# Tree-based Cryptographic Access Control

James Alderman*, Naomi Farley**, and Jason Crampton
james.alderman@rhul.ac.uk, naomi.farley.2010@live.rhul.ac.uk,
jason.crampton@rhul.ac.uk

Royal Holloway, University of London,
Egham, Surrey, TW20 0EX

**Abstract.** As more and more data is outsourced to third party servers, the enforcement of access control policies using cryptographic techniques becomes increasingly important. Enforcement schemes based on symmetric cryptography typically issue users a small amount of secret material which, in conjunction with public information, allows the derivation of decryption keys for all data objects for which they are authorized.
We generalize the design of prior enforcement schemes by mapping access control policies to a graph-based structure. Unlike prior work, we envisage that this structure may be defined *independently* of the policy to target different efficiency goals; the key issue then is how best to map policies to such structures. To exemplify this approach, we design a space-efficient KAS based on a binary tree which imposes a logarithmic bound on the required number of derivations whilst eliminating public information. In the worst case, users may require more cryptographic material than in prior schemes; we mitigate this by designing heuristic optimizations of the mapping and show through experimental results that our scheme performs well compared to existing schemes.

## 1 Introduction

Access control is a fundamental security service in modern computing systems. Informally, requests from users to interact with protected resources are filtered and only those interactions that are *authorized* by a *policy* configured by the resource owner(s) are allowed. Software-based access control mechanisms are not appropriate when resources are stored by an untrusted third party. Instead, we may use cryptographic mechanisms whereby data objects are encrypted and authorized users are given appropriate cryptographic keys. The problem, then, is to efficiently and accurately distribute appropriate keys to users. Symmetric cryptography may be preferred over public key techniques (e.g. Attribute-based Encryption) due to their better efficiency and smaller ciphertext and key sizes.

Thus, in recent years, there has been considerable interest in *Key Assignment Schemes* (KASs) [1, 2, 5, 9, 13, 16, 21], which are particularly suitable for

* James Alderman was supported by the European Comission through H2020-ICT-2014-1-644024 "CLARUS".
** Naomi Farley was supported by the UK EPSRC through EP/K035584/1 "Centre for Doctoral Training in Cyber Security at Royal Holloway".

enforcing information flow policies. Such policies define a partially ordered set (poset) of security labels encoding hierarchical access rights [7]. KASs typically represent the poset as a directed acyclic graph [2, 13–16, 21, 22] and enable iterative key derivation along paths: users are issued a small number of *secrets* and users with security label $x$ can derive the key associated to $y < x$ using the secret associated with $x$ and public information associated with edges in a path from $x$ to $y$.

The general design goals of a KAS [16] are to minimize: a) the cryptographic material required by each user; b) the amount of public information required; and c) the computational cost of key derivations. Unsurprisingly, it is not possible to realize all objectives simultaneously, and so trade-offs have been sought. Derivation in early KASs was based on expensive computations [1]. The performance of more recent KASs is heavily dependent on the graph chosen to represent the policy. The graphs used in prior KASs are subsets of the transitive closure of the poset, often simply the Hasse diagram [14–16, 21]. Many works [2, 4, 12, 22] reduce derivation costs by adding 'shortcut' edges to the Hasse diagram but require a substantial amount of additional public information e.g. $\mathcal{O}(n^2)$ where $n$ is the number of labels in the policy and may itself be large, particularly when labels are defined in terms of subsets of attributes. Recent works [13–15, 17] aim for space-efficient KASs by eliminating public information via partitioning the Hasse diagram into chains or trees; however users may require additional secrets and it is not possible to bound derivation costs (beyond the trivial $\mathcal{O}(n)$).

In this work, we generalize the design approaches of prior KASs to consider mapping the policy poset to *any* directed acyclic graph, not only a subset of the transitive closure of the poset. In particular, one may choose such an enforcement structure *independently* of the poset to target particular design goals of the resulting KAS. The natural questions that then arise are 'what structure should we choose?' and 'how should the policy be mapped to this structure?'. We define the following steps to follow when designing a KAS:

1. Identify the design criteria to be optimized and choose an enforcement structure that provides these properties;
2. Choose a mapping from the policy poset to the enforcement structure that optimizes performance of the remaining criteria;
3. Instantiate a key derivation mechanism over the enforcement structure to define the keys and secrets to be used in the KAS.

Prior KASs were restricted in the choice of enforcement structure due to only considering trivial mappings to enforcement structures (i.e. nodes in the enforcement structure corresponded directly to labels in the poset). In contrast, we introduce additional flexibility by allowing one to optimize the choices of structure and mapping to achieve different design goals. We hope that this flexible design approach will spur the design of novel KASs to target specific requirements.

To illustrate our approach, we shall design a KAS which eliminates public information *and* in which derivation costs are logarithmically bounded; our example therefore bridges the gap between KASs [2, 4, 12, 22] that bound derivation costs and recent works which eliminate public information [13–15, 17] but

which cannot bound derivation. To achieve this goal, we use a binary tree as our enforcement structure. This choice is simple and intuitive to serve as an example, introduces interesting optimization problems when choosing the mapping, and reduces storage costs for users by removing the need for users to store the enforcement structure — derivation paths are immediately apparent from the security labels. Thus, our KAS may be applicable to settings in which storage for (possibly large) derivation information on client devices is limited and in which key derivation should be fast e.g. consider a smart card which must derive temporal access keys. We shall also see that our KAS permits very flexible assignment of access rights, lending itself to settings with diverse user populations.

The remaining design criteria to be optimized (through the choice of mapping from policy poset to enforcement structure) is the amount of cryptographic material required by users. As with [14, 15], removing public information results in users requiring additional secrets; in our case, the worst-case bound is $\lceil n/2 \rceil$ secrets. We develop heuristic methods for finding a mapping which minimizes the average number of secrets users must store and demonstrate via experimental evaluation that our scheme works well in practice. Indeed, we show that our scheme compares favorably with other KASs that require no public information.

We begin with relevant background material. In Section 3, we introduce our KAS based on a binary tree, before proposing methods to optimize the choices of structure and mapping in Section 4. Section 5 experimentally evaluates the KAS, and in Section 6 we discuss interesting policy features enabled by our scheme.

## 2 Background and Notation

A *partially ordered set* (poset) [15] is a pair $(L, \leqslant)$ where $\leqslant$ is a binary, reflexive, anti-symmetric, transitive order relation on $L$. For $x, y \in L$, we may write $y \geqslant x$ if $x \leqslant y$, and $x < y$ if $x \leqslant y, x \neq y$. We say that $y$ *covers* $x$, denoted $x \lessdot y$, if and only if $x < y$ and there exists no $z \in L$ such that $x < z < y$. We say that $x, y \in L$ are *incomparable* if $x \not\leqslant y$ and $y \not\leqslant x$. The *width* of a poset is the size of its largest set of incomparable elements. For $l \in L$, the *order filter* of $l$ is $\uparrow l = \{x \in L : x \geqslant l\}$ and the *order ideal* of $l$ is $\downarrow l = \{x \in L : x \leqslant l\}$.

