Lazy Updates in Key Assignment Schemes for Hierarchical Access Control

Jason Crampton

Technical Report
RHUL–MA–2006–11
5 December 2006

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

Abstract

Hierarchical access control policies are used to restrict access to objects by users based on their respective security labels. There are many *key assignment schemes* in the literature for implementing such policies using cryptographic mechanisms. Updating keys in such schemes has always been problematic, not least because many objects may be encrypted with the same key. We propose a number of techniques by which this process can be improved, making use of the idea of *lazy key updates*, which have been studied in the context of cryptographic file systems. We demonstrate in passing that schemes for lazy key updates can be regarded as simple instances of key assignment schemes. Finally, we illustrate the utility of our techniques by applying them to hierarchical file systems and to temporal access control policies.

1 Introduction

Background An information flow policy associates each user and each object in a computer system with a security label, where the set of security labels L is partially ordered, and requires that a user must be at least as privileged as any object she reads. This is widely known as the *simple security property*, which forms part of the Bell-LaPadula security model [5]. Algebraically, the simple security property is expressed in the following way: for subject s to obtain read access to object o we require that $\lambda(s) \geqslant \lambda(o)$, where λ is a security function that associates an entity with a security label. The enforcement of the simple security property is often referred to as *hierarchical access control* because the partially ordered set of security labels can be represented as a hierarchy.

Although information flow policies were originally used in military systems, they have many other practical applications, particularly if we regard the set of users as being partitioned into a set of *security classes*, with each class being associated with a particular security label. Such applications exist in many different environments, including secure databases, broadcast services (such as Pay TV) and secure mail systems.

Key assignment schemes The use of encryption to enforce information flow policies has been extensively studied; a recent survey paper identified and categorized over 30 schemes in the literature [11]. In such schemes, each security label is associated with a unique cryptographic key: each object is encrypted with the key associated with the object's security label, and each user is given the key associated with her security label. Moreover, each key can be used to derive the keys associated with lower security labels, meaning that a user is required to store a single key. The problem of enforcing the information flow policy thus becomes a problem of assigning keys in a way that is compatible with the policy. For this reason such schemes are typically referred to as *key assignment schemes*. We provide an overview of such schemes in Section 2.

Key updates A number of extensions have been proposed for key assignment schemes. Perhaps the most common in the literature is the use of temporal constraints on the use

of keys [3, 7, 9, 17, 23, 25]. In such schemes, keys are updated periodically and users are able to derive the keys for certain authorized time intervals.

More importantly, it is necessary to update a key assignment scheme in at least two other situations: when a key is compromised and when the association of a user with a security label needs to be revoked. In the latter case, a user may be assigned to a lower security level or an incomparable security level.[1] There are three possible events that require key updates.

- The key for level $l \in L$ is compromised; in this case, all keys for labels less than or equal to l need to be updated.
- A user assigned to level l is removed from the scheme; in this case, all keys for labels less than or equal to l need to be updated.
- A user assigned to level l is assigned to a new security level l'; in this case, all keys for labels less than or equal to l and that are not less than or equal to l' need to be updated.

Collectively we refer to these events as *update events*; the (*label*) *update set* is the set of labels for which new keys are required.

One possibility for key updates is that following an update event, for all l in the update set, and for all u such that $\lambda(u) = l$, we immediately re-encrypt every object with security label l and send u a copy of the new encryption key. An alternative is to postpone these actions until it is absolutely necessary.

We have to assume that if a key κ requires updating, then any objects encrypted with κ are available to any user who could derive κ. Hence, we may as well wait until the contents of an object changes before re-encrypting it. Similarly, we may as well defer sending a user u the new key κ' until such time as u actually requires κ' to decrypt an object. This is sometimes referred to as *lazy update* and *lazy re-encryption*.

One technique for implementing lazy update is *key regression* [13]. Key regression has been used in cryptographic file systems with Unix-like access control, in which each file is associated with a group of users and encrypted with a group key; the group key needs to updated when a user leaves the group. We believe that lazy update is even more important in key assignment schemes, because more than one key may need to be changed following an update event, and hence the problem of re-encrypting objects is exacerbated.

Contributions The first contribution of this paper is to demonstrate that key regression schemes, and other techniques for supporting lazy update, are instances of key assignment schemes (usually for very simple posets). We discuss key regression schemes and their representation as key assignment schemes in Section 3.

