},\mathcal{A}_{st}}$, in which we modify the way the scheme is constructed while ensuring that the distributions of $\mathcal{A}_{st}$'s views remain indistinguishable in any two consecutive experiments. Our modification consists of replacing, in the public data structure corresponding to KA, one (pseudo-random) output produced by the function F^κ with a random sequence. Formally, $\mathbf{Exp}^{i}_{\mathsf{KA},\mathcal{A}_{st}}$ for any $i = 0, \ldots, q$ is:

```
Experiment $\mathbf{Exp}^{i}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}') \leftarrow \mathsf{Set}^i_T(1^\kappa, T)$
    $corr \leftarrow \mathsf{Corrupt}_t(\mathsf{Sec})$
    $k \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}', corr)$
    if $k = k_t$ then return 1
    else return 0
```

Here the algorithm Set^0_T corresponds to the original algorithm Set_T, while Set^{i+1}_T is constructed from Set^i_T by replacing one edge in the data structure with a random string. The edges that we replace are those that were constructed using k_t or any other key material that can lead to derivation of k_t. More precisely, for each level l in the data structure G, there is a unique $v \in G$ that covers t. For each such v, we replace the edges:

1. In $D(v)$, let w denote the leaf node that covers t. Then replace each edge on the path between any two nodes in $Anc(w, G)$ and replace each outgoing edge from every node in $Anc(w, G)$.

2. In $R(v)$ and $L(v)$, replace each edge on the path from the root to the node corresponding to t (call it w) and the edge from w.

The edges are replaced in the top-down fashion to completely exclude from the data structure information about each key on the way from the root to the node corresponding to time interval t.

Additionally, we replace edges from each of the $O(\log\log n)$ enabling keys, which correspond to t in G, to k_t. Thus, $\mathbf{Exp}^{0}_{\mathsf{KA},\mathcal{A}_{st}}$ corresponds to the case where $\mathcal{A}_{st}$ operates on the data structure of experiment $\mathbf{Exp}^{\text{key-rec}}_{\mathsf{KA},\mathcal{A}_{st}}$, while $\mathbf{Exp}^{q}_{\mathsf{KA},\mathcal{A}_{st}}$ corresponds to the case where $\mathcal{A}_{st}$ operates on the data structure with no information related to k_t. Since all of the keys (including k_t) are chosen at random, $\mathcal{A}_{st}$ has at most negligible probability in succeeding in $\mathbf{Exp}^{q}_{\mathsf{KA},\mathcal{A}_{st}}$. The total number of edges replaced is $O(\mathit{space}(\mathcal{S}1(n)\log\log n))$ and thus is polynomial in the security parameter κ.

Using a standard reduction argument, we can show that any non-negligible difference in behavior between experiments $\mathbf{Exp}^{i}_{\mathsf{KA},\mathcal{A}_{st}}$ and $\mathbf{Exp}^{i+1}_{\mathsf{KA},\mathcal{A}_{st}}$ can be used to construct an algorithm that $\mathcal{B}_F$ is able to break the pseudo-random function F with non-negligible advantage. Thus, we conclude that $\mathcal{A}_{st}$ has at most negligible advantage in breaking the security of the scheme. □

To achieve a stronger notion of key indistinguishability, all is needed is to use a different key assignment and derivation method that achieves this property, *i.e.*, the extended scheme of Chapter 3. Then we use this enhanced key derivation mechanism in Step 3 of the Set_T algorithm of Figure 5.5 (*i.e.*, in data structures $D(v)$, $R(v)$, and $L(v)$). This means that now someone with access to a certain key in, for instance, $R(v)$ and who guesses an unauthorized key correctly, cannot use the public information for that data structure to test the key. This change implies the corresponding change in the Derive algorithm.

Theorem 5.3.2 *Assuming the security of the family of PRFs F^κ and the security of the encryption scheme $\mathcal{E}$, our time-based key assignment scheme that uses the extended key assignment scheme of Section 3.3 is both complete and sound with respect to key indistinguishability in the presence of a static adversary.*

Proof The proof is straightforward using the hybrid argument and the proof techniques of Theorems 3.3.1 and 5.3.1. □

In Section 5.7 we show how the lifetime of the system can be extended to new intervals beyond the original m. Also, in the same section we show how one can further decrease public storage space at a slight increase in the number of user keys (*i.e.*, a generalization in terms of keys/space tradeoff).

5.4 Temporal Access Control for a User Hierarchy

Recall that in systems with hierarchically organized access classes, such a hierarchy is normally modeled as a directed acyclic access graph which we denote by G_U. As before, we assume that each node corresponds to an access class and the edges form a partial order relationship between the classes, where an edge from node v to node w means that the parent node v inherits privileges of the node w. Thus, we can assign each class a single secret key and let users obtain keys of their descendant classes through a key derivation process described in Chapter 3.

Now if we equip the model with time-based policies, in addition to computing keys of descendant classes, a user should be able to compute keys based on time. That is, a user $\mathcal{U}$ entitled to access class $v \in V_U$ during a sequence of time intervals $T_{\mathcal{U}} \in \mathcal{P}$ obtains private information that permits her to compute keys $k_{v,t}$ for her access class v and each $t \in T_{\mathcal{U}}$ (time-based key derivation). In addition, the private information allows $\mathcal{U}$ to compute, for each $t \in T_{\mathcal{U}}$, keys $k_{w,t}$ for each descendant access class w in the user hierarchy (class-based key derivation). Thus, key derivation now consists of two dimensions, which can potentially be performed using drastically different techniques.

A definition of a hierarchical time-based KA scheme can be constructed by extending Definition 5.1.1 with user hierarchies.

Definition 5.4.1 *Let T be a set of distinct time intervals, $\mathcal{P}$ be the interval-set over T, and $G_U = (V_U, E_U)$ be a DAG corresponding to a hierarchy of classes. A hierarchi-*

cal time-based key assignment scheme consists of algorithms $(\mathsf{Set}_T, \mathsf{Assign}_T, \mathsf{Derive}_T)$ *such that:*

Set_T *is a probabilistic algorithm, which, on input a security parameter* 1^κ*, the set of time intervals* T*, and the hierarchy* G_U*, outputs (i) a key* $k_{v,t}$ *for any* $v \in V_U$ *and* $t \in T$*; (ii) secret information* Sec *associated with the system; and (iii) public information* Pub*. Let the output of this algorithm be denoted by* $(K, \mathsf{Sec}, \mathsf{Pub})$*, where* K *is the set of all keys.*

Assign_T *is a deterministic algorithm, which, on input a time sequence* $T_\mathcal{U} \in \mathcal{P}$*, class* $v \in V_U$*, and secret information* Sec*, outputs private information* $S_{v,T_\mathcal{U}}$*.*

Derive_T *is a deterministic algorithm, which, on input a time sequence* $T_\mathcal{U}$*, time interval* $t \in T_\mathcal{U}$*, access class* $v \in V_U$*, private information* $S_{v,T_\mathcal{U}}$*, target class* $u \in V_U$*, and public information* Pub*, outputs the key* $k_{u,t}$ *corresponding to class* u *at time interval* t*.*

The correctness requirement is such that, for each class $v \in V_U$*, each target class* $u \in Desc(v, G_U)$*, each time sequence* $T_\mathcal{U} \in \mathcal{P}$*, each time interval* $t \in T_\mathcal{U}$*, each private information* $S_{v,T_\mathcal{U}}$*, each key* $k_{u,t} \in K$*, and each public information* Pub *that* $\mathsf{Set}_T(1^\kappa, T, G_U)$ *and* $\mathsf{Assign}_T(T_\mathcal{U}, v, \mathsf{Sec})$ *can output,* $\mathsf{Derive}_T(T_\mathcal{U}, t, v, S_{v,T_\mathcal{U}}, u, \mathsf{Pub}) = k_{u,t}$*.*

The definition of a secure time-based KA scheme must also be slightly modified for this setting to take into account different access classes. Recall from Section 5.1 that we do need to consider active adversaries, and now have a static adversary $\mathcal{A}_{st}$ who attacks a class $v \in V_U$ at time $t \in T$. $\mathcal{A}_{st}$ is allowed to obtain access to the secret information of all classes $w \in V_U$ at all times $t' \in T$, except classes $Anc(v)$ at time t. This is modeled by an algorithm $\mathsf{Corrupt}_{v,t}$, which now is class-based. We say that a hierarchical key assignment scheme is secure if such $\mathcal{A}_{st}$ has at most negligible probability of guessing $k_{v,t}$ correctly (distinguishing it from a random string) in case of security against key recovery (resp, in case of security with respect to key indistinguishability). The rest

```
Experiment $\mathbf{Exp}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}) \leftarrow \mathsf{Set}_T(1^\kappa, G_U, T)$
    $corr \leftarrow \mathsf{Corrupt}_{v,t}(\mathsf{Sec})$
    $k \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr)$
    if $k = k_{v,t}$ then return 1
    else return 0
```

```
Experiment $\mathbf{Exp}^{\mathrm{key-ind-h}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}) \leftarrow \mathsf{Set}_T(1^\kappa, G_U, T)$
    $corr \leftarrow \mathsf{Corrupt}_{v,t}(\mathsf{Sec})$
    $b \overset{R}{\leftarrow} \{0, 1\}$
    if $b = 0$ then $r \overset{R}{\leftarrow} \{0, 1\}^{|k_{v,t}|}$
                    $b' \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr, r)$
    else $b' \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}, corr, k_{v,t})$
    if $b = b'$ then return 1
    else return 0
```

Figure 5.6. Experiments in which a static adversary attacking a hierarchical time-based scheme participates.

of the security definitions for key recovery and key indistinguishability mimic our previous definitions without a hierarchy of classes.

Definition 5.4.2 *Let $G_U = (V_U, E_U)$ be a DAG corresponding to a hierarchy, T be a set of distinct time intervals, $\mathcal{P}$ be the interval-set over T, and $\mathsf{KA} = (\mathsf{Set}_T, \mathsf{Assign}_T, \mathsf{Derive}_T)$ be a time-based hierarchical KA scheme for G_U, $\mathcal{P}$, and a security parameter κ. Then KA is secure against key recovery in the presence of a static adversary if it satisfies the following properties:*

- Completeness: *A user, who is given private information $S_{v,T_\mathcal{U}}$ for a sequence of time intervals $T_\mathcal{U} \in \mathcal{P}$ and a class $v \in V_U$, is able to compute with probability 1 the access key $k_{w,t}$ for each $t \in T_\mathcal{U}$ and $w \in Desc(v, G_U)$ using only her knowledge of $S_{v,T_\mathcal{U}}$ and public information Pub.*

- Soundness: *Let $\mathcal{A}_{st}$ be a static adversary who attacks the class v at time interval $t \in T$. If we let the experiment $\mathbf{Exp}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}$ be specified as in Figure 5.6, the advantage of $\mathcal{A}_{st}$ is defined as:*

$$\mathsf{Adv}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = \Pr[\mathbf{Exp}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) = 1]$$

We say that KA *is sound with respect to key recovery if for each* $t \in T$*, for each* $v \in V$*, for all sufficiently large* κ*, and every positive polynomial* $p(\cdot)$*,* $\mathsf{Adv}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa) < 1/p(\kappa)$ *for each adversary* $\mathcal{A}_{st}$ *that runs in polynomial time.*

The definition of a hierarchical time-based key assignment scheme secure with respect to key indistinguishability is the same as the definition above with the exception that the adversary now participates in the experiment $\mathbf{Exp}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}$ given in Figure 5.6.