An *information flow policy* [7] defines a poset $(L, \leqslant)$ of security labels, a set of users $U$, a set of data objects $O$, and a function $\lambda : U \cup O \to L$. A user $u \in U$ is authorized to read an object $o \in O$ if and only if $\lambda(o) \leqslant \lambda(u)$.

Key Assignment Schemes (KASs) [2, 16, 21] enforce read-only information flow policies, primarily using symmetric cryptography. A setup authority generates a unique key $\kappa_l$ associated to each label $l \in L$ and each data object $o$ is encrypted using $\kappa_{\lambda(o)}$. Each user $u$ requires the keys $\{\kappa_l : l \leqslant \lambda(u)\}$ to decrypt the objects for which she is authorized. Typically a KAS reduces the number of keys issued to users by giving each user a small amount of secret information from which they can derive all keys for which they are authorized. The strongest notion of security for a KAS (Key Indistinguishability [2]) requires that a collusion of users cannot distinguish a key for which they are not authorized from a random string (i.e. unauthorized users learn nothing about the keys used to pro-

tect objects). To achieve such a notion, one typically requires a strict separation between *secrets*, issued to users, and *keys*, used to encrypt and decrypt objects.

**Definition 1.** *A* Key Assignment Scheme *(KAS) for a poset* $(L, \leqslant)$ *comprises:*

- $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub) \xleftarrow{\$} \mathsf{Gen}(1^\rho, (L, \leqslant))$ *is a probabilistic polynomial-time algorithm run by a setup authority that takes a security parameter* $1^\rho$ *and* $(L, \leqslant)$ *and outputs a symmetric key* $\kappa_l$ *and a secret* $\sigma_l$ *for each* $l \in L$*, along with a set of public derivation information* $Pub$*;*
- $\kappa \leftarrow \mathsf{Derive}((L, \leqslant), x, y, \sigma_x, Pub)$ *is a deterministic polynomial-time algorithm run by a user to derive* $\kappa_y$ *from the secret material* $\sigma_x$*. It takes* $(L, \leqslant)$*, labels* $x, y \in L$*, the secret* $\sigma_x$*, and public information* $Pub$*, and outputs the derived key* $\kappa = \kappa_y$ *assigned to label* $y$ *if* $y \leqslant x$*, and outputs* $\kappa = \perp$ *otherwise.*

A KAS is *correct* if $\kappa_y \leftarrow \mathsf{Derive}((L, \leqslant), x, y, \sigma_x, Pub)$ for all $\rho \in \mathbb{N}$, all $(L, \leqslant)$, all $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub)$ output by $\mathsf{Gen}(1^\rho, (L, \leqslant))$, and all $x, y \in L$ such that $y \leqslant x$.

Let $\epsilon$ denote the empty string and $x \parallel y$ denote the concatenation of strings $x$ and $y$. The *power set* of a set $X$, denoted $2^X$, is the set of all subsets of $X$.

Let $G = (V, E)$ be a directed graph where, for vertices $x, y \in V$, $(x, y) \in E$ denotes a directed edge from $x$ to $y$. We say that $x$ is an *ancestor* of $y$ (and $y$ is a *descendant* of $x$) if there exists a directed path from $x$ to $y$ in $G$. The *Hasse diagram*, $H(L, \leqslant) = (L, E)$, of a poset $(L, \leqslant)$, is a directed graph with vertex set $L$ and where $(x, y) \in E$ if and only if $y \lessdot x$ in $(L, \leqslant)$. Let $H^\star(L, \leqslant) = (L, E^\star)$ be the *transitive closure* of $H(L, \leqslant)$, where $E^\star = \{(x, y) : y < x\}$.

A *matching* of an undirected graph $G = (V, E)$ is a set $M \subseteq E$ of pairwise non-adjacent edges i.e. no two edges in $M$ share a common vertex. When $G$ has weighted edges, a *maximum weight matching* $M$ in $G$ is a matching for which the sum of the weights of the edges in $M$ is maximal.

## 3 Our Construction

We begin by motivating our choice of enforcement structure according to the design goals of our example (to minimize public information and to bound derivation costs). We then show how to instantiate a KAS on this structure using a very simple key derivation mechanism.

### 3.1 Defining the Enforcement Structure

The best approach we currently know to construct KASs *without* public derivation information is to ensure that every vertex in the enforcement structure (directed acyclic graph) has in-degree at most 1 i.e. each secret is derived from at most one other secret [14, 15]. For this reason, we will choose a tree structure.

We shall restrict our focus to *binary* trees, which are simple to discuss in this introductory work whilst enabling a KAS in which users need not store the enforcement structure itself, further reducing storage costs. A *binary* tree also appears to be a reasonable choice in general: we shall see that the number of

secrets that must be issued can be reduced when multiple users are authorized for some set of access rights (security labels) and that these sets correspond to descendants of nodes in the tree; hence we may expect more users to share a set of labels when the size of that set is small i.e. when the degree of nodes is low.

The maximum derivation cost for any key is bounded by the maximal path in the enforcement structure. The minimal depth of a binary tree with $n$ leaves is $\lceil \log n \rceil$.[1] Internal nodes with a single child only increase derivation paths and so we restrict our focus to *full* binary trees (where all nodes have 0 or 2 children).

We therefore define our enforcement structure to be a rooted, full binary tree with $n$ leaves and of depth $\lceil \log n \rceil$. Note that there remain many such trees and many ways in which to map a specific policy poset to such a tree; these choices have a direct effect of the efficiency of the resulting KAS. In this section we shall assume that the specific tree and mapping are given and we shall show how to assign and derive secrets and keys (for an arbitrary policy). We consider methods to optimize these choices to enforce *specific* policies in Section 4.

### 3.2 Instantiating a KAS on our Enforcement Structure

Let $((L, \leqslant), U, O, \lambda)$ be a read-only information flow policy and let $n = |L|$ be the number of security labels in the policy. Suppose that we have chosen a specific full binary tree $T_n = (V, E)$ with $n$ leaves and depth $\lceil \log n \rceil$ and a bijective mapping $\alpha$ from security labels in $L$ to the *leaves* of $T_n$. Intuitively, our construction generates keys using the binary tree structure as follows:

1. We associate a binary string of length at most $\lceil \log n \rceil$ to each vertex in V;
2. We then associate a secret to the root node of $T_n$ from which a secret for each non-root vertex may be derived using standard key derivation methods. The binary string associated to the vertex dictates how the secret is derived;
3. For each security label $l \in L$, we define the key $\kappa_l$ used to protect data objects in the KAS to be the secret assigned to the leaf labeled $\alpha(l)$. To minimize the material issued to users, we issue secrets associated to non-leaf nodes of $T_n$ from which secrets for all descendant nodes can be derived (in particular users can derive all keys for which they are authorized).

**Labeling the tree.** We label the root node of $T_n$ by the empty string $\epsilon$ and, for each node $x \in V$, label the left and right children of $x$ (if they exist) by $x \parallel 0$ and $x \parallel 1$ respectively. Figure 1a gives an example labeling of a tree $T_5$. We may abuse notation by referring to a node of $T_n$ and its associated binary string interchangeably. We denote the set of leaf nodes in $T_n$ by $\overline{V}$.