The major contribution of this paper is to use ideas from schemes for lazy update and from key assignment schemes to design several generic key assignment schemes that support lazy update. This is the first such work in this area. We describe these schemes in Section 4.

[1] We ignore the case when a user is assigned to a higher security level, because in this case, he can simply be given the relevant key, which will enable him to recover all the keys to which he previously had access.

In Section 5 we illustrate the utility of the schemes in Section 4 by demonstrating their application to cryptographic file systems and to temporal hierarchical access control. Finally, in Section 6 we summarize our contributions and discuss the numerous opportunities for future work.

Notation Henceforth we adopt the following conventions: given a poset L and $x, y \in L$, $\downarrow x$ denotes the set $\{y \in L, y \leqslant x\}$; we may write $y < x$ if $y \leqslant x$ and $y \neq x$; if $y \leqslant x$, we may write $x \geqslant y$; $y \lessdot x$ means that there does not exist $z \in L$ such that $y < z < x$; we write $[m]$ to denote the set $\{0, \ldots, m\}$; x, when used as input to some (cryptographic) function, will denote a string identifying $x \in L$; E denotes a symmetric encryption algorithm, $E_k(p)$ denotes the encryption of plaintext p with key k; $\oplus$ denotes the bitwise XOR operation; $s \parallel t$ denotes the concatenation of strings s and t; $m \mid n$ means that integer m divides integer n without remainder; (m, n) denotes the greatest common divisor of integers m and n; in particular, $(m, n) = 1$ means that m and n are co-prime.

2 Key assignment schemes

A key assignment scheme is administered by the policy owner, often referred to as a *trusted center*, who is responsible for generating and distributing keys, generating and managing any public data required by the scheme and updating the scheme in the event of changes to the policy. In addition to a key $\kappa(x)$ for each security label x, a key assignment scheme may also associate a secret value $\sigma(x)$ with x and publish some information that is accessible to all users of the scheme. This information can be used to derive the encryption key for any lower security label. In other words, an object with security level y that has been encrypted with the associated key $\kappa(y)$ can be decrypted by any user with security level $x \geqslant y$ because the user can derive key $\kappa(y)$ from $\kappa(x)$, $\sigma(x)$, and the public information.

We assume the existence of an RSA key generator, a randomized algorithm that takes a security parameter k as input and outputs a triple (n, e, d) such that: $n = pq$, where p and q are distinct odd primes; $e \in \mathbb{Z}^*_{\phi(n)}$, where $\phi(n) = (p-1)(q-1)$, $e > 1$, and $(e, \phi(n)) = 1$; $d \in \mathbb{Z}^*_{\phi(n)}$, where $ed \equiv 1 \mod \phi(n)$. We also assume the existence of an associated hash function, which maps elements of $\mathbb{Z}^*_n$ (in some standard representation) to bit-strings of some desired length.

KASs require each user to store a key and (optionally) some secret information. They also require storage for the public information. The other distinguishing feature of KASs is whether $\kappa(y)$ can be derived directly from $\kappa(x)$, $y \leqslant x$, or whether it is an iterative process.

In their survey paper, Crampton *et al* identify five different generic schemes, each of which differ in terms of the amount of storage required by the user, storage required for public information, and key derivation method [11]. Of these, two were identified as having the best combination of characteristics. We discuss these schemes in the next two sections.

2.1 Iterative key encrypting key assignment schemes

An *iterative key encrypting key assignment scheme* (IKEKAS), uses the transitivity of the partial order relation and "edge encryption". The trusted center may choose any keys from the key space and then defines $Pub = \{E_{\kappa(x)}(\kappa(y)) : y \lessdot x\}$.

A user with security label x can derive $\kappa(y)$ by selecting a path from x to y in the *Hasse diagram*[2] of L and successively decrypting keys for labels on that path, until $\kappa(y)$ is recovered. The advantage of an IKEKAS is that the user is only required to store a key. The disadvantage is that key derivation is necessarily iterative. There are a number of IKEKASs in the literature [8, 10, 19, 20, 26, 27], and provably secure examples of IKEKASs have recently been constructed by Atallah *et al* [2] and by Ateniese *et al* [3]. An example of an IKEKAS for a simple poset is given in Appendix A.1.

