Dynamic and Efficient Key Management for Access Hierarchies

Mikhail Atallah, Keith Frikken, Marina Blanton
Department of Computer Science
Purdue University

ACM Conference on Computer and Communications Security
(CCS'05)
November 2005

Outline

- Problem description
- Base scheme
- Extensions
- Efficiency improvements
- Conclusions

Introduction

- There is a hierarchy of partially ordered access classes
- A user with access to a certain class obtains access to objects at that class and descendant classes in the hierarchy
- Such hierarchies are used in many domains including:
  - Role-Based Access Control (RBAC) models
  - content distribution and cable TV
  - project development
- All objects in the repository are stored encrypted
- Users obtain keys that permit *independent* key derivation and access to the authorized resources

Introduction (cont.)

- We want to assign keys to the repository and users to achieve
  - low number of keys per access class (private storage)
  - moderate server storage space (public info)
  - low key derivation time
  - collusion resilience of the scheme
  - locally contained changes to the hierarchy
- How many keys per node do we actually need?

Results in a Nutshell

- Our scheme has:
  - one key per access class
  - optimal public storage space
  - fast hash-based key derivation
  - locally contained changes
  - strong security

Problem Description

- Access hierarchy is modeled as a directed access graph G
  - nodes $V = \{v_i, \ldots, v_n\}$ correspond to access classes
  - each node has a set of objects associated with it
  - edges $E = \{e_1, \ldots, e_m\}$ are added according to the partial order relationship between the classes
  - a path from v_i to v_j means that v_i is entitled to access objects of v_j
- A key allocation KA implements G if it assigns keys to users and objects, where a user can access an object iff she has a key for that object

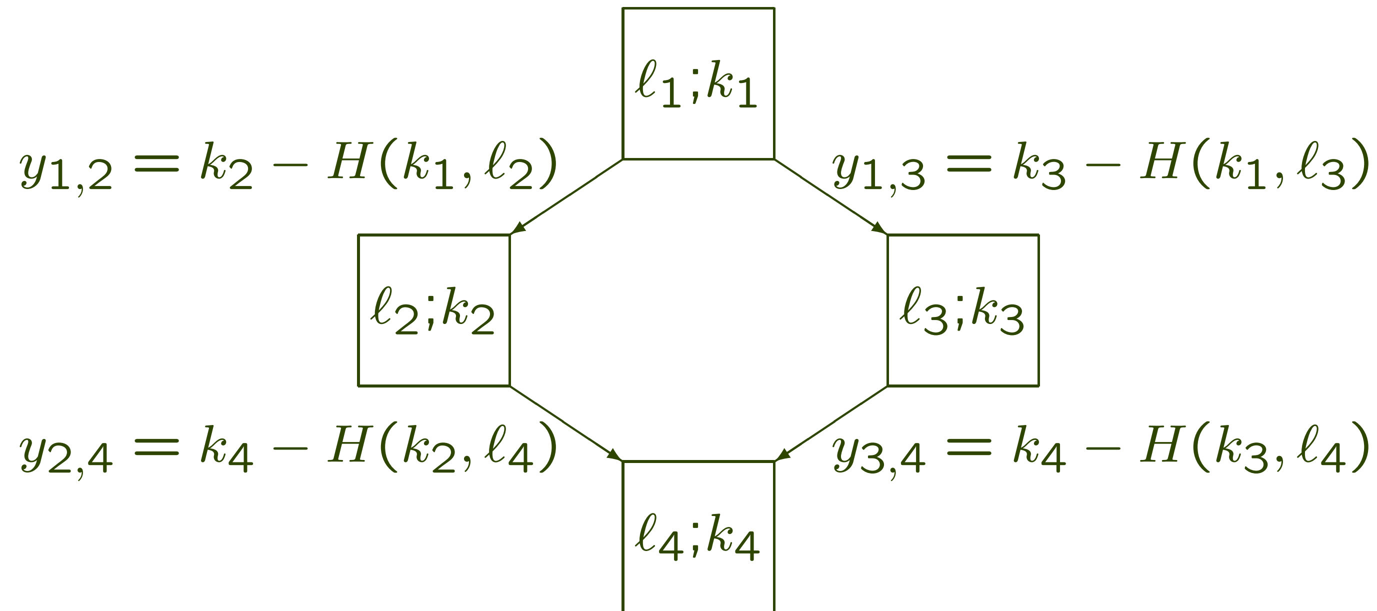
Base Scheme

- Key generation
  - Private key
    - each node v_i is assigned a private key $k_i \xleftarrow{R} \{0,1\}^\rho$.
    - a user with access to a set of nodes V_u obtains keys for all $v_j \in V_u$
  - Public information
    - for each node v_i, there is a unique label $\ell_i \in \{0,1\}^\rho$
    - for each edge (v_i, v_j), the value $y_{i,j} = k_j - H(k_i, \ell_j) \bmod 2^\rho$ is public

Base Scheme (cont.)