**Deriving keys.** We now assign a secret to each node. Let $\rho$ be a security parameter and let $F : \{0,1\}^\rho \times \{0,1\}^\star \to \{0,1\}^\rho$ be a Pseudo-Random Function (PRF) which takes a key $k$ and a string $x$ and outputs a pseudo-random string of the same length as the key. We shall write $F_k(x)$ in preference to $F(k, x)$.

[1] All logarithms are base 2 throughout this paper.

The secret $s(\epsilon)$ associated to the root node $\epsilon \in V$ is chosen uniformly at random: $s(\epsilon) \xleftarrow{\$} \{0,1\}^\rho$. For each non-root node $y = x \parallel b$ in $V$, where $x \in V$ and $b \in \{0,1\}$, we compute the secret $s(y) = F_{s(x)}(b)$. If $x$ is a prefix of $y$, then $s(y)$ may be derived from $s(x)$ by iteratively applying $F$ on each remaining bit of $y$ in turn. This is shown in Figure 1b and in GetSec in Figure 1c. For appropriate choices of $F$, it is computationally infeasible to compute $s(x)$ from $s(y)$.

**Assigning keys.** Recall that $\alpha$ is a bijective mapping associating each security label $l \in L$ to a unique leaf node $\alpha(l)$ in $\overline{V}$. For a *set* of security labels $X \subseteq L$, we define $\alpha(X) = \{\alpha(x) : x \in X\}$. Recall also that each object $o \in O$ is associated with a security label $\lambda(o) \in L$. Hence, $\lambda(o)$ is associated with a leaf node $\alpha(\lambda(o)) \in T_n$. We may refer to the secrets associated to leaf nodes in $T_n$ as *keys*; $o$ should thus be encrypted under the key $\kappa_{\lambda(o)} = s(\alpha(\lambda(o)))$.

Each user $u \in U$ is authorized for the security labels $\downarrow\lambda(u) = \{l \in L : l \leqslant \lambda(u)\}$ and hence requires the keys $\{\kappa_x = s(x) : x \in \alpha(\downarrow\lambda(u))\}$. We may reduce the cryptographic material that $u$ must be issued by using non-leaf nodes of $T_n$ to represent multiple elements of $\downarrow\lambda(u)$. If $\alpha(\downarrow\lambda(u))$ contains *all* descendant leaf nodes of a node $x \in V$, we may instead issue the single secret $s(x)$; keys for all descendant leaf nodes can then be efficiently derived. More formally:

**Definition 2.** *Given $X \subseteq \overline{V}$, we define the* minimal cover*, $\lceil X \rceil$, of $X$ to be the* smallest *subset of $V$ such that:*

1. *for every $x \in X$, there exists an ancestor of $x$ in $\lceil X \rceil$;*
2. *for every $y \in \lceil X \rceil$, every $z \in \overline{V}$ that has $y$ as an ancestor belongs to $X$.*

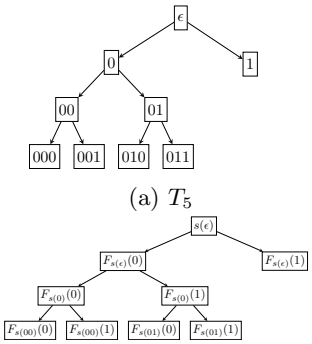
Then, a user issued a set of secrets $\sigma_{\lambda(u)}$ containing $\{s(x) : x \in \lceil \alpha(\downarrow\lambda(u)) \rceil\}$ may derive $\kappa_l = s(\alpha(l))$ if and only if $l \leqslant \lambda(u)$. Condition 1 ensures that a user can derive all keys for which they are authorized (*correctness*), whilst Condition 2 ensures that they cannot derive any other keys (*security*). Since $T_n$ is a full tree (every node has 0 or 2 children), it is easy to see that $\lceil X \rceil$ is unique.

As an example, consider an information flow policy mapped to the tree $T_5$ given in Figure 1a and suppose $\alpha(\downarrow l) = \{010, 011, 1\}$ for some label $l \in L$. Then, $\lceil \alpha(\downarrow l) \rceil = \{01, 1\}$, and $\sigma_l$ contains $F_{F_{s(\epsilon)}(0)}(1)$ and $F_{s(\epsilon)}(1)$.

A simple method to compute $\lceil X \rceil$ for $X \subseteq \overline{V}$ is to observe that a node $x \in V$ is an ancestor of a node $y \in V$ if and only if the binary string associated to $x$ is a prefix of the string associated to $y$. Let us define the *strict prefix* of bit string $b_0 b_1 \dots b_i$ to be $b_0 b_1 \dots b_{i-1}$. Then, if two bit strings in $X$ share a strict prefix, both may be replaced by the strict prefix and the keys for both strings can be computed in a single step. We may continue replacing pairs of bit strings in $X$ (of the same length) with their common strict prefix until no more pairs can be found. With this method, $\lceil X \rceil$ can be computed directly from the set of bit strings $X$ and the set up authority need not store the enforcement structure $T_n$.

### 3.3 Summary and Discussion

Our complete KAS construction is given in Figure 1c. It is easy to see that: 1) no user requires more than $\lceil n/2 \rceil$ secrets; 2) no user requires more than




(a) $T_5$

(b) Secret generation

**Gen**$(1^\rho, (L, \leqslant))$

Let $\alpha : L \to \overline{V}$
$s(\epsilon) \xleftarrow{\$} \{0,1\}^\rho$
$Pub \leftarrow \perp$
**foreach** $l \in L$ :
  $\kappa_l \leftarrow$ GetSec$(\alpha(l), \epsilon, s(\epsilon))$
  $\downarrow(l) \leftarrow \{l' \in L : l' \leqslant l\}$
  **foreach** $x \in \lceil \alpha(\downarrow(l)) \rceil$ :
    $s(x) \leftarrow$ GetSec$(x, \epsilon, s(\epsilon))$
  $\sigma_l \leftarrow \{(x, s(x)) : x \in \lceil \alpha(\downarrow(l)) \rceil\}$
**return** $(\{\kappa_l, \sigma_l\}_{l \in L}, Pub)$

**Derive**$(-, -, \alpha(y), \sigma_x, -)$

**foreach** $(l, s(l)) \in \sigma_x$ :
  **if** $l$ is a prefix of $\alpha(y)$
    **return** GetSec$(\alpha(y), l, s(l))$
**return** $\perp$

GetSec$(a, b, s(b))$

**if** $b$ is not a prefix of $a$
  **return** $\perp$
$z \leftarrow b$
**for** $i = len(b) \ldots len(a) - 1$ :
  $s(z \parallel a_i) = F_{s(z)}(a_i)$
  $z \leftarrow z \parallel a_i$
**return** $s(a)$

(c) Our KAS construction

Fig. 1: Our KAS construction with an example tree $T_5$ and an illustration of secret generation. The inputs to the supporting algorithm GetSec in the KAS are two bit strings $a = a_0 \ldots a_m, b = b_0 \ldots b_n$, where $m, n \in \mathbb{N}$, and a secret $s(b)$.