2.2 The Akl-Taylor scheme

The second generic scheme is a *node-based key assignment scheme* (NBKAS), of which the best known, simplest, and most effective is the Akl-Taylor scheme [1]. Key derivation in a NBKAS is direct and the user is only required to store a single key. The disadvantage of a NBKAS is that key updates may be more disruptive than in an IKEKAS, because keys are defined in terms of public information.

The earliest KAS (of any type) is due to Akl and Taylor. The trusted center:

- runs the RSA key generator to obtain (n, e, d), of which only n is used;
- chooses s at random from $\mathbb{Z}_n^*$;
- chooses a mapping $\phi : L \to \mathbb{N}^*$ such that $\phi(x) \mid \phi(y)$ if and only if $y \leqslant x$;
- defines $\kappa(x) = s^{\phi(x)} \mod n$;
- publishes $\{n\} \cup (\phi(x) : x \in L)$.

A user with security label x can derive $\kappa(y)$, $y \leqslant x$ by computing

$$\big(\kappa(x)\big)^{\frac{\phi(y)}{\phi(x)}} = \big(s^{\phi(x)}\big)^{\frac{\phi(y)}{\phi(x)}} = s^{\phi(y)} = \kappa(y).$$

An example of a NBKAS for a simple poset is given in Appendix A.2.

2.3 Key updates in KASs

We have already noted that an update to $\kappa(x)$ generally requires $\kappa(y)$ to be updated for all $y \leqslant x$. Crampton *et al* note that none of the five generic schemes they identify (or the 30+ concrete instances of those generic schemes) enjoys any particular advantage with respect to updating keys [11].

However, as in cryptographic file systems, many objects may be encrypted with the same key. Therefore, we believe that a fruitful line of research is to construct generic KASs that support lazy updates. We consider this problem in Section 4; first we discuss lazy updates in cryptographic file systems.

[2]The Hasse diagram of L is the directed graph of the transitive, reflexive closure of the partial order relation. In other words, it is the graph $(L, \lessdot)$.

3 Key regression and key assignment schemes

Key regression schemes have been used to support the revocation of keys shared by users of a Unix-like file system in which files are encrypted [12]. If a user belonging to a particular group leaves, then it is necessary to update the key(s) associated with members of that group. Note that access to files is determined by the usual Unix permissions (not by a hierarchical access control policy). The cryptographic element is introduced so that the files may be stored on an untrusted server.

A key regression scheme (KRS) generates a sequence of keys that will be used to encrypt the file, with the property that it is not possible to derive future keys from either the current key or previous ones.[3] However, it is possible to derive all previous keys from the current one. The file attributes contain a flag, which indicates whether a key update event has occurred. The next time the file is written, it is re-encrypted with the new key, and when a user wishes to access the file, a copy of the new key is made available. Similar techniques have been explored by Backes *et al* [4].

We now show that we can model a KRS as a KAS. We describe **KR-RSA**, the preferred KRS of Fu *et al*, using our generic framework.

We use the poset $(\mathbb{N}, \leqslant)$, where $\mathbb{N}$ denotes the set of natural numbers with the usual ordering. We write κ_i to denote the key associated with $i \in \mathbb{N}$ and s_i to denote the secret associated with $i \in \mathbb{N}$. The intuition is that κ_i is the key used to encrypt the file during the period between ith update event and the $(i+1)$th update event.[4] The trusted center

- runs the RSA key generator to produce (n, e, d);
- chooses s at random from $\mathbb{Z}_n^*$;
- defines $\sigma_0 = s$, $\sigma_i = \sigma_{i-1}^d \mod n$;
- defines $\kappa_i = h(\sigma_i)$, where h is a suitable hash function;
- publishes $\{n, e\}$.

To compute κ_j from κ_i, $j < i$, the user successively computes $\sigma_{i-1}, \ldots, \sigma_j$ using the fact that $\sigma_{t-1} = \sigma_t^e$ (since $\sigma_t^e = (\sigma_{t-1}^d)^e \mod n = \sigma_{t-1}$), and finally computes $h(\sigma_j)$ to obtain κ_j. In other words, this is simply a variant of an IKEKAS, albeit with very modest public storage requirements.