We can create a hierarchical time-based KA scheme by applying our solution independently to each access class in the user hierarchy. Then for each $t \in T$, the nodes with keys $k_{v,t}$ for each $v \in V_U$ are connected with edges to form the original hierarchy of classes. In more detail, for each $v \in V_U$ we use the improved scheme to build the data structure for T and generate access keys $k_{v,t}$ for every $t \in T$. This will result in $|V_U|$ instances of the time-based graph G, each of which permits key derivation for a specific access class. Since the structure of such graphs is the same for all of them, but the keys assigned to nodes and keys encoded in the public information will differ, we denote the public information generated for access class v according to G as Pub^G_v. Then for any $t \in T$, the public information for G_U is constructed according to the current keys for each access class using the key derivation method of the base scheme in Chapter 3. We denote the public information at time interval t generated according to G_U by $\mathsf{Pub}^{G_U}_t$. For a user with access privileges for time interval $T_\mathcal{U} \in \mathcal{P}$ at access level $v \in V_U$ consists of time-based key derivation (using Pub^G_v) of the key $k_{v,t}$ followed by class-based key derivation of the key $k_{w,t}$ (using $\mathsf{Pub}^{G_U}_t$); this is assuming that $t \in T_\mathcal{U}$ and $w \in Desc(v, G_U)$. A more precise description of our scheme is given in Figures 5.7 and 5.8.

In the Set_T algorithm, we first build the data structure G and generate public labels for the time intervals (Steps 1–3). Then for each class u in the user hierarchy, we select secret keys for its copy of G and generate public information according to those keys (Step 5). Next, we connect the data structures corresponding to different user classes according to the partial order relationship between those classes (Step 6). That is, for each time interval t, if user class u_1 is a parent of user class u_2, we

```
Algorithm $\mathsf{Set}_T(1^\kappa, T, G_U)$:
  1. Create a root node $root$ for the data structure and run $\mathsf{DataStuctBuild}(root, T)$. Let $G = (V, E)$ denote the tree structure returned.
  2. For each $v \in V$, choose a unique public label $\ell_w \in \{0,1\}^\kappa$ for every node $w$ in $D(v)$, $R(v)$, and $L(v)$.
  3. For each $t \in T$, choose a unique public label $\ell_t \in \{0,1\}^\kappa$.
  4. For each $u \in V_U$, choose a unique public label $\ell_u \in \{0,1\}^\kappa$.
  5. For each node $u \in V_U$, perform the following:
     (a) For each $v \in V$, randomly choose a secret key $k_{u,w} \in \{0,1\}^\kappa$ associated with each node $w$ in $D(v)$, $R(v)$, and $L(v)$.
     (b) For each $v \in V$, construct public information about each edge in $D(v)$, $R(v)$, and $L(v)$ using the key derivation method.
     (c) For each $t \in T$, randomly choose a secret key $k_{u,t} \in \{0,1\}^\kappa$.
  6. For each $t \in T$, compute public information to permit key derivation between classes: for each edge $(u_1, u_2) \in E_U$ compute public information by setting $S_{u_1} = k_{u_1,t}$ and $S_{u_2} = k_{u_2,t}$ and using the key derivation method and public labels $\ell_{u_1}$ and $\ell_{u_2}$.
  7. For each $t \in T$, let $V_t \subset V$ denote the set of nodes in $G$ access to which implies access to $t$. Then for each $V_t$, for each $v \in V_t$:
     (a) Find in $D(v)$ the node corresponding to the time interval $t$; call it $w$.
     (b) For each $u \in V_U$, compute public information to permit derivation of $t$'s access key from $w$'s enabling key $k_{u,w}$ using the key derivation method and public label $\ell_t$. Mark such an edge with the level of $v$ and type $D$.
     (c) repeat (a) and (b) for $R(v)$ and $L(v)$, using types $R$ and $L$, respectively.
  8. Let $K$ consist of the secret keys $k_{u,t}$ for each $t \in T$ and $u \in V_U$, and let $\mathsf{Sec}$ consist of the remaining secret keys $k_{u,w}$. Let $\mathsf{Pub}$ consist of $G$, all public labels, and public information about all edges generated above.
```

Figure 5.7. Description of proposed time-based hierarchical key assignment scheme.

```
Algorithm $\mathsf{Assign}_T(u, T_{\mathcal{U}}, \mathsf{Sec})$:
  1. Execute $\mathsf{AssignKeys}(T_{\mathcal{U}}, root, T)$ using the data structure stored in $\mathsf{Pub}_u^G$, where $root$ is the root node of $G$.
  2. Set $S_{u,T_{\mathcal{U}}}$ to the keys computed and return $S_{u,T_{\mathcal{U}}}$.
Algorithm $\mathsf{Derive}_T(u_1, u_2, T_{\mathcal{U}}, t, S_{v,T_{\mathcal{U}}}, \mathsf{Pub})$:
  1. If $t \notin T_{\mathcal{U}}$ or $u_2 \notin Desc(u_1, G)$, return $\perp$.
  2. Execute $\mathsf{DeriveKey}(T_{\mathcal{U}}, t, S_{u_1,T_{\mathcal{U}}}, \mathsf{Pub}_{u_1}^G)$ to compute an enabling key for $t$, $k'_{u_1,t}$.
  3. Use $k'_{u_1,t}$ along with its (level-type) label and $\mathsf{Pub}_{u_1}^G$ to derive key $k_{u_1,t}$.
  4. Use $k_{u_1,t}$ and $\mathsf{Pub}_t^{G_U}$ to derive $k_{u_2,t}$ using the key derivation method.
```

Figure 5.8. Description of proposed time-based hierarchical key assignment scheme (continued).

compute public information that permits derivation of $k_{u_2,t}$ from $k_{u_1,t}$. Finally, Step 7 is similar to Step 5 in Figure 5.5 and allows computation of t's access keys from an enabling key corresponding to t at any level of granularity in the data structure G.

The fact that keys for an access class are assigned independently of the keys for other access classes allows us to state the following result:

Theorem 5.4.1 *Assuming the security of the family of pseudo-random functions F^κ, the time-based key assignment scheme for hierarchically organized access classes given in Figure 5.7 is both complete and sound with respect to key recovery in the presence of a static adversary.*

Proof Similar to the proof of Theorem 5.3.1, in this case we also use a hybrid argument and construct a sequence of experiments $\mathbf{Exp}^0_{\mathsf{KA},\mathcal{A}_{st}}, \ldots, \mathbf{Exp}^q_{\mathsf{KA},\mathcal{A}_{st}}$ for adversary $\mathcal{A}_{st}$ who attacks the scheme at class v during time interval t, defined as follows:

```
Experiment $\mathbf{Exp}^{i}_{\mathsf{KA},\mathcal{A}_{st}}(1^\kappa)$
    $(K, \mathsf{Sec}, \mathsf{Pub}') \leftarrow \mathsf{Set}^{i}_{T}(1^\kappa, G_U, T)$
    $corr \leftarrow \mathsf{Corrupt}_{v,t}(\mathsf{Sec})$
    $k \leftarrow \mathcal{A}_{st}(1^\kappa, \mathsf{Pub}', corr)$
    if $k = k_{v,t}$ then return 1
    else return 0
```

In the experiments, Set^0 corresponds to the original algorithm Set, and Set^{i+1} is constructed from Set^i by replacing public information about a single edge in the data structure by a random string. The edges replaced are:

1. For each access class $u \in Anc(v, G_U)$, replace in Pub^G_u all of the edges that were replaced in Pub for a single resource in the proof of Theorem 5.3.1 (in $D(w)$, $R(w)$, and $L(w)$ for all w of interest and in the top-down fashion).

2. Replace in $\mathsf{Pub}^{G_U}_t$, starting at the root[1], information about edges $(u, w) \in E_U$ for each $u \in Anc(v)$.

Thus, $\mathbf{Exp}^{0}_{\mathsf{KA},\mathcal{A}_{st}}$ is the same as $\mathbf{Exp}^{\mathrm{key-rec-h}}_{\mathsf{KA},\mathcal{A}_{st}}$, while $\mathbf{Exp}^{q}_{\mathsf{KA},\mathcal{A}_{st}}$ has no information related to $k_{v,t}$ at the level of v or any of its ancestors. This means that $\mathcal{A}_{st}$ has at most negligible probability in succeeding in $\mathbf{Exp}^{q}_{\mathsf{KA},\mathcal{A}_{st}}$.

Since the number of edges replaced is clearly polynomial in the security parameter, we can use a standard reduction argument to show that any non-negligible advantage between any $\mathbf{Exp}^{i}_{\mathsf{KA},\mathcal{A}_{st}}$ and $\mathbf{Exp}^{i+1}_{\mathsf{KA},\mathcal{A}_{st}}$ can be used to break the security of pseudo-random functions. Since by our assumptions the PRF is secure, the scheme is secure as well. □

To achieve key indistinguishability in this scheme, as before, we need to utilize the extended key derivation method that prevents key testing. In this case we need to use this method within the data structure G itself (in Step 5b of Set_T) to prevent a member of class u from testing keys of unauthorized time intervals. We also need to

[1] If several roots exist in G_U, sort the nodes using any topological ordering.

use this key derivation method between user classes (in Step 6 of Set_T) to prevent a member of class u from testing keys of its ancestor classes.

5.5 Practical Considerations

As was mentioned earlier, the goal of the solutions of this chapter is efficiency under the assumption that the number of unit intervals m in the system is large. In systems when this is not the case, other, simpler solutions will suffice (*e.g.*, a simple binary tree built on top of m intervals), and it is common sense to assume that a suitable for the context solution will be chosen. We, however, believe that our solution will find its uses in a number of domains such as, for instance, access to historical data. And even in applications where access is based on the current time, the service provider will be free to choose the level of granularity for time-based access rights. For broadcast-based services, there is no overhead in changing keys often.

Another consideration is that, in subscription-based services where access is based on current time, dues might be paid in installments. That is, a user subscribes only to a rather short sequence of intervals and renews her subscription on a periodic basis. But even such systems might be setup for a long time in the future, and the service provider will choose a solution that minimizes system and user resources.

5.6 Comparison with Existing Solutions

Table 5.3 compares performance of our scheme with other existing solutions; only security against recovery was considered. In the table, $diam(G_U)$ denotes the diameter of the graph (*i.e.*, maximum distance between nodes) that bounds the number of operations which, given a class key, are necessary to derive the key of the target descendant class within the user hierarchy. Also, $|E_U|$ denotes the number of edges in a user hierarchy G_U. The table does not list private storage at the server since it is equivalent for all solutions. Before proceeding with comparing existing results, we briefly explain what these parameters mean.

Table 5.3
Comparison of time-based hierarchical KA schemes.

Scheme	Public information	Private information	Key derivation	Operation type	Complexity assumption
Encryption-based [9]	$O(\|V_U\|^2\|T\|^3)$	1	1	decryption	one-way functions
Pairing-based [9]	$O(\|V_U\|^2)$	$O(\|T\|)$	1	pairing evaluation	Bilinear Diffie-Hellman
Binary tree	$O(\|E_U\|\|T\|)$	$O(\log \|T\|)$	$O(\log \|T\| + diam(G_U))$	PRF	one-way functions
ISPIT+(3,1)-CSBT +EBC [65]	$O(\|E_U\|\|T\| + \|V_U\|\|T\| \times \log \|T\| (\log\log \|T\|)^2)$	≤ 3	$O(diam(G_U))$	decryption	`IND-P1-CO` encryption [98]
Our 4HS-based	$O(\|E_U\|\|T\| + \|V_U\|\|T\| \times \log^* n \log\log \|T\|)$	≤ 3	$O(diam(G_U))$	PRF	one-way functions
ISPIT+(3,1)-CSBT +EBC [65]	$O(\|E_U\|\|T\| + \|V_U\|\|T\| \times \log \|T\| \log\log \|T\|)$	≤ 3	$O(\log^* \|T\| + diam(G_U))$	decryption	`IND-P1-CO` encryption [98]
Our log*HS-based	$O(\|E_U\|\|T\| + \|V_U\|\|T\| \times \log\log \|T\|)$	≤ 3	$O(\log^* \|T\| + diam(G_U))$	PRF	one-way functions

In the great majority of cases, the depth of user hierarchies is a small constant, resulting in $diam(G_U)$ being a small constant as well. In cases where the depth of the original graph G_U is fairly large and it is unacceptable to have the user perform $diam(G_U)$ operations, we can insert shortcut edges either at random (if $diam(G_U) = O(V_U)$) or using the techniques of Chapter 4 that reduce the diameter of the hierarchy to a small constant at the expense of small increase in the public storage.[2] Thus, in this case $diam(G_U)$ is also a small constant, and parameter $|E_U|$ will need to be replaced with a slightly larger value.