$\lceil \log n \rceil$ steps to derive a decryption key; and 3) no additional information is required to perform key derivation. In contrast, for an iterative KAS with public information [16]: 1) users require a single secret; 2) derivation may take up to $n$ steps; 3) up to $\mathcal{O}(n^2)$ items of public information may be required. In other words, our scheme has advantages in terms of public information and derivation cost, but users may need to manage additional secrets. A more detailed comparison with related work is given in Section 5.

Derivation in our construction requires knowledge of a binary label $\alpha(y)$ for $y \in L$; hence one may argue that the $\alpha$ mapping should constitute public information. It seems apparent, however, that storing some representation of labels is an inherent requirement of any efficient KAS — data objects must be labeled by their security label to identify the objects to be retrieved from the file-system and the decryption keys to use, whilst secrets must be labeled such that they can be used to derive appropriate decryption keys.[2]

In our scheme, $\sigma_{\lambda(u)}$ contains the appropriate binary labels and we assume that each object $o \in O$ is labeled by $\alpha(\lambda(o))$ *instead* of $\lambda(o)$. (In fact, $\alpha(\lambda(o))$ is a compact way to uniquely represent security labels and may actually decrease storage costs.) Thus, the input to Derive in our KAS includes $\alpha(y)$ instead of $y \in L$, and $\alpha$ need *not* be public. Derive requires *only* the binary string $\alpha(y)$ of the target label $y$ and a suitable secret $\sigma_x$; we omit other unrequired inputs.

To our knowledge, all prior KASs (*including* those without public derivation information) require that users store the enforcement structure for use during

[2] It is unfortunate that existing KAS definitions do not permit consideration of such implementation details. In our case, permitting Gen to take the full policy rather than just $(L, \leqslant)$ could aid defining $\alpha$. Alternatively, the input could be $(\alpha(L), \leqslant)$.

Derive. In schemes that use public information, this is to identify the information needed to derive the next secret in the derivation "path". In schemes based on tree or chain partitions [13–15, 17], the algorithm must know which secret should begin the derivation. In contrast, a nice feature of our scheme with the above method for computing $\lceil \alpha(\downarrow\lambda(u)) \rceil$ is that Derive need only test whether one binary string is a prefix of another. Thus, it is sufficient for users to provide *only* the binary labels $\alpha(\lambda(o))$ and $\lceil \alpha(\downarrow l) \rceil$, which we have already argued represent necessary knowledge for users of any KAS. Furthermore, the steps required to derive a key are immediately apparent from the binary label itself, without requiring user knowledge of $T_n$ or $(L, \leqslant)$. In short, our scheme means that only the administrator need know the actual structure of the security policy. This clearly has practical advantages, but is also useful if policy privacy is required.

*Correctness and Security.* It is easy to see that our KAS is *correct* due to Condition 1 of Definition 2 and the iterative nature of the key generation. The iterative function $s$ computes $s(x)$ from any prefix $y$ of $x$, and Condition 1 of Definition 2 ensures that, for all labels $l \in \downarrow\lambda(u)$, there exists a prefix of $\alpha(l)$ in $\lceil \alpha(\downarrow\lambda(u)) \rceil$.

Our scheme meets the strongest security property currently defined for KASs:

**Theorem 1.** *Let* $F : \{0,1\}^\rho \times \{0,1\}^\star \to \{0,1\}^\rho$ *be a secure pseudo-random function with security parameter* $\rho \in \mathbb{N}$. *Then, for any information flow policy* $P = ((L, \leqslant), U, O, \lambda)$, *the KAS in Figure 1c is* strongly key indistinguishable.

The full version of this paper gives a security proof bounding the advantage of an adversary against our KAS by the (negligible) advantage of a set of distinguishers against $F$.

Our scheme is somewhat unusual in that each label is associated with a single value. All prior schemes, to our knowledge, that achieve key indistinguishability require each label to be associated with a secret *and* a key. In our case, secrets are associated with interior nodes of the tree (which are not associated to a security label), while keys are just secrets associated with leaf nodes; the values issued to users (i.e. secrets $\sigma_{\lambda(u)}$) may, and do, contain keys themselves.

*Related Work.* Our construction is similar to the Goldreich, Goldwasser and Micali (GGM) *puncturable PRF* [19]. In Section 6, we take advantage of the inherent puncturing mechanism to enforce additional policy features such as separation of duty and limited-depth inheritance. The iterative application of a PRF over a tree structure superficially resembles the forward-secure key updating scheme of Backes *et al.* [6] in which all keys are generated independently for the purpose of key refreshing (e.g. for a single label); we define multiple, related security labels and keys. Finally, Blundo *et al.* [8] also considered methods to derive keys using tree structures in the context of access control matrices, showed that finding optimal trees to minimize user secrets is an NP-hard problem and introduced heuristic approaches; our work focuses on the design of KASs for information flow policies and considers different heuristic techniques in Section 4.

## 4 Optimizing the Enforcement Structure and Mapping

We now complete our KAS by considering methods to fine-tune the specific choice of enforcement structure and to choose the mapping from policy poset to enforcement structure. We have seen that our KAS has some advantages over prior KASs but that users may require many secrets in the worst-case. We therefore aim to design methods that, given a policy poset, mitigate this concern and optimize the performance of the resulting KAS. (Prior schemes are limited in this regard as they only consider a trivial mapping and are hence limited to enforcement structures based directly on the poset e.g. Hasse diagrams.)

Recall that each user $u \in U$ is issued a set of secrets $\sigma_{\lambda(u)}$ associated to the minimal cover $\lceil \alpha(\downarrow\lambda(u)) \rceil$ of their authorized set. Thus, whenever $\alpha(\downarrow\lambda(u))$ contains *both* children of a node in $T_n$, the size of $\sigma_{\lambda(u)}$ is reduced by one. To minimize the average size of $\sigma_{\lambda(u)}$ over all users $u \in U$, we therefore aim to define $\alpha$ such that the authorized sets $\alpha(\downarrow\lambda(u))$ contain as many such pairs of child nodes as possible. Of course, every such reduction increases the derivation cost by one but the maximal derivation path remains bounded by $\lceil \log n \rceil$. Figure 2 illustrates the effect of choosing two different $\alpha$ mappings when $n = 5$.

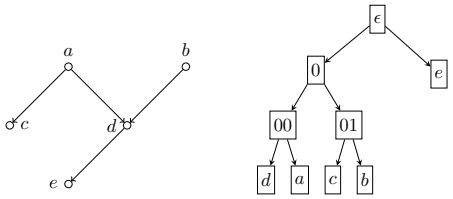
Unfortunately, we conjecture that finding an *optimal* mapping is a hard problem. The number of permissible trees and mappings grows exponentially and it appears difficult to optimally group labels (to share a common prefix in $T_n$) without considering a global view — each choice restricts the possible groupings for other labels and whilst some label groupings would benefit some users, they may lead other users to require a large number of secrets.

Our goal in this section, therefore, is to introduce heuristics to find 'good' $\alpha$ mappings. We first describe our best performing heuristic, based on finding maximal matchings between sets of labels with respect to suitable weightings. We then discuss a considerably cheaper heuristic which, in our experiments, provides reasonable performance.