It is worth noting that the hash function is only required to defeat certain attack models. We could simply define $\kappa_i = \kappa_{i-1}^d \mod n$, thereby obtaining a true IKEKAS. It would not be feasible for a user with κ_i to recover κ_j, $j > i$, because this would imply that there existed a feasible method for solving the RSA problem.[5]

[3] Such a scheme is rather similar to one-time password schemes, which generate passwords by successively hashing a random seed and use the resulting passwords in reverse order of their generation [14].

[4] Fu *et al* present three concrete schemes, two of which are essentially KASs that limit the number of update events. Such schemes can be modeled as KASs using the poset $([m], \leqslant)$, for some $m \geqslant 1$.

[5] Fu *et al* included σ_i and use h to derive keys from σ_i, because the resulting keys are actually encrypted with users' public keys and they wanted the whole scheme to have the property of key indistinguishability.

4 Key assignment schemes for lazy updates

In this section we define a number of KASs that support lazy updates. In other words, different objects with the same security label may have been encrypted with different keys. A user u must be able to derive all the encryption keys in use for all labels $l \leqslant \lambda(u)$.

We assume there are a number of update events indexed by the natural numbers, and that the ith key is used to re-encrypt objects between the ith and $(i+1)$th update event; we write $\kappa_i(x)$ to denote the ith key for security label x.[6] In other words, a user u in time interval i can derive any key in the set $\{\kappa_j(l) : l \leqslant \lambda(u), 0 \leqslant j \leqslant i\}$.

4.1 Naïve approach

The simplest way of constructing a scheme in which keys are refreshed periodically (for example, after an update event), is to construct a KAS for the poset $L \times [m]$, where m denotes the maximum number of update events and $(y, j) \leqslant (x, i)$ if and only if $y \leqslant x$ and $j \leqslant i$. Obviously, m must be chosen in advance by the trusted center.

One disadvantage of this approach is that the resulting poset $L \times [m]$ (whose cardinality is $(m+1)|L|$) is likely to be very large. A large number of keys will need to be generated and a large amount of public information, proportional to the number of edges in the Hasse diagram of L, must be maintained.

4.2 NBKAS

The Akl-Taylor scheme is often criticized because it is claimed that it is difficult to change a key if it is compromised (see [2, 15], for example). It is certainly true that it is easier to change a key in an IKEKAS, because the keys are chosen independently of the public information. However, it is usually the case that a set of keys needs to be changed following an update event, and the Akl-Taylor scheme is generally no more difficult than an IKEKAS to update [11]. Moreover, the Akl-Taylor scheme has one distinct advantage, which is that it is possible to update all the keys, simply by choosing a new system secret; no public information needs to change, unlike any IKEKAS or direct key encrypting key assignment scheme [11].

Using this observation, we propose a scheme in which all keys are updated following an update event. Initially, the trusted center

- runs the RSA key generator to obtain n;
- chooses a system secret $s_0 \in \mathbb{Z}_n^*$ at random;
- defines a mapping $\phi : L \to \mathbb{N}^*$ such that $\phi(x) \mid \phi(y)$ if and only if $y \leqslant x$;
- publishes $\{n\} \cup \{\phi(x) : x \in L\}$;
- defines $\kappa_0(x) = s_0^{\phi(x)} \mod n$.

Following the ith update event, the trusted center

- chooses a new system secret $s_i \in \mathbb{Z}_n^*$, where $(s_i, s_j) = 1$ for all $j < i$;

[6] The scheme is initialized following the 0th update event.

- defines $\kappa_i(x) = s_i^{\phi(x)} \mod n$.

A user with security label x can derive $\kappa_j(y)$, $y \leqslant x$, using $\kappa_j(x)$ and the public information.

A significant advantage of this scheme is that the number of update events does not need to be defined before the scheme is instantiated. Of course, the user now accumulates keys as the scheme develops, so private storage requirements increase. The other disadvantage with this scheme is that all keys are updated following each update event, unlike in the naïve approach. We refer to this feature as *synchronous scheme update* (SSU).

4.3 KRS/NBKAS

A refinement of the scheme above is to use a KRS for generating new system secrets. The scheme is initialized in the same was as before. However, following the ith update event, the trusted center

- computes a new system secret $s_i = s_{i-1}^d \mod n$;
- defines $\kappa_i(x) = s_i^{\phi(x)} \mod n$.