We also would like to mention that the schemes of [63] and [64] are not listed in the table. These solutions allow a user to obtain access to an arbitrary subsequence of time intervals (*i.e.*, not restricted to contiguous sequences considered in all other schemes), but require far slower key derivation of $O(|V_U| \cdot |T|)$ modulo exponentiations.

Considering that small private user storage and fast key derivation, followed by reasonable server storage are the main evaluation criteria, we can analyze the solutions as follows. The Pairing-based scheme of [9] will have the slowest key derivation time among all of the schemes listed here, as it uses pairing evaluation rather than fast encryption or PRF operations. Additionally, the number of secret keys a user has to maintain is large.

Compared to the Encryption-based scheme of [9], our key derivation time is higher by a constant factor, private storage is similar (*i.e.*, three keys instead of one), but the amount of public information the server must maintain is much lower than in that scheme. That is, for modest values of $|T| = 1000$ and $|V_U| = 10$, the encryption-based schemes requires storing on the order of 10^{11} labels, while in our case it will be bound by the order of 10^6 labels.

While the simple binary-tree approach has asymptotically higher performance, for small values of $|T|$ it will be preferred due to its simplicity. However, for the

[2] It should be noted that the shortcutting techniques of Chapter 4 may fail on hierarchies the dimension of which is hard to approximate, but we believe that such cases are very rare for the applications we consider in this work.

applications we envision, other solutions exhibit better performance. Thus, our recommendation is to use the simplest approach suitable for a particular setup.

The work of De Santis *et al.* [65] lists solutions with different performance parameters, and we include in the table only selected two schemes. That is, we chose two schemes that require a user to store 3 private keys (just like in our solutions) and where time-based key derivation involves $O(1)$ and $O(\log^* n)$ decryptions, respectively. This allows us to directly compare the schemes of [65] with our schemes. As can be seen from the table, the solutions exhibit very similar performance, with CSBT-based constructions having an additional factor of $\log |T|$ in the public storage space.

To summarize, our solution offers very attractive characteristics and superior performance compared to other existing solutions: each user in the system receives a small (≤ 3) number of keys, (off-line) computation of any access key involves a small number of very efficient operations, and the public storage required by our solution is only slightly higher than the number of access keys that the system must maintain. It is a very balanced solution in terms of its performance.

5.7 Extensions

5.7.1 Extending the Lifetime of the System

So far in all of our discussion we considered the lifetime of the system to consist of a fixed set of time intervals $\{t_1, \ldots, t_m\}$. In many applications, however, there might eventually be a need to support time intervals beyond the original m intervals. In this section, we describe techniques for extending the number of time intervals.

One simple approach is to apply the techniques of Section 5.3 to a second set of intervals. The interesting case is when a user's access rights span across the boundary (*i.e.*, t_n and t_{n+1}), and this case results in two sets of keys being issued to that user. This is particular appealing in applications where users purchase a subscription for a period of time (*e.g.*, they can view a collection of media objects on a specific day or month), after expiration of which there is no need to maintain keys for that period. However, this approach is less desirable in applications where objects are assigned a date (*e.g.*, a user requests access to all movies released in 1977), because previous intervals need to be maintained even after they have elapsed.

Suppose that the keys for previous time intervals need to be maintained. One approach is to extend the time intervals, rebuild the data structure, and recompute the public information. The downside of this approach is that all of the public information has to be recomputed (previous shortcuts may no longer be necessary and other shortcuts may need to be added), but if extensions to the time intervals are rare (which we assume is the case almost all of the time), then this may be acceptable. If recomputing all of the public information is unacceptable, then in some cases we can reuse the previous information. The simplest technique to achieve this is to set the new number of time intervals to m^2 (recall that building the tree data structure involves partitioning the time intervals into chunks of size of square-root of their number). Unfortunately, squaring the number of intervals is prohibitively expensive, but if we assume that m is a power of 2 and is a perfect square, then we can achieve full reuse of the previous information by doubling the length of a time interval. The basic idea of this approach is that, in the data structure for m^2 intervals, the subset of the data structure that effects the first $2^c m$ $(c < \log m)$ intervals has size $O(\mathit{space}(\mathcal{S}1(2^c m)) \log\log(2^c m))$. Thus, we can use this subset for the intervals, and when we need to add more intervals we can simply add the new information from the data structure for m^2 intervals.

In more detail, the part of the data structure for m^2 intervals that corresponds to the first $2^c m$ time intervals has the following characteristics. The depth of recursion is $\log\log m + 1$, and at the first level of recursion the time intervals are partitioned into blocks of size m. Since we already established bounds for systems with m intervals, here we need to evaluate performance at the highest level of recursion. When we build the data structure $D(\mathit{root})$ for 2^c coarse units, it will result in $O(2^c \cdot \mathit{space}(\mathcal{S}1(2^c)))$ space. This space is dominated by the space required for data structures $L(\mathit{root})$

and $R(root)$, both of which introduce $O(space(\mathcal{S}1(2^c m)))$ shortcut edges. The space needed for the scheme at all other levels of recursion is the sum of space for each m-interval block: $O(2^c \cdot space(\mathcal{S}1(m)) \log\log m)$. Thus, the total space for the data structure does not exceed $O(space(\mathcal{S}1(2^c m)) \log\log(2^c m))$.

The bounds of the key assignment and derivation algorithms are not affected by this increase, because we are only adding one additional level of recursion to the data structure, and time key derivation time at all levels of recursion is equivalent.

5.7.2 Handling Changes to the Hierarchy

In a number of applications, there might be situations where the user hierarchy needs to be changed after the keys have been distributed to users. For example, suppose that in a subscription-based service users are given privileges for one year (and thus can compute keys for up to a year in the future). Now suppose that some time later the service provider decides to add a new subscription class or to remove a subscription class (and re-assign the clients to other classes). In this section, we discuss techniques for making such changes to the hierarchy.

Before we proceed with outlining how such changes can be performed, recall from Chapter 3 that removal of access rights (such as edge and node deletions, as well as user revocation) will be transparent to the users only if the extended key assignment scheme is used.

Adding nodes. To add a node v to the access graph for time intervals $t_i, \ldots, t_j$, we choose keys $k_{v,t_i}, \ldots, k_{v,t_j}$ and compute the public information Pub_v^G for these time intervals according to the keys. Then using the procedure for adding nodes from Section 3.4, the access node v is added to the hierarchy at times $t_i, \ldots, t_j$ including the necessary information to $\mathsf{Pub}_{t_i}^{G_U}, \ldots, \mathsf{Pub}_{t_j}^{G_U}$.

Adding edges. To add an edge from v to w for time intervals $t_i, \ldots, t_j$, the necessary information is added to $\mathsf{Pub}_{t_i}^{G_U}, \ldots, \mathsf{Pub}_{t_j}^{G_U}$ as described in Section 3.4.

Changing an access class's key. To change a specific access level v's key at a specific time t_i, the procedure of Section 3.4 is used to change the hierarchy key at time t_i. Then we update $\mathsf{Pub}_v^{G_u}$ and $\mathsf{Pub}_{t_i}^{G_U}$ accordingly. Note that this can be done without re-keying any users.

Removing an edge. To remove an edge (u, v) at a specific time, the edge is removed and the keys of all descendants of u are changed according to the previous algorithm. Note that no user needs to be re-keyed.

Removing a node. To remove a node u at a specific time, all edges that point to u or originate from u are removed. Then u's vertex information at that time is forgotten. If u is removed for all time intervals, then its temporal keys can be forgotten as well.

So far this discussion has focused on changes to the access hierarchy. However, changes to user permissions are more likely than changes to hierarchy. For example, the system may want to add new users, or remove a user (*e.g.*, that has stopped paying), or to change a user's access level. To avoid having to re-key users other than the specific user, we use ideas from Section 3.4 for handling user-level changes. Specifically, users are not given direct access to keys, but rather each user is given her own key. Such key is then used to generate the system keys which is achieved by encrypting the keys that the user should obtain with the user's key or adding public information that permits derivation of the appropriate key using user's key. This increases the required public storage by an additive factor linear in the number of users (which the server provides already maintains), but this prevents unnecessary re-keying.

Adding a user. To add a user, the user's keys are generated and then the temporal keys that are to be given to the user are encrypted with that user's keys and this information is added to the public storage.

Removing a user. To remove a user, the system needs to remove access to previous material (to prevent the ex-member problem). And thus every key that the user could derive must be changed and the public information must be updated so that other users can derive the updated information. This is a relatively expensive operation, and thus should be done infrequently or in batches.

Changing a user. To change the access class of a user or the time at which a user can access material, the system simply needs to remove the user and then add the user with new rights. Note that in some special cases this can be done more efficiently. For example, if the user's rights are being escalated (*i.e.*, the new rights of the user are a superset of its current rights) then there is no need to remove the user or change any system keys (as the ex-member problem is irrelevant).

5.7.3 Faster Key Assignment

In situations where very fast key assignment to users is important, we can modify the AssignKeys algorithm of Section 5.3.2 to result in constant-time performance. To be able to achieve this, we store the recursion tree (call it RT) of the AssignKeys algorithm and use it to speed up the key assignment process. The time-consuming part of this algorithm is the step-by-step descent from the root until the node u of RT at which the desired keys reside. Thus, the keys we seek would be easy to assign in constant time if we could go directly to that node u. This, however, is easy to do once we observe that (i) the parent of u in RT is the lowest node whose interval contains $T_{\mathcal{U}}$ (*i.e.*, u is the nearest common ancestor in RT of the two leaves that correspond to the endpoints of I), and (ii) in any tree it is possible to answer nearest common ancestor (NCA) queries in constant time (see [92] for details). Thus, we augment the recursion tree with the information needed to answer such NCA queries in constant time and during the key assignment process after finding the right node u in RT we locally retrieve keys for the user time sequence $T_{\mathcal{U}}$.

6 KEY ASSIGNMENT IN GEO-SPATIAL SYSTEMS

The focus of this chapter is extending key management techniques to support user access rights in the geo-spatial domain. In particular, we assume we are given a large area which is partitioned into small cells, and a user obtains access to a sub-area on that grid. As before, we attempt to minimize the resources used at both the user and the server, and smart resource utilization becomes possible by appropriately exploiting the grid-based structure of the space.

6.1 Problem Description

The space consists of N cells in an $n_1 \times n_2$ grid, $N = n_1 n_2$; let us refer to the grid as S. Without loss of generality, we assume that $n_1 \geq n_2$ and that the grid has n_1 rows and n_2 columns. A user is permitted to obtain access to any specific sub-area within the grid. In general, user rights might permit access to areas of arbitrary shape (subject to the cell partitioning), which can be represented as a set of rectangles. Since the number of such rectangles in user access rights will be small in most applications, we will assume, for simplicity, that the user is given access to a single rectangular area R.

Then a grid cell will have an access key that permits access to the resources associated with that cell. This means that during the system initialization the grid cells will be assigned certain keys. Note that it will not always be the case that each cell has a unique key, because in some systems access to certain cells will always be granted in an all or none fashion (*i.e.*, if a user is allowed to access one cell in the group, then that user can access all cells in the group), in which case such cells can share the same key.