### 4.1 The FindTree Heuristic

Recall that the size of a binary label represents the depth of the associated node in $T_n$; thus we may fully describe the structure of $T_n$ *and* the assignment of labels to leaves via an $\alpha$ mapping that outputs binary labels of varying sizes. To represent such a mapping, let us define a *partition* to be a recursive data structure with an associated *depth* function $\mathsf{D}$. For each $l \in L$, let $P = [l]$ be a partition (of depth $\mathsf{D}(P) = 0$). For two partitions $P$ and $Q$, let $[P, Q]$ also be a partition of depth $\max(\mathsf{D}(P), \mathsf{D}(Q)) + 1$. Any binary tree $T$ can be represented by a partition e.g. $T_5$ in Figure 2b is represented by [[[[b], [e]], [d]], [[a], [c]]].

Our aim is to find a partition $P$ of depth $\mathsf{D}(P) = \lceil \log n \rceil$ that maximizes the number of shared strict prefixes in the authorized sets of all users. Our approach is to find pairs of labels that most commonly occur *together* in authorized sets, and to which the greatest number of users are assigned; such pairs shall be assigned to sibling leaf nodes in $T_n$. Every time a user is authorized for the pair of labels, they may instead be issued the single secret associated to their parent.




(a) Poset (b) $T_5$ generated by $\alpha_2$

| $l$ | $\downarrow l$ | $\alpha_1(l)$ | $\alpha_2(l)$ | $\lceil \alpha_1(\downarrow l) \rceil$ | $ceil\alpha_2(\downarrow l)$ |
|---|---|---|---|---|---|
| $a$ | $\{a, c, d, e\}$ | 001 | 10 | $\{00, 010, 1\}$ | $\{001, 01, 1\}$ |
| $b$ | $\{b, d, e\}$ | 011 | 000 | $\{011, 000, 1\}$ | $\{0\}$ |
| $c$ | $\{c\}$ | 010 | 11 | $\{010\}$ | $\{11\}$ |
| $d$ | $\{d, e\}$ | 000 | 01 | $\{000, 1\}$ | $\{001, 01\}$ |
| $e$ | $\{e\}$ | 1 | 001 | $\{1\}$ | $\{001\}$ |

Fig. 2: An example showing the effects of two different choices of $\alpha$ mappings. Observe that the average size of $\lceil \alpha_2(\downarrow l) \rceil$ is smaller than that of $\lceil \alpha_1(\downarrow l) \rceil$.

Intuitively, to optimally pair sets of labels, we form a weighted graph where vertices represent partitions of labels and edge weights represent the number of users authorized for *all* labels in the connected partitions. We find a *maximum weight matching* on this graph which selects edges to maximize the associated weights; matched vertices represent partitions that should be grouped as a sub-tree in $T_n$. We iterate this process to form larger groups, beginning with pairs since smaller sets of labels are most likely to occur in multiple authorization sets and hence benefit the most users. Ultimately we create a sequence of nested partitions (of differing sizes) describing which labels should be grouped, and at which level, in $T_n$. Each chosen partition size dictates the structure of $T_n$; the optimal structure is thus derived from the specific policy being enforced.

Our FindTree heuristic is given in Figure 3. Figure 3 illustrates the heuristic on the poset in Figure 2a; the selected maximum weight matchings are illustrated by solid edges. The average number of secrets required is $\frac{6}{5}$ using the mapping found via FindTree compared to $\frac{8}{5}$ when using the $\alpha_2$ mapping from Figure 2b.

FindTree begins by defining a set of vertices $V$ for a graph, where each vertex is a trivial partition $[l]$ for a label $l \in L$. A loop then iteratively groups labels together to form sub-trees in $T_n$. On each iteration, Step 2 forms a graph in which vertices represent previously found partitions and edges represent potential groupings; restrictions on permissible groupings are discussed below. Step 3 assigns a weight to each edge corresponding to the number of users authorized for all labels in the connected partitions: let $U(l) = |\{u \in U : \lambda(u) = l\}|$ be the number of users assigned to a label $l \in L$, and recall the *order filter* $\uparrow l = \{x \in L : x \geqslant l\}$ describes the labels authorized for $l$. For a partition $P$, let $\mathsf{elems}(P)$ denote the set of labels in a partition $P$ e.g. $\mathsf{elems}([[d, b], [a]]) = \{a, b, d\}$ and let $\uparrow P = \bigcap_{l \in \mathsf{elems}(P)} \uparrow l$ be the set of labels in the order filter of *all* labels in

$P \xleftarrow{\$} \mathsf{FindTree}((L, \leqslant), U, \lambda)$:

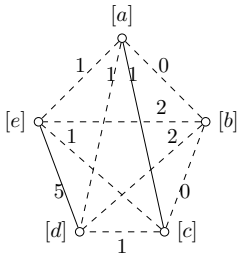
Let $i = 1$. Define $V = \{[l] : l \in L\}$. While $|V| > 2$:

1. If $|V| \leqslant 2^{\lceil \log |L| \rceil - i}$ then increment $i$.
2. Construct the undirected graph $G = (V, E)$ where each vertex is a partition and
$$E = \{PQ : P, Q \in V, P \neq Q, \mathsf{D}(P), \mathsf{D}(Q) \leqslant i - 1\}.$$
3. For each edge $PQ \in E$, define the weight $w(PQ) = \sum_{z \in (\uparrow P \cap \uparrow Q)} U(z)$ to be the number of users authorized for all labels in the partitions $P$ and $Q$.
4. Find a maximum weight matching $M$ of $G$.
5. Define a new set of vertices $V' = \{[P, Q] : PQ \in M\}$, where each vertex is a new partition comprising two partitions that were paired in the maximal matching.
6. For any unmatched vertices (i.e. vertices $X \in V$ such that no edge in $M$ includes $X$), add $X$ to $V'$.
7. Redefine the vertex set $V = V'$ and go to next iteration.

If $|V| = 1$, return $V$, else return the partition $[V[0], V[1]]$.

| $v \in V$ | $\uparrow v$ | $U(v)$ |
|---|---|---|
| $[a]$ | $\{a\}$ | 1 |
| $[b]$ | $\{b\}$ | 2 |
| $[c]$ | $\{a, c\}$ | 3 |
| $[d]$ | $\{a, b, d\}$ | 2 |
| $[e]$ | $\{a, b, d, e\}$ | 1 |

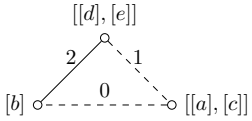
(a) Initial vertices and user assignments



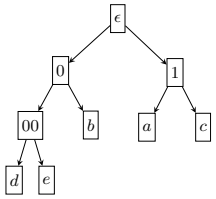

(b) First matching

| $v$ | $\uparrow v$ |
|---|---|
| $[[d], [e]]$ | $\{a, b, d\}$ |
| $[[a], [c]]$ | $\{a\}$ |
| $[b]$ | $\{b\}$ |

(c) Vertices formed from first matching




(d) Second matching




(e) Final partition $[[[[d], [e]], [b]], [[a], [c]]]$

| $l$ | $\alpha(l)$ | $\lceil \downarrow \alpha(l) \rceil$ |
|---|---|---|
| $a$ | 10 | $\{00, 1\}$ |
| $b$ | 01 | $\{0\}$ |
| $c$ | 11 | $\{11\}$ |
| $d$ | 000 | $\{00\}$ |
| $e$ | 001 | $\{001\}$ |

(f) Resulting mapping $\alpha$ and minimal covers

Fig. 3: The FindTree heuristic to find a suitable binary tree partition and example application on the poset in Figure 2a with user assignments shown in Table 3a.