With this scheme, the user only needs to retain the most recent key; he can derive $\kappa_j(y)$ from $\kappa_i(x)$, $y \leqslant x$, $j \leqslant i$, by first computing $\kappa_i(y)$ from the public information, and then iteratively computing $\kappa_{j'}(y)$, $i > j' \geqslant j$.

4.4 NBKAS/IKEKAS

For this scheme, the trusted center must choose m in advance. The basic idea is to define a NBKAS to generate a sequence of keys $\{\kappa_i(x) : 0 \leqslant i \leqslant m\}$ for each $x \in L$. Again, we assume the use of a suitable RSA key generator that outputs n (e and d are not used). For each $x \in L$, the trusted center

- chooses a secret $s(x) \in \mathbb{Z}_n^*$, such that $(s(x), s(y)) = 1$ if and only if $x = y$;
- defines $\kappa_j(x) = \big(s(x)\big)^{2^{m-j}} \mod n$.

Each of these schemes is an instance of the Akl-Taylor scheme for $[m]$, where $\phi(i) = 2^{m-i}$. Clearly, $\phi(i) \mid \phi(j)$ if and only if $j \leqslant i$; in fact, $\phi(j)/\phi(i) = 2^{i-j}$.

We use the keys derived in the construction of the above family of schemes to construct a further family of schemes; for each $i \in T$, define $Pub_i = \{E_{\kappa_i(x)}(\kappa_i(y)) : y \lessdot x\}$. In other words, for each time period, we construct an IKEKAS for L. The trusted center publishes Pub_i following the ith update event; all previous public information can be destroyed. The trusted center must also publish the generation number i.

In order to recover $\kappa_j(y)$, given $\kappa_i(x)$, with $y \leqslant x$ and $j < i$, the user first computes $\kappa_i(y)$ using the IKEKAS. The user then computes $\big(\kappa_i(y)\big)^{2^{i-j}} = \kappa_j(y)$, since

$$\big(\kappa_i(y)\big)^{2^{i-j}} = \big(\kappa_i(y)\big)^{\frac{2^{m-j}}{2^{m-i}}} = \big(s(y)^{2^{m-i}}\big)^{\frac{2^{m-j}}{2^{m-i}}} = s(y)^{2^{m-j}} = \kappa_j(y).$$

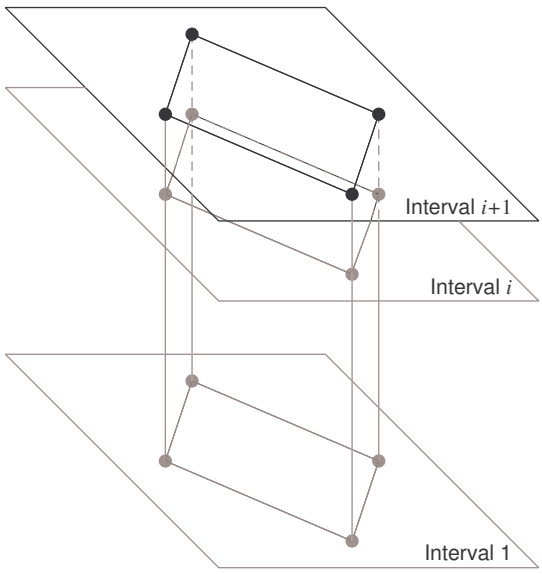


Figure 1: A stack of IKEKASs

Figure 1 illustrates schematically how the scheme develops over time. The stack defines a "virtual" family of KASs; only the public information for the top of the stack is required and public information for all earlier schemes can be discarded.

We note that the NBKAS defined for $\{\kappa_i(x) : 0 \leqslant i \leqslant m\}$ could be used as a KRS. It offers a number of advantages over **KR-RSA**. In particular, key derivation is direct and no public information is required. Given that key derivation is iterative and uses exponentiation in **KR-RSA**, direct derivation may well be useful in practice. The one disadvantage is that the trusted center has to specify the maximum number of updates in advance.

4.5 IKEKAS/IKEKAS

As an alternative to NBKAS/IKEKAS, we can use an IKEKAS instead of a NBKAS. For each $x \in L$, the trusted center

- for each i, chooses a key $\kappa_i(x)$;
- computes $\{E_{\kappa_i(x)}(\kappa_{i-1}(x)) : i = 1, \ldots, m\}$.