When a user joins the system and obtains access to an area R, she will be given a key (or a set of keys) that permits access to every single cell within the area (through a key derivation process). This means that, as before, we need algorithms to (i) setup the system, (ii) assign keys to users, and (iii) perform key derivation. And, from a user perspective, the interaction with the system consists of two phases:

(i) At the time of signing up, the user obtains secret keys that correspond to the area R to which access is being granted.

(ii) When the user would like to obtain access to a certain cell within R, she will use her secret keys (in combination with the public data made available by the server) to independently derive the access key for that cell. It is assumed that access to that key will permit her to either access the area or access information about the area, based on the context.

The security of a geo-spatial key assignment scheme is defined in a standard way. That is, we require the properties of completeness and soundness to hold, as defined below:

Completeness A user with access privileges to a rectangular area R is able to compute the access key for each cell within R.

Soundness Any coalition of users with access to rectangle areas $R_1, \ldots, R_k$ is unable to obtain access to any cell other than those contained in $R_1 \cup \cdots \cup R_k$.

As was mentioned above, the key assignment can be such that each cell obtains a unique key, but for efficiency reasons it might be advisable to assign the same key to multiple cells (when access to a certain area is always granted as whole and not at the level of individual cells). The fact that parts of the grid might have different access granularity (*i.e.*, the level of individual cells versus the level of blocks of cells) will allow us to achieve significant savings in the data structure and key derivation time in certain systems. However, we initially consider the case where the $n_1 n_2$ cells have distinct access rights (hence there needs to be a separate key for each cell). Later we discuss the case when groups of grid cells share a key (*i.e.*, there are disjoint, arbitrarily shaped regions of the grid, and each region has its own key).

Our result. If 4HS one-dimensional scheme is used as the underlying scheme for our construction, for an $n_1 \times n_2$ grid of cells that have distinct keys, a user who is entitled to access a rectangular sub-grid R of such cells is given a constant number of private keys, from which the key of any cell in R can be derived in constant time. Moreover, given any such R, it is possible to compute the private keys for it in constant time as well. The public storage space that the server needs to maintain is $O(n_1 n_2 (\log\log n_1)^2 \log^* n_1)$. Throughout this chapter, we assume that 4HS is used in the constructions and provide performance analysis according to the bounds of this scheme. Similar analysis can be carried out with out underlying schemes as well.

6.2 A Preliminary Scheme

In this section we examine the solution that follows from the results of Chapter 4. While the shortcut solution given in Chapter 4 performs well for general graphs, it does not exploit the spatial structure. As a result, more efficient solutions are possible, and we will use the scheme described in this section to build such a solution.

Chapter 4 provides a key derivation mechanism for a graph of dimension d. In the geo-spatial domain, we can represent each rectangular area on the grid by the coordinates of its four corners. For each corner, we can use its coordinates to form a total order relationship, which for all of them results in four total orders. By incorporating all possible rectangular sub-areas of the grid, we obtain a graph of dimension $d = 4$ to which the techniques of Chapter 4 can be applied.

Let us denote a cell by its x and y coordinates. A rectangular region R within the grid is described by two x coordinates $a \leq b$ and two y coordinates $c \leq d$, *i.e.*, by a 4-tuple representation (a, b, c, d), where (a, c) is R's bottom-left corner and (b, d) is R's top-right corner. If R is a single cell, then $a = b$ and $c = d$. For instance, in

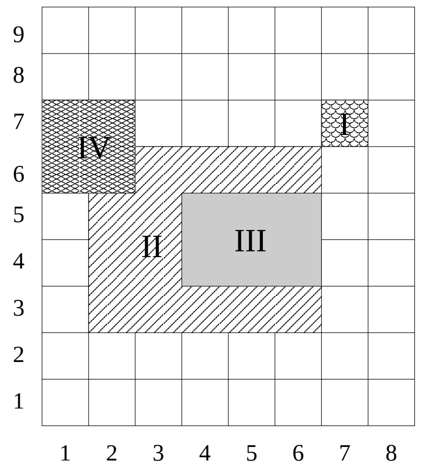


Figure 6.1. Illustration of regions on the spatial grid with $n_1 = 9$ and $n_2 = 8$.

Figure 6.1 Region I has coordinates (7, 7, 7, 7), Region II has coordinates (2, 6, 3, 6), Region III has coordinates (4, 6, 4, 5), etc.

Next, we create an N^2 $(= n_1^2 n_2^2)$ node graph G whose vertices correspond to all possible rectangles of the grid. That is, each vertex v in G is associated with a rectangle $R(v)$ whose bottom-left corner is $(a(v), c(v))$ and whose top-right corner is $(b(v), d(v))$. For reasons that will become apparent soon, we associate with such a vertex v the 4-tuple:

$$\tau(v) = (n_2 - a(v), b(v), n_1 - c(v), d(v)).$$

To illustrate this function on an example, we go back to Figure 6.1. Here if we associate node v_1 with Rectangle I, node v_2 with Rectangle II, and node v_3 with Rectangle III, then we have $\tau(v_1) = (1, 7, 2, 7)$, $\tau(v_2) = (6, 6, 6, 6)$ and $\tau(v_3) = (4, 6, 5, 5)$. Note that we have nodes in G and the corresponding values of the τ function for all possible rectangles on the grid.

Now observe that rectangle $R(v)$ contains rectangle $R(w)$ if and only if all 4 of the following inequalities hold:

$$a(v) \leq a(w),\ b(w) \leq b(v),\ c(v) \leq c(w),\ d(w) \leq d(v)$$

This is equivalent to:

$$n_1 - a(v) \geq n_1 - a(w),\, b(v) \geq b(w),$$
$$n_2 - c(v) \geq n_2 - c(w),\, d(v) \geq d(w)$$

which is the same as $\tau(v) \geq \tau(w)$. Hence rectangle $R(v)$ contains rectangle $R(w)$ if and only if $\tau(v) \geq \tau(w)$. This is also true when $R(w)$ is a single cell, *i.e.*, when $a(w) = b(w)$ and $c(w) = d(w)$.

We now describe the edge set of G: there is an edge in G from vertex v to vertex w if and only if $\tau(v) \geq \tau(w)$, *i.e.*, every one of the 4 components of the 4-tuple $\tau(v)$ is greater than or equal to the corresponding component of $\tau(w)$. Using rectangles from Figure 6.1, there will be an edge from $R(v_2)$ to $R(v_3)$, but no edges between any other ordered pair of the four rectangles depicted in the figure.

Now we have a 4-dimensional partial order G in which it is desired that v can derive the key of w if and only if v precedes w in G. Thus, we can use the solution of Chapter 4 to solve the problem with performance of 1 key, constant key derivation time, and $O(N^2(\log N)^3 \log^* N)$ space.

The following sections improve the performance of this data structure and result in a scheme that does not require the quadratic space while maintaining the constant number of keys and the constant key derivation computation.

6.3 Special Cases

Before presenting our scheme for the general case, we cover special cases. Solutions to these special cases will be used in the overall construction for the general case. The special cases considered in this section are rectangles that have less than 4 degrees of freedom, *i.e.*, each of them shares one or more of its 4 sides with the boundary of the grid.

6.3.1 Rectangles that Span the Grid

Let us first consider rectangles that span the whole width or whole height of the $n_1 \times n_2$ grid. This happens in one of two ways:

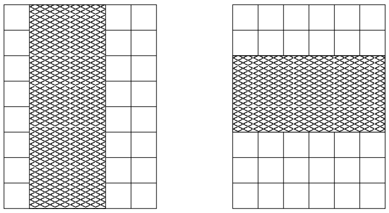


Figure 6.2. Illustration of rectangles that span the grid vertically and horizontally.

- *Vertical spanning*: The top and bottom boundaries of the rectangle are at rows 1 and (respectively) n_1.
- *Horizontal spanning*: The left and right boundaries of the rectangle are at columns 1 and (respectively) n_2.

Figure 6.2 illustrates such rectangles.

Given that users can only obtain access to rectangles that span the grid, our goal is to build a data structure that will permit a user to possess a small number of secret keys and derive access key to each cell of her region R. Without loss of generality, we give a solution for the case of horizontal spanning, and the case of vertical spanning can be addressed analogously.

In this special case, we can treat every row as a single "super-cell," ignoring the fact that it consists of many cells. Thus, we assign a key to each row, and this turns the problem into a problem with a single parameter. That is, now the only parameter that can change is the number of rows in user rectangle R. This makes it possible to apply the techniques of Chapter 5 for contiguous time sequences. In the current context, we allow a user to obtain access to a contiguous set of rows with each row having a different key. In other words, now the n_1 rows play the role the m time units played in Chapter 5. Then a user with access to a rectangle that spans the grid obtains secret keys created according to the solution of Section 5.3.

The above solution allows us to obtain the data structure of size $O(n_1 \log\log n_1 \times \log^* n_1)$ with the distance between any node and its descendant being at most a

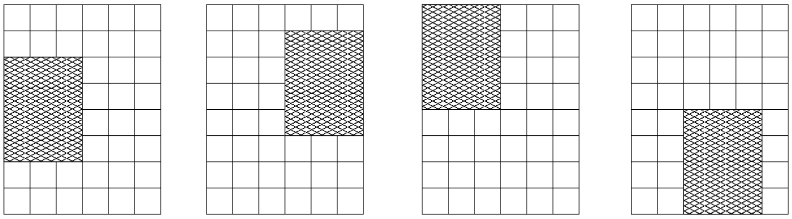


Figure 6.3. Illustration of rectangles that touch a grid's boundary.

constant number of edges. Note that none of the above characteristics depend on n_2, even though the grid is $n_1 \times n_2$.

The case of vertical (rather than horizontal) spanning is treated similarly to the above, except that the roles of rows and columns are interchanged, as are the roles of n_1 and n_2. Thus, the space complexity for vertical spanning is $O(n_2 \log\log n_2 \log^* n_2)$.

6.3.2 Rectangles that Share a Grid Boundary

Next, we consider rectangles that share at least one of their boundaries with a grid's boundary. This means that at least one of the bounding four coordinates of each rectangle of this type is in the set $\{1, n_1, n_2\}$. Figure 6.3 shows examples of such rectangles.

We use "row" as an abbreviation for "y coordinate" and "column" as an abbreviation for "x coordinate." Without loss of generality, assume that the rectangle touches the right boundary of the grid, *i.e.*, its right side is at column n_2 as in the second grid from the left in Figure 6.3. We call such a problem *right-anchored*, and the other three cases are analogously referred to as *left-*, *top-*, or *bottom-anchored.*

Our solution to the right-anchored case consists of applying a solution to one-dimensional graphs from Chapter 4 to cells in each row (because key derivation is unidirectional in this case, *i.e.*, from a certain point to the boundary). And we also apply the solution of Chapter 5 to each column (in this case, key derivation must be bounded by two points and key derivation is permitted within that interval only, just like in Chapter 5 for an interval of time). The algorithm for building the data structure is then as follows:

1. For each individual row i of the grid, we create a one-dimensional structure H_i that allows any position j in that row to have a short path to any other position j' of that same row iff $j < j'$.

2. For each individual column j of the grid, we use the single-parameter structure, call it V_j, of Chapter 5 to permit key derivation between any two positions i and i' ($i < i'$) in that column.

User Key Assignment and Derivation

Let a user be given access rights to a right-anchored rectangle R, the leftmost column of which is j. Then the user is given the keys that correspond to the set of rows of R from V_j. Derivation of the key of the cell at location (i', j') within R consists of using the user's secret keys and the V_j structure to obtain the key for the cell (i', j) at the same column, and then $H_{i'}$ structure of that row to derive the target key for cell (i', j').