$P$. Then the weight assigned to an edge connecting $P$ and $Q$ is the sum of $U(z)$ for $z \in \uparrow P \cap \uparrow Q$ i.e. the number of users authorized for *all* labels in $P$ and $Q$.

Step 4 applies a *maximum weight matching* algorithm which selects a set of non-adjacent edges from $G$ with the greatest total weight (i.e. the groupings

that benefit the *most* users). Step 5 forms a set of vertices to create the graph for the next iteration; each vertex is a partition formed from a pair of partitions matched in Step 4. Step 6 also defines vertices for partitions left unmatched in Step 4 such that later iterations may consider them to form a sub-tree containing triples of labels. The process is repeated until a single partition remains; to ensure termination, we assume that maximal matchings contain at least one edge.

We maintain a counter $i$ representing the *level* of $T_n$ at which sub-trees induced by the current partition matchings shall be rooted. The *level* of the root node is equal to the depth of the tree and the level of the lowest leaf node is 0. To ensure that the tree has depth $\lceil \log n \rceil$, we only add an edge in Step 2 between partitions $P$ and $Q$ if the depth of $P$ and $Q$ does not exceed $i-1$; thus, when $i = 1$, we only pair singleton labels, and when $i = \lceil \log n \rceil$, we only pair partitions of depth at most $\lceil \log n \rceil - 1$. In Step 1 we also check that the number of partitions remaining is at most $2^{\lceil \log n \rceil - i}$ before incrementing $i$ to ensure that enough groupings are performed at each level for the final tree to be binary.

If one stores $\uparrow v$ and $\mathsf{D}(v)$ for each $v \in V$, we may construct each weighted graph $G$ in $\mathcal{O}(n^3)$ time. Finding the maximum weight matching requires $\mathcal{O}(n^3)$ time [18]. Since we iterate $\mathcal{O}(n)$ times, our heuristic requires $\mathcal{O}(n^4)$ time.

### 4.2 The Order Filter Sort Heuristic

FindTree is our best-performing heuristic. From experimental evaluation, however, we observe that when there is a choice of tree (i.e. when $|L|$ is not $2^x$ or $2^x - 1$ for some $x$), FindTree chooses a structure (isomorphic to) a *left-balanced* tree approximately half the time. (A left-balanced, or complete, tree has all levels completely filled except possibly the last, and the leaves are as far left as possible.) In the full version of this paper, we show that amending FindTree to *only* map labels to a *fixed* left-balanced tree structure does not significantly degrade the heuristic's performance but reduces the run-time to $\mathcal{O}(n^3 \log n)$. We conjecture that the maximal weight matching algorithm chooses as many pairs as possible during the first iteration causing most tuples to comprise pairs and making it likely that the resulting tree structure resembles a left-balanced tree.

However, if one is willing to fix the tree-structure to be left-balanced, a very cheap heuristic is to simply sort labels by the size of their order filters $\uparrow l$ in decreasing order, and to map labels to leaf nodes from left to right. Intuitively one hopes that by pairing labels with large order filters, the order filters are likely to intersect. Users authorized for a label within the intersection require at least one fewer secret. This heuristic requires $\mathcal{O}(n \log n)$ time, and we shall see in Section 5 that it performs remarkably well in practice. Unlike FindTree, this heuristic does not consider the number of users assigned to labels. We therefore expect FindTree to be more optimal in general, although we may hope that many realistic policies may have many users assigned to 'low' labels (with large order filters) which would favor this cheaper heuristic.

| **Scheme** | Max. Keys **K** | Public Info **P** | Derivations **D** |
|---|---|---|---|
| Trivial [16] | $\mathcal{O}(n)$ | 0 | 0 |
| Iterative [2, 16] | 1 | $\lvert E^\star \rvert$ | $\mathcal{O}(n)$ |
| Direct [2, 16] | 1 | $\lvert E \rvert$ | 1 |
| Tree [15] | $\mathcal{O}(\ell)$ | 0 | $\mathcal{O}(n)$ |
| Chain [14] | $\mathcal{O}(w)$ | 0 | $\mathcal{O}(n)$ |
| Our Scheme | $\mathcal{O}(\lceil \frac{n}{2} \rceil)$ | 0 | $\mathcal{O}(\lceil \log(n) \rceil)$ |

Table 1: Comparison of different KASs. $|E^\star|$ and $|E|$ represent the number of edges in the Hasse diagram $H(L, \leqslant)$ and its transitive closure, respectively.

## 5 Evaluation

We now compare our scheme to prior KASs with respect to the following parameters: **K** is the maximum number of keys/secrets a user must be issued, **P** is the amount of public derivation information, and **D** is the maximum number of derivation steps required. The discussion is summarized in Table 1.

Many schemes issue users a single key ($\mathbf{K} = 1$) and enable iterative derivation along paths in the enforcement structure using public information. In many schemes [16, 21], the enforcement structure is simply the Hasse diagram of $(L, \leqslant)$, in which case $\mathbf{P} = \mathcal{O}(n^2)$ and $\mathbf{D} = \mathcal{O}(n)$. An alternative is to define a directed graph where $xy$ is an edge if and only if $y < x$, in which case $\mathbf{P} = \mathcal{O}(n^2)$ and $\mathbf{D} = 1$. The 'trivial' KAS supplies users with the keys associated to all $y \leqslant \lambda(u)$; hence $\mathbf{K} = \mathcal{O}(n)$, $\mathbf{P} = 0$ and $\mathbf{D} = 0$. Recent schemes remove public information by forming a sub-graph of the Hasse diagram which is either a tree [15] or a chain partition [14, 13, 17]. In these schemes, $\mathbf{P} = 0$, while $\mathbf{D} = \mathcal{O}(n)$ (or, more precisely, the depth of the poset) but users may require several keys: for schemes based on chain partitions, $\mathbf{K} = w$ where $w$ is the width of $(L, \leqslant)$; in schemes based on tree partitions, $\mathbf{K} = \ell$ keys, where $\ell \geqslant w$ is the number of leaves in the tree. Recall that in our scheme: $\mathbf{K} = \lceil n/2 \rceil$ keys; $\mathbf{P} = 0$; and $\mathbf{D} = \mathcal{O}(\lceil \log n \rceil)$.