Following the ith revocation, the trusted center publishes Pub_i for the ith IKEKAS, as in Scheme 2. In addition, the trusted center also publishes $\{E_{\kappa_i(x)}(\kappa_{i-1}(x)) : x \in L\}$. The public information following the ith revocation is $Pub_i \cup \{E_{\kappa_j(x)}(\kappa_{j-1}(x)) : j = 1, \ldots, i\}$.

In order to recover $\kappa_j(y)$, given $\kappa_i(x)$, with $y \leqslant x$ and $j < i$, the user first computes $\kappa_i(y)$ using the IKEKAS. The user then iteratively derives $\kappa_{j'}(y)$, $i > j' \geqslant j$, using the public information for y's IKEKAS.

4.6 ASU NBKAS/IKEKAS

Of course the method of updating keys used in the schemes described thus far (with the exception of the naïve approach) is undesirable in some ways. In particular, every key is updated following a update event. This means that all keys will subsequently need to be distributed to users, even when lazy revocation and lazy re-encryption is employed. In short, all the schemes have SSU. We now describe a scheme with *asynchronous scheme update* (ASU).

We assume that the trusted center has already constructed a NBKAS/IKEKAS as in Section 4.4. Suppose that we need to update the keys in the update set $U \subseteq L$. In this case, following a update event, the trusted center

- for each $x \in U$, replaces $\kappa(x)$ in the IKEKAS with the next $\kappa(x)$ defined in the NBKAS for x;
- refreshes the public information by re-computing $\{E_{\kappa(x)}(\kappa(y)) : y \in U, x \in L, y \lessdot x\}$.

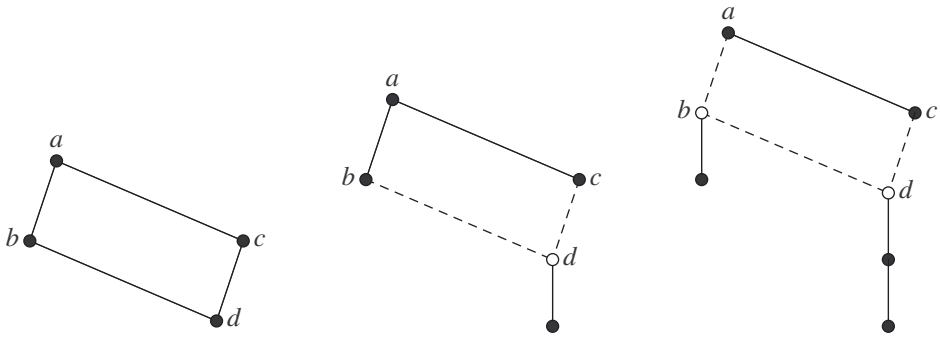
In addition, the trusted center will need to maintain a public list, indicating how many times each key has been refreshed.

Figure 2 illustrates how such a scheme would evolve over time. We have four security labels a, b, c, d, where $d \lessdot b \lessdot a$ and $d \lessdot c \lessdot a$. Prior to any updates, a single key exists for each security label. A user with security label d is then removed from the scheme, meaning that $\kappa(d)$ has to be refreshed. Since d is the minimal element in the poset, no other keys need to be changed. We must recompute the public information $E_{\kappa(b)}(\kappa(d))$ and $E_{\kappa(c)}(\kappa(d))$, the "edge encryptions" for (d, b) and (d, c). This is depicted in Figure 2(b), in which labels in the update set are shown as unfilled nodes, and edges for which the public information must be updated are shown as broken lines. Then a user with security label b is removed from the scheme, meaning that $\kappa(b)$ has to be refreshed. Since $d < b$, we must also refresh $\kappa(b)$. In addition, we must re-compute all edges from elements immediately greater than either b or d. This is shown in Figure 2(c). Notice that if, instead of removing the user with security label b from the scheme, we changed her security label to c, then the update set is $\{b\}$ rather than $\{b, d\}$ (since $d < c$); in this case, we would only update $\kappa(b)$ and re-compute $E_{\kappa(a)}(\kappa(b))$ and $E_{\kappa(b)}(\kappa(d))$.