Performance

The data structure built in Step 1 of the above algorithm for a single row has the characteristics of one secret key per user, constant key derivation time, and the size of the data structure (public storage) of $O(n_2 \log^* n_2)$ space. The total space for all rows is $O(n_1 n_2 \log^* n_2)$. The data structure built in Step 2 of the algorithm achieves, for a single column, a constant number of secret user keys, constant key derivation time, and $O(n_1 \log\log n_1 \log^* n_1)$ storage space for the data structure. The total public storage space for all columns is then $O(n_1 n_2 \log\log n_1 \log^* n_1)$ with constant-time key derivation.

The case of left-anchored rectangles is handled very similar to the case of right-anchored rectangles. The only difference is in Step 1 of the algorithm, where the data structure built should permit key derivation from a cell at position j to a cell at position j' iff $j > j'$. For top-anchored and bottom-anchored rectangles, the solution will consists of the roles of rows and columns reversed in the original algorithm. That is, we apply the solution of Chapter 4 to each column, and the solution of Chapter 5 to each row.

6.4 The General Case: The Initial Solution

This section describes our initial scheme for the general case of rectangles R with no assumptions other than that they must be contained within the $n_1 \times n_2$ grid. In Section 6.5 we show how to lower the space complexity associated with the storage required for the data structure.

Here we first describe in Section 6.4.1 how to build the data structure that permits efficient key derivation. This computation must be performed when the geo-spatial system is being setup. Then we show in Section 6.4.2 how, given a rectangle area R access to which is to be granted to a user, user secret keys are generated. This procedure is performed at the time a user joins the system and grants access privileges to access area R. Lastly, given access to R, we show in Section 6.4.3 how a user can obtain keys for a cell contained in R. It is assumed that access to such a key enables the user to obtain access to the area or information about the area, depending on the context.

6.4.1 The Data Structure

Now we describe how, for a grid S, to build the public tree data structure, which we will use for key management. Without loss of generality, we assume $n_1 = 2^{2^s}$

and $n_2 = 2^{2^q}$, where $s \geq q$.[1] The algorithm for building the data structure takes, as inputs, a node v and an $n_1 \times n_2$ grid S. It builds a tree for S rooted at v, using a recursive construction.

The idea behind it is to partition the grid into tiles of size $\sqrt{n_1} \times \sqrt{n_2}$ each, and apply the (preliminary) scheme of Section 6.2 on them treating each tile as a giant cell. Such a data structure will be able to handle key assignment and derivation at the granularity of tiles: now only rectangles that consist of a whole number of tiles are supported. Then the algorithm builds support for the special cases of Section 6.3 on the grid. In more detail, it builds data structures to support rectangles that border the boundary of the grid or span (vertically or horizontally) across one or more tiles. Finally, the algorithm is invoked recursively on each of the tiles to build the equivalent data structures at finer levels of granularity.

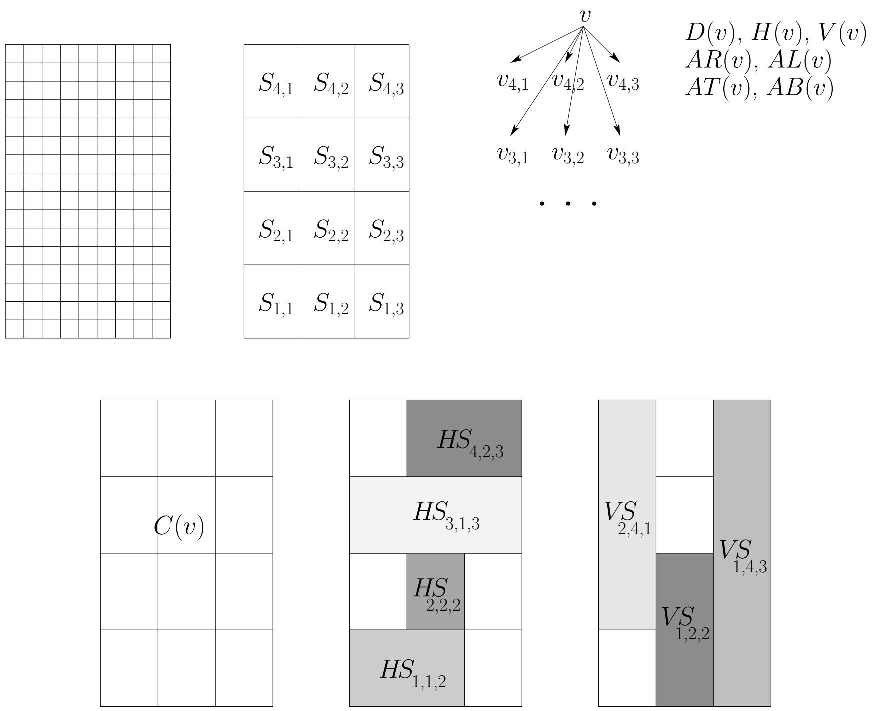
DataStructBuild(v, S):

1. If $n_2 = 2$ (*i.e.*, $q = 0$), then S consists of two columns of length n_1 each. For each of these columns, create and store at node v the data structure for the n_1 cells described in Chapter 5 (which will permit key assignment and derivation for any contiguous set of cells within those n_1 cells).

2. Partition S into a $\sqrt{n_1} \times \sqrt{n_2}$ array of *tiles* $S_{i,j}$, $1 \leq i \leq \sqrt{n_1}$ and $1 \leq j \leq \sqrt{n_2}$, where each tile is itself a $\sqrt{n_1} \times \sqrt{n_2}$ grid. That is, $S_{i,j}$ consists of the cells of S whose row number is in the interval $[(i-1)\sqrt{n_1}+1, i\sqrt{n_1}]$ and whose column number is in the interval $[(j-1)\sqrt{n_2}+1, j\sqrt{n_2}]$. Create a node $v_{i,j}$ for each $S_{i,j}$, and make $v_{i,j}$ a child of v.

3. Generate a grid $C(v)$, derived from S by treating each $S_{i,j}$ as a single cell (*i.e.*, "merging" the cells of $S_{i,j}$ into a single unit). Note that $C(v)$ is $\sqrt{n_1} \times \sqrt{n_2}$.

[1] Note that these assumptions are for presentation purposes only, and our data structures can easily be generalized to other grids that are not of this form. That is, it is not necessary to increase the actual size of the grid to obtain n_1 and n_2 of the above form, but instead rounding can be used in computing partitions of the grid.



$S_{4,1}$ $S_{4,2}$ $S_{4,3}$
$S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$
v
$v_{4,1}$ $v_{4,2}$ $v_{4,3}$
$v_{3,1}$ $v_{3,2}$ $v_{3,3}$
. . .
$D(v)$, $H(v)$, $V(v)$
$AR(v)$, $AL(v)$
$AT(v)$, $AB(v)$
$C(v)$
$HS_{4,2,3}$
$HS_{3,1,3}$
$HS_{2,2,2}$
$HS_{1,1,2}$
$VS_{2,4,1}$
$VS_{1,4,3}$
$VS_{1,2,2}$

Figure 6.4. Illustration of building the data structure for a grid 16 × 9 (first level of recursion).

4. Store at node v the scheme of Section 6.2 for $C(v)$, which we denote by $D(v)$. $D(v)$ will allow for user key assignment and derivation *at the granularity of tiles*, *i.e.*, it can process a rectangle only if that rectangle is the union of a subset of the $S_{i,j}$'s, but it cannot handle rectangles whose corners are inside the $S_{i,j}$'s.

5. Also store at node v a solution for each of the 4 "anchored" special cases (rectangles that have at least 1 side along a boundary for S). Call these structures $AL(v)$ for rectangles anchored at the left, $AR(v)$, $AT(v)$, and $AB(v)$ for rectangles anchored at the right, top, and bottom, respectively. Having these structures enables the handling of anchored rectangles.

6. Let $HS_{i,j',j''}$, where $1 \leq i \leq \sqrt{n_1}$ and $1 \leq j' \leq j'' \leq \sqrt{n_2}$, be the horizontal slab consisting of the union of all of the tiles $S_{i,j'}, S_{i,j'+1}, \ldots, S_{i,j''}$. For every such $HS_{i,j',j''}$, we store at v a "horizontal spanning" structure of Section 6.3.1 for

processing rectangles that horizontally span it. Since there are $\sqrt{n_1}$ choices for i and $\sqrt{n_2}$ choices for each of j', j'', the total number of such slabs is $n_2\sqrt{n_1}$. We denote by $H(v)$ the information stored at v for all of these $O(n_2\sqrt{n_1})$ horizontal slabs. $H(v)$ can handle any rectangle that horizontally spans any one of those slabs.

7. Similarly, let $VS_{i',i'',j}$, $1 \leq i' \leq i'' \leq \sqrt{n_2}$, be the vertical slab consisting of the union of all of the tiles $S_{i',j}, S_{i'+1,j}, \ldots, S_{i'',j}$. For every such $VS_{i',i'',j}$, we store at v a "vertical spanning" structure for processing rectangles that vertically span it. The number of such slabs is $n_1\sqrt{n_2}$. We denote by $V(v)$ the information stored at v for all of these vertical slabs. $V(v)$ can handle any rectangle that vertically spans any one of those slabs.

8. Recursively apply the scheme to each child of v, *i.e.*, call $\mathsf{DataStructBuild}(v_{i,j}, S_{i,j})$ for all $1 \leq i \leq \sqrt{n_1}$ and $1 \leq j \leq \sqrt{n_2}$.

Now we analyze the performance of the data structure built. In Step 1, we obtain a construction of $O(n_1 \log\log n_1 \log^* n_1)$ space, a constant distance between nodes, and a constant number of keys per user. In Step 4, the data structure built has the space complexity of $O(n_1 n_2 (\log n_1)^3 \log^* n_1)$ with a constant distance between nodes and one key per user. In Step 5, the construction gives us the space complexity of $O(n_1 n_2 \log\log n_1)$ with a constant distance between nodes and a constant number of user secret keys. In Step 6, we have that each structure $HS_{i,j',j''}$ has space complexity of $O(\sqrt{n_1} \log\log n_1 \log^* n_1)$, a constant number of keys per user, and a constant distance between nodes. Since there are $\sqrt{n_1}$ choices for i and $\sqrt{n_2}$ choices for each of j', j'', the total space for all such structures is $O(n_1 n_2 \log\log n_1 \log^* n_1)$. Finally, in Step 7, each structure $VS_{i',i'',j}$ similarly has space complexity of $O(\sqrt{n_2} \log\log n_2 \log^* n_2)$, with the total space over all choices of j, i', and i'' being $O(n_2 n_1 \log\log n_2 \log^* n_2)$.

Since Step 1 is at the bottom of the recursion, the total space satisfies the following recurrence if $n_2 > 2$:

$$f(n_1, n_2) \leq \sqrt{n_1 n_2} f(\sqrt{n_1}, \sqrt{n_2}) + c_1 n_1 n_2 (\log n_1)^3 \log^* n_1$$

and $f(n_1, 2) = c_2 n_1 \log\log n_1 \log^* n_1$, where c_1 and c_2 are constants. The solution to it is $f(n_1, n_2) = O(n_1 n_2 (\log n_1)^3 \log\log n_1 \log^* n_1)$.

The above "augmented tree" data structure (call it G) is used to set up the system, *i.e.*, we associate with each node of G a key, and we create public information associated with each edge (v, w) in G that allows anyone with v's key to derive w's key in one simple step.

In addition to the above data structure, the server will need to maintain public information associated with another, simple graph that maps keys corresponding to tiles to cell keys. A description of such an auxiliary data structure is given in Section 6.4.3 where we explain how key derivation is performed.