We now present an experimental evaluation showing the performance of these KASs *in practice* in the worst- and average-cases. For each value of $|L|$, we average the results on 30 posets generated randomly by choosing a 'connection probability' $p$ for each node $x$ uniformly at random; for each other node $y$, a covering relation $y \lessdot x$ is added to the poset with probability $p$. The number of users assigned to each label is chosen randomly between 0 and 100. For a fair comparison, we aim to evaluate the KASs on a variety of posets and have not aimed towards any particular policy. A priority for future work is to evaluate KASs on specific real-world policies of interest; unfortunately we have thus far been unable to find real examples of interesting sizes. Most KAS literature does not provide experimental evaluations; ours is certainly the first to compare the efficiency of chain and tree-based KASs, which may be of independent interest.

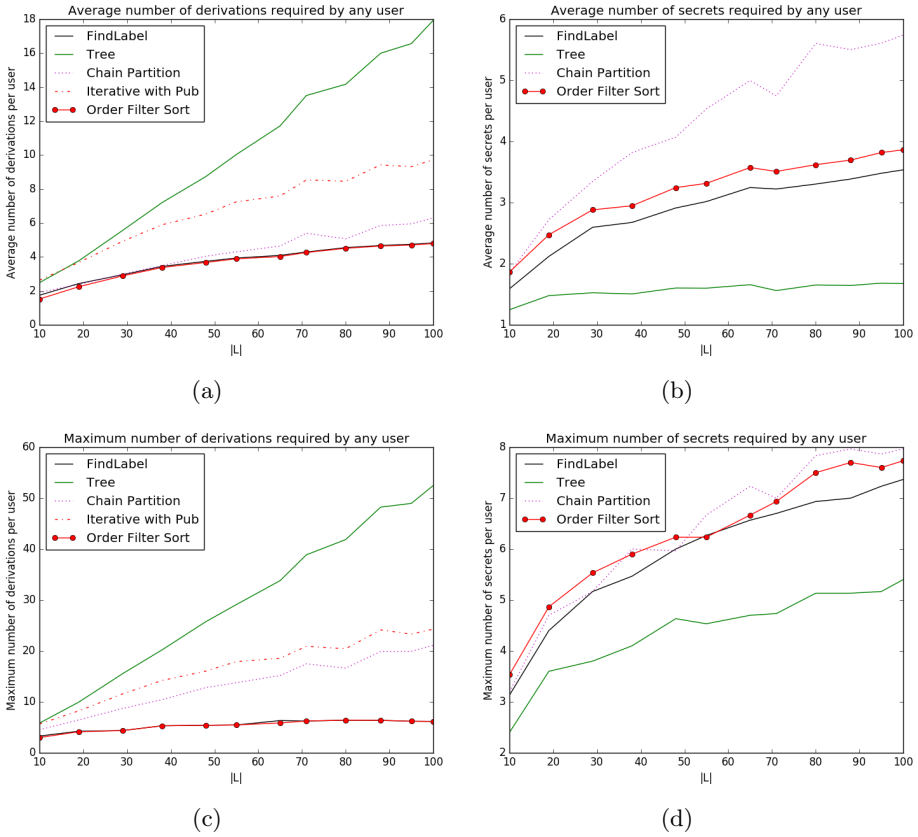



Fig. 4: Experimental evaluation

We compare an *Iterative* scheme that uses public derivation information (the extended scheme by Atallah *et al.* [2] instantiated on the Hasse Diagram of the poset), *Chain-* [14] and *Tree*-based Schemes [15] (which do not require public information), and our KAS using both the FindTree and the order filter-based heuristics. Figures 4a and 4c show the average and maximum number of derivation steps required to compute any key. Derivation steps are considered to be PRF evaluations (the iterative scheme [2] also requires a number of decryptions which are not counted). Figures 4b and 4d show the average and maximum number of secrets (or keys) required by any user in each scheme. The iterative scheme is omitted for clarity, since each user requires one secret.

Recall that the design goals of this example were to bound derivation costs whilst eliminating public information, and it can be seen that this is achieved. Our scheme outperforms all other KASs in terms of derivation costs in these tests. In particular, our logarithmic growth contrasts with the linear cost of tree-based schemes and, particularly in the worst-case, can become rather high.

Furthermore, recall that the storage costs are further reduced in our scheme compared to other KASs since users need not store the enforcement structure. With regards to the number of secrets a user requires (which was *not* one of our primary design goals), our KAS outperforms chain-based schemes but does not quite match tree-based schemes. However, in concrete terms, the actual number of secrets required does not vary greatly between any scheme. Importantly, in these experiments, our theoretical worst-case bound of $\lceil n/2 \rceil$ is *not* met. Whilst it remains possible to obtain this bound (e.g. if the poset is highly symmetrical with equal user assignments over all labels), we expect that such policies may be rather unlikely and that our heuristics will mitigate the concern in practice. Remarkably, the heuristic based on order-filters (with runtime $\mathcal{O}(n \log n)$) performs comparably to FindTree heuristic (with runtime $\mathcal{O}(n^4)$).

Ultimately, the best choice of KAS will always depend on the requirements of the specific application setting and on the policy being enforced. Our scheme appears to be a good well-rounded candidate and may be the best choice if derivation costs or storage requirements are a concern. Our scheme out-performs chain-based schemes in terms of both derivation costs and the number of user secrets required. Furthermore, the analysis required to find an optimal chain-partition requires $\mathcal{O}(n^4 w)$ time, where $w$ is the width of the poset [14], whilst our cheapest heuristic requires just $\mathcal{O}(n \log n)$. Thus, in many settings, our scheme may be preferable over chain-based schemes.

## 6 Flexible Access Management

In this section, we summarize some additional features enabled by our KAS; the full version of this paper will also introduce a general policy representation (subsuming information flow, temporal and role-based policies) and an associated KAS allowing flexible grouping of access rights.

Prior KASs require *all* keys, secrets and derivation information to be defined and assigned during Gen which may be inefficient when policies define a large number of labels, some of which may never actually be assigned or used. In particular, some policies define a set of primitive labels (e.g. roles, attributes or time periods) and must include security labels for all combinations that *may* be assigned during the system lifetime (e.g. role-based policies define $2^R$ labels for $R$ roles [11]). In contrast, using our KAS, one can define $T_n$ for $n$ primitive labels and define a *single* secret (for the root node of $T_n$) during Gen. Instead of defining additional labels for each potential combination, one can dynamically issue secrets corresponding to the minimal cover of a required set of primitives as required — one can dynamically form new 'labels' that cover the required access rights as users join the system. Our mechanism is similar to the GGM puncturable PRF [19] and this can be viewed as utilizing the puncturing mechanism to define access rights. A puncturable PRF issues keys restricting the pseudo-random outputs that may be computed, which is precisely the goal of a KI-secure KAS. This puncturing technique enables useful features such as:

*Limited Depth Inheritance* is an important component of hierarchical access policies to prevent senior users aggregating excessive access rights [2, 10, 20]. Encoding such restrictions directly into the poset may increase the number of labels and derivation paths (and hence the amount of public information) or increase the width of the poset (and hence the number of secrets users must hold [14, 15]). To our knowledge, the only KAS that directly allows limited depth inheritance [2] requires public information and, crucially, is not collusion resistant (and hence not KI-secure). In contrast, our KAS *can* enable limited depth inheritance to be efficiently implemented. Intuitively, we wish to change the authorized set of a user from $\downarrow u = \{y \in L : y \leqslant \lambda(u)\}$ to $\downarrow u_l = \{y \in L : y \leqslant \lambda(u), y \not< l\}$ where $l$ is a threshold label beyond which derivation should be prevented. Clearly, it is rather difficult to terminate derivation in typical iterative KASs where the key for $l \in L$ is determined by the secrets of labels $l' > l$. In our KAS, on the other hand, secrets correspond to interior nodes of $T_n$ which are *not* associated to security labels. Thus, one can simply issue the minimal cover $\lceil \alpha(\downarrow u_l) \rceil = \lceil \{\alpha(l') : l' \in \downarrow u_l\} \rceil$ and ignore any labels below the threshold when selecting the set of secrets.