It should be noted that many updates for label l will result in an update set $\downarrow l$. In some scenarios, it may be simpler to use SSU. However, if we can assume (or know) that most of the update activity will be for relatively junior security labels, then there may well be a distinct advantage to using this incremental approach. This may well be a reasonable assumption: there will probably be fewer users with relatively senior labels, and those users are probably less likely to be re-assigned to new security labels or removed from the scheme. Note also that the amount of public information remains constant, and is proportional to the number of edges in the Hasse diagram of L. Of course, we can use a similar method to construct an ASU IKEKAS/IKEKAS scheme.

4.7 KRS/IKEKAS

A disadvantage of NBKAS/IKEKAS and IKEKAS/IKEKAS is that m, the maximum number of key revocations for any particular security label, must be chosen in advance.



(a) No updates (b) $\kappa(d)$ is updated (c) $\kappa(b)$ is updated

Figure 2: Incremental key updates

In this scheme we construct a hybrid scheme using a KRS and an IKEKAS. Initially, for each label $x \in L$, the trusted center

- runs the RSA key generator to produce (n, e, d);
- chooses secret $s(x)$ at random from $\mathbb{Z}_n^*$;
- defines $\kappa_0(x) = s(x)$;[7]
- publishes $\{n, e\}$;
- constructs an IKEKAS for L.

Following the ith key revocation for security label x, the trusted center

- computes, $\kappa_i(x) = \kappa_{i-1}^d \mod n$;
- replaces $\kappa_{i-1}(x)$ in the IKEKAS with $\kappa_i(x)$;
- updates the public information for the IKEKAS.

This scheme can also be used in ASU mode.

4.8 Security considerations

4.8.1 IKEKASs

Ateniese *et al* recently examined the security of IKEKASs [3]. They introduce the notions of *key recovery* and *key indistinguishability*, which correspond to plaintext recovery and plaintext indistinguishability in conventional security analyses. Key indistinguishability is an important consideration in IKEKASs because keys are encrypted and stored in the public information. Ateniese *et al* considered active and passive adversaries and showed an active adversary has no advantage over a passive one for either key recovery or key indistinguishability.

[7]Note that, unlike the original scheme of Fu *et al*, we do not use hash functions to compute keys from secrets. Instead, we compute keys directly; key indistinguishability is not an issue here.

An IKEKAS is secure against key recovery if the algorithm used to encrypt keys (stored in Pub) is secure with respect to a non-adaptive chosen plaintext attack, in which a polynomial-time adversary has access to an encryption oracle (but not to a decryption oracle) and wins if it can recover the plaintext from a challenge ciphertext with non-negligible advantage (**PR-P1-CO** [3]).

An IKEKAS is secure in the sense of *key indistinguishability* if the algorithm used to encrypt keys is secure with respect to a non-adaptive chosen plaintext attack, in which a polynomial-time adversary has access to an encryption oracle and wins if it can determine with non-negligible advantage which of two plaintexts yielded a challenge ciphertext (**IND-P1-CO** [18]).

In other words, the security properties of an IKEKAS appear to be dependent only on the security properties of the algorithm used to encrypt keys stored in Pub.

4.8.2 NBKASs

NBKASs are different from IKEKASs in that no encrypted keys are stored in the public information. This means that key indistinguishability is not an issue.

The Akl-Taylor scheme, which is the NBKAS we will use throughout this paper, remains one of the few schemes with a formal proof that it is *collusion secure*: that is, for any $x \in L$, it is not feasible to derive $\kappa(x)$ from any (sub)set of $\{\kappa(y) : y \not\geq x\}$ [1]. This is analogous to the fact that an adversary can obtain the key for any label provided it cannot be used to derive the desired key in the key recovery analysis by Ateniese *et al*. The Akl-Taylor scheme is collusion secure based on the assumption that it is hard to compute integral roots mod n.

4.8.3 Hybrid schemes for lazy updates

Space constraints preclude a formal security analysis of the schemes proposed in this paper. We hope to prove in future work that the hybrid schemes are as secure as the components from which they are constructed (using methods analogous to those of Ateniese *et al* for IKEKASs). Given that there exist KASs with known security properties, we would then have a "recipe" for instantiating provably secure hybrid schemes that support lazy updates.