6.4.2 Key Assignment

We now turn our attention to describing how keys are assigned to a user who is being granted access to a rectangle area of cells R. In what follows, v is the root node of the above tree data structure, S is the $n_1 \times n_2$ grid associated with v (with $n_1 \geq n_2$), and R is an arbitrary rectangle in S. Although we can ultimately achieve computing the key assignment of any such R in constant time, we begin with describing a key assignment algorithm that does so in $O(\log\log n_2)$ time.

Initial Key Assignment Mechanism

Given a user's rectangle R, the recursive procedure below returns a constant-size set of secret user keys that will permit access to R. The algorithm majorly follows the data structure built using DataStructBuild to find the largest blocks of cells, keys for which are encoded in the data structure.

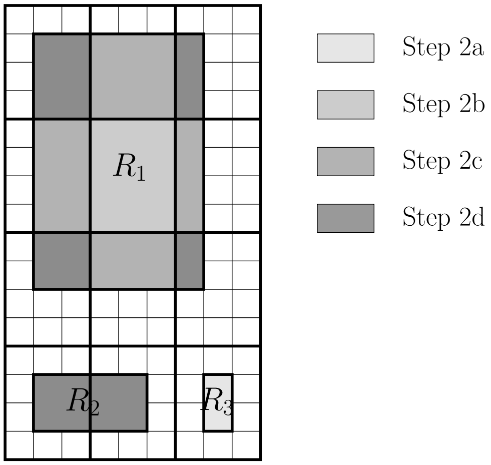


Figure 6.5. Illustration of the spatial key assignment procedure for various user rectangles. The color of an area indicates where in the AssignKeys algorithm the area is handled.

AssignKeys(R, v, S):

1. If v is not a leaf, continue with the next step. Otherwise, $n_2 = 2$ with a small n_1, and the data structure at v stores two solutions to the single-parameter problem of n_1 cells (one for each column). If R consists of a single column, we retrieve from the data structure corresponding to that column the keys that permit access to the range of R's rows. If R consists of two columns, we retrieve such keys from both data structures. Return the keys computed.

2. Recall from the algorithm that builds the data structure that the $v_{i,j}$'s are the children of v, and that $S_{i,j}$ is the $\sqrt{n_1} \times \sqrt{n_2}$ tile associated with node $v_{i,j}$. We distinguish different cases, based on how R overlaps with the $S_{i,j}$'s:

   (a) If R overlaps with only one $S_{i,j}$, then we recursively call AssignKeys$(R, v_{i,j}, S_{i,j})$ and return the keys returned by that recursive call. Otherwise, continue with the next steps.

   (b) Let R_1 be the maximal sub-rectangle of R that consists of the union of one or more $S_{i,j}$'s. If no such R_1 exists, then continue to the next step.

Otherwise, obtain the key for R_1 from the $D(v)$ structure stored at v (by indexing into it in constant time).

(c) Let R_2 be any of the maximal sub-rectangles of R that (i) are disjoint from R_1, and (ii) horizontally or vertically span one of the slabs $HS_{i,j',j''}$ or $VS_{i',i'',j}$. There can be at most 2 such horizontally spanning R_2's (a top and a bottom one), and at most 2 vertically spanning R_2's (a left and a right one). For each of such R_2's we obtain, in constant time, $O(1)$ keys by using the $H(v)$ or $V(v)$ structure stored at v.

(d) The previous steps have considered all but the "corners" of R, each of which could lie inside an $S_{i,j}$. Then let R_3 be any of the maximal sub-rectangles of R that contain a corner of R and are disjoint from R_1 and R_2's of the previous steps. Note that there will be at most four but could be fewer than four such R_3's, when, for instance, such an R_3 contains 2 corners of R (when both corners lie within the same $S_{i,j}$).

Each R_3 is surely "anchored," therefore the $O(1)$ keys for it can be obtained in constant time from one of the four structures $AL(v)$, $AR(v)$, $AT(v)$, and $AB(v)$. That is, we first index to the appropriate V_j data structure for left- and right-anchored rectangles (and H_j for top- and bottom-anchored rectangles) and then retrieve the right keys from it in constant time (as in Chapter 5).

(e) Return the keys computed in the previous three steps.

The different cases of Step 2 of this algorithm are depicted in Figure 6.5. The above procedure assigns a constant number of keys per rectangle R and does so in $O(\log\log n_2)$ time. In the next subsection we sketch a modification that brings the time down to constant.

As can be seen from above, Step 2b of the AssignKeys procedure returns from $D(v)$ keys associated with tiles, but not with individual cells. When users, however, want to obtain access to cells, they will need to have keys associated with those cells.

For that reason, the server must maintain a mapping from tile keys to the keys that compose the corresponding tiles. The data structure that allows such mapping is explained in Section 6.4.3 along with the key derivation process.

Constant-Time Key Assignment Mechanism

The above AssignKeys procedure takes longer than constant time because we are going down the tree G (working with blocks of finer granularity) until we find the node u at which R overlaps with more than one $S_{i,j}$. To achieve $O(1)$ time performance for key assignment, we need to find this u is constant time. This can be done as follows:

1. Let cells c_1, c_2, c_3, c_4 be the four corners of R. For every c_i, let $\ell(c_i)$ denote the leaf of G that contains c_i.

2. Use the constant-time algorithm for computing nearest common ancestors (NCA) in a tree [92] to compute the lowest (*i.e.*, farthest from the root) node of G which is an ancestor of all of $\ell(c_1)$, $\ell(c_2)$, $\ell(c_3)$, and $\ell(c_4)$. That node is the u we seek.

6.4.3 Constant-Time Key Derivation

The above key assignment process could also be used to guide the processing of a key derivation request. But before we proceed with giving it, we describe an auxiliary data structure, G', that will allow users to map keys associated with tiles to cell keys. Given the augmented tree G, G' is constructed as follows:

1. Starting with the root node v of G and going down the tree, add to the set of nodes of G' a node associated with each tile in $C(v)$.

2. Add to the set of nodes of G' a node for each cell of S.

3. For each node of G' that corresponds to a tile, insert an edge from it to every cell contained in that tile.

Given the above graph G', we assign a fresh unique public label to every node of it as in the key derivation mechanisms of Chapter 3. We then use the corresponding secret keys from G to compute public information associated with G'. Now each user who obtains a key for a tile in S will be able to use the public information for G' to obtain the key for any cell within that tile. The space complexity of G' is $O(n_1 n_2 \log\log n_1)$, which is lower than that of G.

Going back to the user key derivation procedure, we have that a user with secret keys for an area R should be able to obtain the key of a cell within R using the public information associated with the augmented tree G and the additional graph G' above. To do so, the user locates (in constant time using the NCA algorithm, as in the previous sub-section) the node u at which R overlaps with more than one $S_{i,j}$. Having u, the user derives the key depending on whether the target grid cell is in an R_1 (Step 2b), an R_2 (Step 2c), or an R_3 (Step 2d). All that is needed to carry out the constant-time derivation of the target cell's key is the local information stored at node u, *i.e.*, $AL(u)$, $AR(u)$, $AT(u)$, $AB(u)$, $H(u)$, $V(u)$, or data structures stored at leaves. The only exception is the case when the key is returned from $D(u)$ in Step 2b. In this case, such a key will correspond to a tile, but not to an individual cell. This means that the user will need to refer to G' to compute the cell's key from the tile key obtained above (by following one edge in G').

An alternative way to the use of NCA computations in locating u would be to provide the user who possesses access rights to R with a pointer to the node u, thereby allowing constant access to that node whenever that user needs to do key derivation.

6.5 Improving the Space Complexity

We now give an improvement in the space complexity of the solution. The improved scheme looks just like the initial scheme of Section 6.4, except that in Step 4 of $\mathsf{DataStructBuild}(v, S)$, instead of using the preliminary scheme of Section 6.2 for $C(v)$, it uses the better scheme of Section 6.4. This implies that the space for Step 4 of

DataStructBuild reduces to $\sqrt{n_1}\sqrt{n_2}(\log n_1)^3 \log\log n_1 \log^* n_1$, which is dominated by the $O(n_1 n_2 \log\log n_1 \log^* n_1)$ space for other structures $AL(v)$, $AR(v)$, $AT(v)$, $AB(v)$, $H(v)$, and $V(v)$. The recurrence for the total space thus becomes for $n > 2$:

$$f(n_1, n_2) \leq \sqrt{n_1 n_2} f(\sqrt{n_1}, \sqrt{n_2}) + c_1 n_1 n_2 \log\log n_1 \log^* n_1$$

and $f(n_1, 2) = c_2 n_1 \log\log n_1 \log^* n_1$, where c_1 and c_2 are constants. The solution is $f(n_1, n_2) = O(n_1 n_2 (\log\log n_1)^2 \log^* n_1)$.

As was mentioned earlier, in practice it is quite likely that not every cell will have its own distinct access key, and that groups of cells may have the same key. The easiest way to exploit this structure is for the recursive construction of G to stop as soon as its corresponding sub-grid consists of cells that all have the same key. That is, Step 1 of DataStructBuild needs to contain a termination test for when all of the $n_1 n_2$ cells share the same key (in which case it stops even if $n_2 > 2$). This is likely to result in less space that the worst-case theoretical bound we showed, especially when the cells that share a key tend to be contiguous (but not when they form a checkered pattern). We can quantify the improvement if we make assumptions about the shapes of those sub-regions of cells that share the same key (*e.g.*, assume a rectangular shape), which cannot be done without a specific application and setup parameters and usage patterns in mind.

The security of our construction is due to the key derivation mechanism used, which provides an appropriate protection mechanism for any DAG, and not due to the details of the data structure we build.

6.6 Handling Updates

It was already described earlier how keys are issued to a new user, therefore this section focuses on what happens when: (i) a cell's key is modified or (ii) a user's access rights are revoked. These issues were treated in Chapter 3 (Section 3.4) for hierarchical access control systems, and the same basic techniques work for the geo-spatial problem considered here.

Recall from Chapter 3 that we can use the extended scheme to avoid re-keying many users when a cell's key changes, because the scheme separates user secret information from the access keys. This approach avoids having to re-key every user who shares access to a cell with the revoked user, but there is still a need to re-key users who share the same key (from the data structures) with the user who is being revoked. In some environments, however, re-keying even a single user is expensive (or simply not possible). In such environments it is possible, using techniques of Section 3.4, to not require rekeying of any user for revocation. In more detail, each user is now given her own node in the public structure, and edges are added to the public structure from the user's node to the nodes containing keys for that user. Now one can change the keys for the underlying structure and can update the user's keys by modifying only public information. An added benefit of this approach is that each user needs to store only a single key. Note that this benefit comes as a cost: the public structure now grows linearly as the number of users increases.

6.7 Extensions

The scheme we gave extends to higher dimensions: Every additional dimension causes an additional $\log\log N$ factor in the space complexity, an extra constant number steps in key derivation, and a multiplicative constant factor in the number of keys. Therefore for a dimension d problem, we obtain:

(i) The number of keys is $O(c^d)$ for some constant c (and thus is only efficiently applicable when the number of dimensions is small);

(ii) The key derivation time becomes $O(d)$;

(iii) The space complexity becomes $O(N(\log\log N)^d \log^* N)$.

7 CONCLUSIONS

This work presents a comprehensive study of key management problem in systems where user privileges are based on hierarchical relationship between different classes. We addressed key management in the following contexts:

Hierarchies of user classes. Despite a large volume of work on hierarchical key assignment schemes, our scheme is the first solution that achieves provable security and supports arbitrary changes to the hierarchy without the need for key re-distribution to the current users. Our solution is also one of the most efficient and at the same time simple solutions to date.

In addition, our techniques for improving key derivation time by inserting of additional, shortcut, edges resulted in the possibility of applying key assignment solutions to a wider range of hierarchies and support for even very weak clients. These techniques were also proven very useful in extending the key management solutions to time- and space-based policies.