*Separation of Duty* policies form an important business practice which compartmentalize objects and users to avoid conflicts of interests. In essence, users assigned a label $l$ should no longer inherit the access rights of a set of labels $X \subseteq L$ which, again, often requires complex and costly modifications to the poset. Using our KAS, one may simply issue $\lceil \alpha(\downarrow u \setminus X) \rceil = \lceil \{\alpha(l) : l \in \downarrow u \setminus X\} \rceil$.

*Interval-based Policies* such as temporal or geo-spatial policies [3, 12] can be handled in the same way. Consider a temporal policy where $L$ is a set of time periods $[0, n]$ and users are authorized for time intervals $[a, b)$ for $0 \leqslant a, b < n$. Prior KASs require a label for each possible interval. Using our KAS, we may instead define $L$ to be simply $[0, n]$ and issue precisely the secrets corresponding to $\lceil \{\alpha(x) : x \in [a, b)]\} \rceil$. Intuitively, one may think of $L$ as a total order and use the limited depth inheritance constraint to restrict derivation from $a$ down to $b$.

## 7 Conclusion

We have introduced a novel approach to designing KASs by mapping policies to enforcement structures which need not be derived directly from the policy poset. We have given an example of a very simple KAS based on a binary tree and introduced heuristics to optimally map the policy to a tree. We have shown that our KAS performs favorably to prior schemes, and reduces the storage requirements of user devices and logarithmically bounds derivation costs.

It is also important to consider how keys and secrets can be updated. KASs with public information [2, 16] may amend a portion of that information to define new secrets using the same derivation mechanism, but prior work [14, 15] has not considered how to perform updates without public information. A natural solution is to include counters in the PRF inputs when deriving keys; each derivation step may have a 'version' indicated by the counter. Derivation costs will not increase but users must learn current counter values in some way. Investigating such methods and their associated costs will be a priority for future

work. We would also like to use our experimental implementations to perform a thorough comparison of the relative costs and strengths of KASs compared to public key schemes e.g. Attribute-based Encryption.

We hope that future work will also consider enforcement structures to target different design goals of KASs and develop interesting optimization strategies for the mappings e.g. one could generalize our construction to $n$-ary trees or trees with varying degrees. Finally, we hope that our work spurs the development of efficient constrained PRFs tailored to enforcing access control policies.

## References


1. S. G. Akl and P. D. Taylor. Cryptographic Solution to a Problem of Access Control in a Hierarchy. *ACM Trans. Comput. Syst.*, 1(3):239–248, 1983.
2. M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken. Dynamic and Efficient Key Management for Access Hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.
3. M. J. Atallah, M. Blanton, and K. B. Frikken. Efficient techniques for realizing geo-spatial access control. In F. Bao and S. Miller, editors, *ASIACCS*, pages 82–92. ACM, 2007.
4. M. J. Atallah, M. Blanton, and K. B. Frikken. Incorporating Temporal Capabilities in Existing Key Management Schemes. In J. Biskup and J. Lopez, editors, *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2007.
5. G. Ateniese, A. D. Santis, A. L. Ferrara, and B. Masucci. Provably-Secure Time-Bound Hierarchical Key Assignment Schemes. *J. Cryptology*, 25(2):243–270, 2012.
6. M. Backes, C. Cachin, and A. Oprea. Secure Key-Updating for Lazy Revocation. In *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 2006.
7. D. E. Bell and L. J. LaPadula. Computer security model: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., 1975.
8. C. Blundo, S. Cimato, S. D. C. di Vimercati, A. D. Santis, S. Foresti, S. Paraboschi, and P. Samarati. Managing key hierarchies for access control enforcement: Heuristic approaches. *Computers & Security*, 29(5):533–547, 2010.
9. A. Castiglione, A. D. Santis, B. Masucci, F. Palmieri, A. Castiglione, J. Li, and X. Huang. Hierarchical and Shared Access Control. *IEEE Trans. Information Forensics and Security*, 11(4):850–865, 2016.
10. J. Crampton. On permissions, inheritance and role hierarchies. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 85–92. ACM, 2003.
11. J. Crampton. Cryptographic Enforcement of Role-Based Access Control. In *Formal Aspects in Security and Trust*, volume 6561 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.
12. J. Crampton. Practical and efficient cryptographic enforcement of interval-based access control policies. *ACM Trans. Inf. Syst. Secur.*, 14(1):14, 2011.
13. J. Crampton, R. Daud, and K. M. Martin. Constructing Key Assignment Schemes from Chain Partitions. In S. Foresti and S. Jajodia, editors, *Data and Applications Security and Privacy XXIV, 24th Annual IFIP WG 11.3 Working Conference, Rome, Italy, June 21-23, 2010. Proceedings*, volume 6166 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2010.

14. J. Crampton, N. Farley, G. Gutin, and M. Jones. Optimal Constructions for Chain-Based Cryptographic Enforcement of Information Flow Policies. In *DBSec*, volume 9149 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2015.
15. J. Crampton, N. Farley, G. Gutin, M. Jones, and B. Poettering. Cryptographic Enforcement of Information Flow Policies Without Public Information. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, pages 389–408, 2015.
16. J. Crampton, K. M. Martin, and P. R. Wild. On Key Assignment for Hierarchical Access Control. In *CSFW*, pages 98–111. IEEE Computer Society, 2006.
17. E. S. V. Freire, K. G. Paterson, and B. Poettering. Simple, Efficient and Strongly KI-Secure Hierarchical Key Assignment Schemes. In *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 101–114. Springer, 2013.
18. Z. Galil. Efficient Algorithms for Finding Maximum Matching in Graphs. *ACM Comput. Surv.*, 18(1):23–38, Mar. 1986.
19. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
20. R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The NIST model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63, 2000.
21. A. D. Santis, A. L. Ferrara, and B. Masucci. Efficient Provably-Secure Hierarchical Key Assignment Schemes. In L. Kucera and A. Kucera, editors, *Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*, volume 4708 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2007.
22. A. D. Santis, A. L. Ferrara, and B. Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. In V. Lotz and B. M. Thuraisingham, editors, *SACMAT 2007, 12th ACM Symposium on Access Control Models and Technologies, Sophia Antipolis, France, June 20-22, 2007, Proceedings*, pages 133–138. ACM, 2007.