- Key derivation
  - suppose v_i is a parent of v_j
  - having k_i and public $y_{i,j}$, the key of v_i's child is computed as $k_j = y_{i,j} + H(k_i, \ell_j) \bmod 2^\rho$
  - repeat this procedure for nodes several edges away from the user key
- Properties
  - the above scheme is sound
  - the above scheme is complete, even in the presence of an active adversary who can adaptively corrupt nodes

Base Scheme (Example)



Key allocation for a sample access graph; all arithmetic is modulo 2^ρ.

Dynamic Version

- Modified private key
  - each node v_i is assigned a private key $\widehat{k}_i \stackrel{R}{\leftarrow} \{0,1\}^\rho$
  - as before, a user with access to nodes V_u obtains a key for each $v_i \in V_u$
  - the actual key used for v_i is now $k_i = H(\widehat{k}_i, \ell_i)$.
- The rest of the scheme remains unchanged

Dynamic Version (cont.)

- Deletion of an edge
  - want to prevent ex-member access after edge removal
  - suppose (v_i, v_j) is deleted, for each descendant v_t of v_j:
    - change v_t's label (call it ℓ'_t), which will change node's key to $k'_t = H(\widehat{k}_t, \ell'_t)$
    - for each parent v_p of v_t, update $y_{p,t}$ according to k'_t
- Key replacement
  - update v_i's key $\widehat{k}_i$ with a new key $\widehat{k}'_i$
  - update the access key to $k'_i = H(\widehat{k}'_i, \ell_i)$
  - update public information about incoming and outgoing edges according to the new key k'_i

Other Access Models

- Downward inheritance
  - construct the reverse of the access graph, $G^R = (V, E')$, such that for each $(v_i, v_j) \in E$, $(v_j, v_i) \in E'$
  - each node now has two keys and both upward and downward inheritance are supported
- Limited depth permission inheritance
  - let G be *layered* (nodes are partitioned into sets $S_1, \ldots, S_r$, where $\forall (v_i, v_j) \in E$, if $v_i \in S_k$, then $v_j \in S_{k+1}$
  - create a graph with a node for S_i and an edge from S_k's node to S_{k+1}'s node; create the reverse of this graph
  - assign keys to the graphs to support limited depth access
- Combine these techniques to support more complex policies

Efficiency Improvements

- In the base scheme, key derivation for an n-node access graph G takes $O(n)$ steps in the worst case
- This number can be reduced by adding so-called “shortcut” edges to G
- Let us assume that the access graph is an n-node tree T
- By adding $O(n)$ edges to T, key derivation can be done in $O(\log\log n)$ steps
- By adding $O(n \log\log n)$ edges to T, key derivation can be done in at most 3 steps

Efficiency Improvements (cont.)

- Background definition
  - a *centroid* of T is a node whose removal from T leaves no connected components of size $\geq n/2$
- Basic idea
  - for a node v of T, let T_v be the subtree of T rooted at v
  - compute the centroid of T_v (call it c_v)
  - add a shortcut edge from v to c_v
  - remove from T_v its subtree rooted at c_v
  - repeat this process with new T_v until it becomes empty

Efficiency Improvements (cont.)

- Executing this procedure for each node results in $O(n \log n)$ additional edges and $O(\log n)$ distance between any two nodes
- Other techniques involve *centroid decomposition*:
  - given a tree T, compute its centroid, remove it from T
  - recursively repeat with the remaining trees (of size $\leq n/2$)
- We use so-called "prematurely terminated" centroid decomposition
  - decompose as above, except stop the recursion when the tree reaches a certain size

Efficiency Improvements (cont.)

- Centroid nodes of the decomposition are special and are directly connected
- Special nodes are used as a “beltway” to connect small trees of T
- Shortcut edges within the small residual trees are added according to the basic procedure
- This allows us to lower both the number of shortcut edges and key derivation time

Conclusions and Future Work

- Features of the scheme developed
  - fast hash-based key derivation with a single key
  - collusion resilience against active adversaries
  - updates to the hierarchy are locally contained
  - key derivation is bound by the depth of the hierarchy and can be significantly reduced for trees
- Future directions
  - support for temporal constraints
  - support for "limited depth" permission inheritance in a collusion-resilient manner
  - other techniques for improving key derivation for general hierarchies