Time-based access control in hierarchical systems. In hierarchical systems, where time is partitioned into short intervals and access keys and privileges change during each time intervals, we present a solution that allows a user to join the system for a (possibly unique) consecutive set of time intervals while preserving user access privileges based on the hierarchical relationship between classes. Our solution provides a balance between the number of user keys, key derivation computation, and server requirements, resulting in very competitive performance.

Geo-spatial access control. By extending the notion of one-dimensional time to higher dimensions, we were able to construct key management solutions in the geo-spatial context as well. That is, we are given space partitioned into cells, and a user obtains access to a sub-area within the grid. Our solution heavily relies on the structured nature of cells and user access rights allowing for solutions with low overheads.

As a direction of future research, key management in file and storage systems appears as a promising direction for pursuing work. In particular, the inherently hierarchical structure of user directories and files makes it a natural fit for hierarchical key management solutions. As an evidence of potential in this research direction, the work of Grolimund *et al.* on file systems [99] was inspired by our work on user hierarchies [10]. Furthermore, users themselves are likely to be organized into groups resulting in application of key management techniques at both the levels of users and levels of resources. The drastic different in design principles, however, is that in file and storage systems users control their own data (there is no a single central authority anymore); they have read and write access to the data (instead of only read privileges); and users may grant access to their data to other users. Such differences are likely to require new key management techniques that are worth investigation.

LIST OF REFERENCES

LIST OF REFERENCES

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

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

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

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

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

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

[7] 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.

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

[9] G. Ateniese, A. De Santis, A. Ferrara, and B. Masucci. Provably-secure time-bound hierarchical key assignment schemes. In *ACM Conference on Computer and Communications Security (CCS'06)*, 2006. Full version is available as Cryptology ePrint Archive Report 2006/255, `http://eprint.iacr.org/2006/225`.

[10] M. Atallah, M. Blanton, N. Fazio, and K. Frikken. Dynamic and efficient key management for access hierarchies. Preliminary version appeared in *ACM Conference on Computer and Communications Security (CCS'05)*. Full version is available as Technical Report TR 2006-09, CERIAS, Purdue University, 2006.

[11] M. Atallah, M. Blanton, and K. Frikken. Key management for non-tree access hierarchies. In *ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 11–18, 2006. Full version is available as Cryptology ePrint Archive Report 2007/245, `http://eprint.iacr.org/2007/245`.

[12] M. Atallah, M. Blanton, and K. Frikken. Efficient techniques for realizing geo-spatial access control. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, March 2007.

[13] M. Atallah, M. Blanton, and K. Frikken. Incorporating temporal capabilities in existing key management schemes. In *European Symposium on Research in Computer Security (ESORICS'07)*, September 2007.

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

[15] J. Birget, X. Zou, G. Noubir, and B. Ramamurthy. Hierarchy-based access control in distributed environments. In *IEEE International Conference on Communications (ICC'01)*, pages 229–233, June 2001.

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

[17] C. Chang, I. Lin, H. Tsai, H. Wang, and T. Taichung. A key assignment scheme for controlling access in partially ordered user hierarchies. In *International Conference on Advanced Information Networking and Application (AINA'04)*, pages 376–379, March 2004.

[18] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[31] I. Lin, M. Hwang, and C. Chang. A new key assignment scheme for enforcing complicated access control policies in hierarchy. *Future Generation Computer Systems*, 19(4):457–462, 2003.

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

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

[34] I. Ray, I. Ray, and N. Narasimhamurthi. A cryptographic solution to implement access control in a hierarchy and more. In *ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 65–73, June 2002.

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

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

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

[38] Y. Sun and K. Liu. Scalable hierarchical access control in secure group communication. In *IEEE INFOCOM*, volume 2, pages 1296–1306, March 2004.

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

[40] Q. Zhang and Y. Wang. A centralized key management scheme for hierarchical access control. In *IEEE Global Telecommunications Conference (Globecom'04)*, volume 4, pages 2067–2071, 2004.

[41] Y. Zheng, T. Hardjono, and J. Pieprzyk. Sibling intractable function families and their applications. In *Advances in Cryptology – AsiaCrypt'91*, volume 739 of *LNCS*, pages 124–138, 1992.

[42] Y. Zheng, T. Hardjono, and J. Seberry. New solutions to the problem of access control in a hierarchy. Technical Report 93–02, Department of Computer Science, University of Wollongong, January 1993.

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

[44] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 98–111, 2006.

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

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

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

[48] W. Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(1):182–188, 2002.

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

[50] T.-S. Chen and Y.-F. Chung. Hierarchical access control based on Chinese remainder theorem and symmetric algorithm. *Computers & Security*, pages 565–570, 2002.

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

[52] A. De Santis, A. Ferrara, and B. Masucci. Efficient provably-secure hierarchical key assignment schemes. In *International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*, August 2007.

[53] A. Zych, J. Doumen, P. Hartel, and W. Jonker. A Diffie-Hellman based key management scheme for hierarchical access control. Technical Report TR-CTIT-05-57, Centre for Telematics and Information Technology, University of Twente, Enschede, December 2005.

[54] J. Fuh-gwo and W. Chung-ming. A practical and dynamic key management scheme for a user hierarchy. *Journal of Zhejiang University – Science A*, 7(3):296–301, March 2006.

[55] P. Vadnala and A. Mathuria. An efficient key assignment scheme for access control in a hierarchy. In *International Conference on Information Systems Security (ICISS'06)*, volume 4332 of *LNCS*, pages 205–219, December 2006.

[56] M. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT-LCS-TR-212, 1979.

[57] X. Yi and Y. Ye. Security of Tzeng's time-bound key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4):1054–1055, 2003.

[58] H. Chien. Efficient time-bound hierarchical key assignment scheme. *IEEE Transactions of Knowledge and Data Engineering (TKDE)*, 16(10):1301–1304, 2004.

[59] J. Yeh. An RSA-based time-bound hierarchical key assignment scheme for electronic article subscription. In *ACM International Conference on Information and Knowledge Management (CIKM'05)*, pages 285–286, 2005.

[60] Q. Tang and C. Mitchell. Comments on a cryptographic key assignment scheme for access control in a hierarchy. *Computer Standards & Interfaces*, 27:323–326, 2005.

[61] X. Yi. Security of Chien's efficient time-bound hierarchical key assignment scheme. *IEEE Transactions of Knowledge and Data Engineering (TKDE)*, 17(9):1298–1299, 2005.

[62] A. De Santis, A. Ferrara, and B. Masucci. Enforcing the security of a time-bound hierarchical key assignment scheme. *Information Sciences*, 176(12):1684–1694, 2006.

[63] Shyh-Yih Wang and Chi-Sung Laih. Merging: An efficient solution for a time-bound hierarchical key assignment scheme. *IEEE Transactions on Dependable and Secure Computing*, 3(1):91–100, 2006.

[64] W. Tzeng. A secure system for data access based on anonymous authentication and time-dependent hierarchical keys. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pages 223–230, 2006.

[65] A. De Santis, A. Ferrara, and B. Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. In *ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, June 2007.

[66] B. Briscoe. MARKS: Zero side effect multicast key management using arbitrarily revealed key sequences. In *First International Workshop on Networked Group Communication (NGC'99)*, LNCS, pages 301–320, 1999.

[67] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Advances in Cryptology – EUROCRYPT'02*, volume 2332 of *LNCS*, pages 65–82, 2002.

[68] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In *Workshop on Practice and Theory in Public Key Cryptography (PKC'03)*, volume 2567 of *LNCS*, pages 130–144, 2003.

[69] M. Bellare and A. Palacio. Protecting against key-exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 16(6):379–396, January 2006.

[70] G. Itkis and L. Reyzin. Sibir: Signer-base intrusion-resilient signatures. In *Advances in Cryptology – CRYPTO'02*, volume 2442 of *LNCS*, pages 499–514, 2002.

[71] Y. Dodis, M. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *CT-RSA'03*, volume 2612 of *LNCS*, pages 19–32, 2003.

[72] K. Fu. Integrity and access control in untrusted content distribution networks. Ph.D. Thesis, MIT, September 2005.

[73] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *European Symposium On Research In Computer Security (ESORICS'06)*, 2006.

[74] C. Patterson, R. Muntz, and C. Pancake. Challenges in location-aware computing. *IEEE Pervasive Computing*, 2(2):80–89, 2003.

[75] V. Atluri and S. Chun. An authorization model for geospatial data. *IEEE Transactions on Dependable and Secure Computing*, 1(4):238–254, 2004.

[76] E. Bertino, B. Catania, M. Damiani, and P. Perlasca. GEO-RBAC: A spatially aware RBAC. In *ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 29–37, 2005.

[77] H. Hu and D. Lee. Energy-efficient monitoring of spatial predicates over moving objects. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 28(3):19–26, 2005.

[78] C. Ardagna, M. Cremonini, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. Supporting location-based conditions in access control policies. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pages 212–222, 2006.

[79] M. Mokbel, W. Aref, S. Hambrusch, and S. Prabhakar. Towards scalable location-aware services: Requirements and research issues. In *ACM International Symposium on Advances in Geographic Information Systems (GIS'03)*, pages 110–117, 2003.

[80] U. Varshney. Location management for mobile commerce applications in wireless internet environment. *ACM Transactions on Internet Technology (TOIT)*, 3(3):236–255, 2003.

[81] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15, 1996.

[82] Y. Dodis, N. Fazio, A. Kiayias, and M. Yung. Scalable public-key tracing and revoking. *Journal of Distributed Computing*, 17(4):323–347, 2005.

[83] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption scheme secure against adaptive chosen ciphertext attack. *SIAM Journal of Computing*, 33(1):167–226, 2003.

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

[85] J. Crampton. On permissions, inheritance and role hierarchies. In *ACM Conference on Computer and Communications Security (CCS'03)*, pages 85–92, October 2003.

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

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

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

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

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

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

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

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

[94] 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.

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

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

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

[98] J. Katz and M. Yung. Characterization of security notions for probabilistic private-key encryption. *Journal of Cryptology*, 19:67–95, 2006.

[99] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 189–198, 2006.

VITA

VITA

Contact Information

Marina Blanton (née Bykova)
Computer Science and Engineering Department
University of Notre Dame, Notre Dame, IN
Office: 356C Fitzpatrick Hall
Email: `mblanton@cse.nd.edu`
WWW: `http://www.cse.nd.edu/~mblanton`

Research Interests

My research interests lie in information security and, in particular, include anonymity in access control systems, key management and authentication, privacy-preserving computation, and applied cryptography.

Education

Aug. 2007	PhD in CS, Purdue University. Advisor: Mikhail Atallah
Dec. 2004	MS in CS, Purdue University. GPA: 4.00.
Mar. 2002	MS in EECS, Ohio University. GPA: 3.93. Advisor: Shawn Ostermann
Jun. 1999	BS in CS with Honors, Tyumen State Oil and Gas University, Russia. GPA: 4.00. Advisor: Mikhail Karatun

Work Experience

Aug. 2002 – Jul. 2007	Teaching/Research Assistant, Department of Computer Science, Purdue University, West Lafayette, Indiana
Sep. 2001 – Jun. 2002	Network Engineer, Communications Network Services, Ohio University, Athens, Ohio
Sep. 1999 – Aug. 2001	Software Designer, Communications Network Services, Ohio University, Athens, Ohio
Sep. 1999 – Jun. 2001	Graduate/Teaching Assistant, School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio
Aug. 1998 – Aug. 1999	Systems Engineer, Sibnefteprovod, JSC, Tyumen, Russia
Dec. 1997 – Jun. 1998	System Coordinator Helper, Facilities Management, Ohio University, Athens, Ohio

Publications

In Refereed Journals

1. M. Blanton and M. Atallah, "Succinct Representation of Flexible Privacy-Preserving Access Rights," *Special Issue (Privacy-Preserving Data Management) of the International Journal on Very Large Data Bases (VLDBJ)*, Vol. 15, No. 4, pp. 334–354, Nov. 2006.

2. R. Balupari, B. Tjaden, S. Ostermann, M. Bykova, and A. Mitchell, "Real-Time Network-Based Anomaly Intrusion Detection," *Special Issue (Real Time Security) of the Journal of Parallel and Distributed Computing Practices*, Vol. 4, No. 2, Jun. 2001.

In Refereed Conference Proceedings

Acceptance rate included where known.

3. M. Atallah, M. Blanton, and K. Frikken, “Incorporating Temporal Capabilities in Existing Key Management Schemes,” *European Symposium on Research in Computer Security (ESORICS’07)*, Sep. 2007.

4. M. Atallah, M. Blanton, M. Goodrich, and S. Polu, “Discrepancy-Sensitive Dynamic Fractional Cascading, Dominated Maxima Searching, and 2-d Nearest Neighbors in Any Minkowski Metric,” *Workshop on Algorithms and Data Structures (WADS’07)*, Aug. 2007. (accept. rate: 26.7%)

5. M. Atallah, M. Blanton, and K. Frikken, “Efficient Techniques for Realizing Geo-Spatial Access Control,” *ACM Symposium on Information, Computer and Communications Security (ASIACCS’07)*, pp. 82–92, Mar. 2007. (accept. rate: 18.9%)

6. G. Ateniese, M. Blanton, and J. Kirsch, “Secret Handshakes with Dynamic and Fuzzy Matching,” *Network and Distributed System Security Symposium (NDSS’07)*, pp. 159–177, Feb. 2007. (accept. rate: 15.3%)

7. M. Atallah, M. Blanton, V. Deshpande, K. Frikken, J. Li, and L. Schwarz, “Secure Collaborative Planning, Forecasting, and Replenishment (SCPFR),” *Multi-Echelon/Public Applications of Supply Chain Management Conference*, Jun. 2006.

8. M. Atallah, M. Blanton, and K. Frikken, “Key Management for Non-Tree Access Hierarchies,” *ACM Symposium on Access Control Models and Technologies (SACMAT’06)*, pp. 11–18, Jun. 2006. (accept. rate: 30.5%)

9. M. Atallah, M. Blanton, K. Frikken, and J. Li, “Efficient Correlated Action Selection,” *Financial Cryptography and Data Security (FC’06)*, LNCS 4107, pp. 296–310, Feb. 2006. (accept. rate: 19.8%)

10. M. Atallah, K. Frikken, and M. Blanton, “Dynamic and Efficient Key Management for Access Hierarchies,” *ACM Conference on Computer and Communications Security (CCS’05)*, pp. 190–201, Nov. 2005. (accept. rate: 15.3%)

11. M. Atallah, M. Blanton, V. Deshpande, K. Frikken, J. Li, and L. Schwarz, "Secure Collaborative Planning, Forecasting, and Replenishment (SCPFR)," *Manufacturing and Service Operation Management (M&SOM)*, Jun. 2005.

12. M. Blanton and M. Atallah, "Provable Bounds for Portable and Flexible Privacy-Preserving Access Rights," *ACM Symposium on Access Control Models and Technologies (SACMAT'05)*, pp. 95–101, Jun. 2005. (accept. rate: 21.1%)

13. K. Frikken, M. Atallah, and M. Bykova, "Remote Revocation of Smart Cards in a Private DRM System," *Australasian Information Security Workshop (AISW'05), Digital Rights Management*, pp. 169–177, Jan. 2005. (accept. rate: 37.1%)

14. M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara, "Private Collaborative Forecasting and Benchmarking," *ACM Workshop on Privacy in the Electronic Society (WPES'04)*, pp. 103–114, Oct. 2004. (accept. rate: 23.3%)

15. M. Atallah and M. Bykova, "Portable and Flexible Document Access Control Mechanisms," *European Symposium on Research in Computer Security (ESORICS'04)*, LNCS 3193, pp. 193–208, Sep. 2004. (accept. rate: 17.0%)

16. M. Bykova and M. Atallah, "Succinct Specifications of Portable Document Access Policies," *ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, pp. 41–50, Jun. 2004. (accept. rate: 28.1%)

17. M. Bykova and S. Ostermann, "Statistical Analysis of Malformed Packets and Their Origins in the Modern Internet," *ACM Internet Measurement Workshop (IMW'02)*, pp. 83–88, Nov. 2002.

18. M. Bykova, S. Ostermann, and B. Tjaden, "Detecting Network Intrusions via a Statistical Analysis of Network Packet Characteristics," *IEEE Southeastern Symposium on System Theory (SSST'01)*, pp. 309–314, Mar. 2001.

Theses

19. M. Bykova, “Statistical Analysis of Malformed Packets and Their Origins in the Modern Internet,” *Master’s Thesis*, Ohio University, Mar. 2002.

Selected Technical Reports

Only technical reports that correspond to unpublished work or work under submission are listed.

20. M. Atallah, M. Blanton, and K. Frikken, “Incorporating Temporal Capabilities in Existing Key Management Schemes,” *Cryptology ePrint Archive Report 2007/245*, IACR, Jun. 2007.
21. M. Atallah, M. Blanton, and K. Frikken, “Efficient Key Derivation for Access Hierarchies,” *CERIAS Technical Report TR 2007-30*, Purdue University, Jun. 2007.
22. M. Blanton, “Online Subscriptions with Anonymous Access,” *CERIAS Technical Report TR 2007-29*, Purdue University, Jun. 2007.
23. M. Atallah, M. Blanton, N. Fazio, and K. Frikken, “Dynamic and Efficient Key Management for Access Hierarchies,” *CERIAS Technical Report TR 2006-09*, Purdue University, Apr. 2006.
24. M. Blanton, “Empirical Evaluation of Secure Two-Party Computation Models,” *CERIAS Technical Report TR 2005–58*, Purdue University, Aug. 2005.
25. M. Bykova, “What Should a Good Security Model Be?” *CERIAS Technical Report TR 2004–38*, Purdue University, Sep. 2004.

Editorial Activities

May 2006 – present M. Atallah and M. Blanton (Ed.), *“Handbook of Algorithms and Theory of Computation. Volume I: Foundations.”*

May 2006 – present	M. Atallah and M. Blanton (Ed.), *"Handbook of Algorithms and Theory of Computation. Volume II: Applications."*

Refereeing for Conferences/Journals

ACM Conference on Computer and Communications Security (CCS)

ACM Symposium on Access Control Models and Technologies (SACMAT)

ACM Symposium on Principles of Database Systems (PODS)

ACM Transactions on Information and System Security (TISSEC)

ACM Transactions on Internet Technology (TOIT)

Annual Computer Security Applications Conference (ACSAC) (On the 2006 reviewer committee)

Applied Cryptography and Network Security (ACNS)

Elsevier Computer Standards & Interfaces (CSI)

Elsevier Computers & Security

IEEE/ACM International Conference on High Performance Computing (HiPC)

IEEE Conference on Electronic Commerce (CEC)

IEEE Network and Distributed System Security Symposium (NDSS)

IEEE Security and Privacy (S&P)

IEEE Transactions on Computers (TC)

IEEE Transactions on Knowledge and Data Engineering (TKDE)

IEEE Transactions on Software Engineering

Information and Software Technology, International Journal

Information Processing Letters (IPL)

International Conference on Distributed Computing Systems (ICDCS)

International Conference on Information and Communications Security (ICICS)

International Conference on Information Security and Cryptology (ICISC)

International Journal of Information Security (IJIS)

International Symposium on Algorithms and Computation (ISAAC)

Workshop on Privacy Enhancing Technologies (PET)

Invited Talks

Apr. 2007	CS Seminar, Texas A&M University, College Station, TX
Apr. 2007	EECS Seminar Series, Case Western Reserve University, Cleveland, OH
Mar. 2007	ECE Seminar, Iowa State University, Ames, IA
Mar. 2007	CSE Seminar Series, University of Notre Dame, Notre Dame, IN
Mar. 2007	DCS Colloquium, Rutgers University, Piscataway, NJ
Mar. 2007	IBM T. J. Watson Research Center, Hawthorne, NY
Feb. 2007	CS Colloquium, Florida State University, Tallahassee, FL
Feb. 2007	Center for Applied Cybersecurity Research (CACR) Seminar, Indiana University, Bloomington, IN
Jan. 2007	CS Seminar, California Institute of Technology, Pasadena, CA
Jan. 2007	Security Seminar, University of Illinois at Urbana-Champaign, Urbana, IL
Oct. 2006	CSE Colloquium, the Pennsylvania State University, University Park, PA
Mar. 2006	CERIAS Security Seminar, Purdue University, West Lafayette, IN
Oct. 2004	Guest lecture at Database Security CS 590S, Purdue University, West Lafayette, IN
Sep. 2004	INRIA, the French National Institute for Research in Computer Science and Control, Rocquencourt, France
Jul. 2001	NASA Glenn Research Center, Cleveland, OH

Conference Talks Given

Mar. 2007	ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)
Mar. 2007	Network and Distributed System Security Symposium (NDSS'07)
Mar. 2006	Financial Cryptography and Data Security (FC'06)
Nov. 2005	ACM Conference on Computer and Communications Security (CCS'05)
Oct. 2004	ACM Workshop on Privacy in the Electronic Society (WPES'04)
Sep. 2004	European Symposium on Research in Computer Security (ESORICS'04)
Jun. 2004	ACM Symposium on Access Control Models and Technologies (SACMAT'04)
Nov. 2002	ACM Internet Measurements Workshop (IMW'02)
Mar. 2001	IEEE Southeastern Symposium on System Theory (SSST'01)

Teaching Experience

- Worked as a Teaching Assistant for the total of 4 years and 8 different CS undergraduate and graduate courses (some with repetitions).
- Taught an undergraduate CS/COMT lab for 1 semester.
- Assisted in practice lab sessions for 2 years for an undergraduate CS course.
- Gave guest lectures in several courses.

Professional Service

Conferences

Publicity chair, International Conference on Information Systems Security (ICISS'07)

University

2004–2006 Graduate Student Board, Department of Computer Science, Purdue University

2004–2005 Graduate Committee, Department of Computer Science, Purdue University

2004–2005 Grade Appeal Committee, School of Science, Purdue University

2000–2001 Electronic Theses and Dissertations Committee, Ohio University

Awards and Recognitions

May 2007 CRA Travel Grant for attending CRA-W Career Mentoring Workshop

Apr. 2007 Women in Science Program (WISP) Travel Grant, Purdue University

Apr. 2007 Siemens scholarship in recognition of research achievements

Mar. 2007 Diamond Award (annual award for outstanding academic achievement), Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University

May 2006 UCLA Institute for Pure and Applied Mathematics (IPAM) Award for participation in the Securing Cyberspace (SC'06) Program (core participant, fall 2006)

Apr. 2006 Intel Foundation Ph.D. Fellowship Award for 2006–2008 academic years (second year declined)

Apr. 2004 Purdue Research Foundation (PRF) Summer Research Scholarship (support for June–July 2006)

Mar. 2004 Women in Science Program (WISP) Travel Grant, Purdue University

Sep. 2002 ACM/USENIX Student Travel Grant

Mult. years Winner of university-wide and participant of regional programming contests and olympiads in physics

Leadership

2006–2007	Member of Women in Science Program (WISP) leadership team
2006	Purdue Ballroom Competition Chair
2005–2006	Chair of the CS Graduate Student Board, Purdue University
1998–1999	Chair of the CS Undergraduate Student Board, Tyumen State Oil and Gas University

Membership

Upsilon Pi Epsilon (UPE), International Honor Society for the Computing Sciences

Association for Computing Machinery (ACM)

Institute of Electrical and Electronics Engineers (IEEE)