4.9 Summary and comparison

We summarize the features of the schemes described above in Table 1: we write "Dir" to denote direct key derivation and "Ind" to denote indirect key derivation; E denotes the edge set in the Hasse diagram of L; the "bounded" column indicates whether a maximum number of updates is specified when the scheme is instantiated; v denotes that a user is required to store v keys after v updates; the storage required is proportional to the value given in the table (the constant of proportionality being determined by the cryptographic primitives chosen).

It is difficult to make a definitive statement about which of these schemes is the best, because "best" may depend on the context in which a scheme is to be deployed. In an

Scheme	Derivation	Storage		Bounded	ASU/SSU
		Public	Private		
Naïve	Scheme dependent			Yes	ASU/SSU
NBKAS	Dir	$\|L\|$	v	No	SSU
KRS/NBKAS	Dir + Ind	$\|L\|$	1	No	SSU
NBKAS/IKEKAS	Dir + Ind	$\|E\|$	1	Yes	ASU/SSU
IKEKAS/IKEKAS	Ind + Ind	$\|E\| + m\|L\|$	1	Yes	ASU/SSU
KRS/IKEKAS	Ind + Ind	$\|E\|$	1	No	ASU/SSU

Table 1: A summary of key assignment schemes for lazy updates

ideal world, a scheme would have direct key derivation, low storage costs, be unbounded, and support ASU. Of course, none of these schemes have all these features.

If update events are likely to be rare, then it is probably better to select a scheme with bounded updates, because the key derivation methods are often more efficient. If, on the other hand, it is not appropriate to bound the number of updates, then KRS/IKEKAS offers a nice combination of features.

5 Applications

5.1 Cryptographic file systems

We assume that we have a file system and that we wish to control access to those files. Directories are special types of files that are used to arrange the file system in a tree-like hierarchy. In order to access a file, it is necessary to have access to the directory that contains the file. Every file has a unique pathname, constructed by traversing a path from the root directory to the file, and concatenating the node identifiers, separated by some suitable character. We will use the familiar Unix-style notation for pathnames. Hence `/users/dave/foo.c` is the pathname of the file `foo.c`, which is contained in the `dave` directory, itself contained in the `users` directory, which is in the root directory.

We wish to use hierarchical access control policies to control access to such a file system. We will implement these policies using a KAS.

5.1.1 Implementing a hierarchical access control policy

We model a hierarchical file system as a partially ordered set $(\mathcal{F}, \preccurlyeq)$, where $\mathcal{F}$ denotes the set of files in the file system. Given two files f_1 and f_2, $f_1 \preccurlyeq f_2$ if and only if the pathname of f_2 is a substring of the pathname of f_1. (In other words, f_1 is contained in some (sub)directory of f_2.)

An access control policy for $\mathcal{F}$ is defined to be a function $\lambda : \mathcal{F} \to L$, where L is some set of security labels. We say the policy λ is *consistent* if for all $f_1, f_2 \in \mathcal{F}$ such that $f_1 \preccurlyeq f_2$, $\lambda(f_1) \geqslant \lambda(f_2)$. Intuitively, a consistent policy applies at least as high a security label to a file as the security label for each of the directories in which that file is contained. In other words, it is consistent with the usual access control semantics for file

systems, where access to a file is denied if access is denied to any directory in the file's pathname. In what follows, we only consider consistent policies.

We can now apply a KAS to L to obtain encryption keys for the files in $\mathcal{F}$. We assign labels to users who wish to access files in $\mathcal{F}$. The advantage of this approach is that any consistent policy automatically enforces access control requirements for hierarchical file systems. Moreover, different files in the same directory can be given different security labels.

5.1.2 Role-based access control

This type of approach could be used to implement a limited form of cryptographically enforced role-based access control for hierarchical file systems. The security function λ acts as both a user-role and permission-role assignment relation. In this case, each user is assigned to a single role. However, there is no reason why we can't define a relation $\Lambda : U \times L$. In this case, users would require one or more keys, rather than a single key. Obviously, we can not extend λ to a relation for objects, because this would imply that an object could be associated with two or more labels, whereas an object is only encrypted with one key.

5.1.3 Read-write access

It is rather easy to extend this type of scheme to discriminate between read-only and read-write access. This is something that many cryptographic file systems cannot do, because a single symmetric encryption key is associated with each file.

Specifically, we (logically) extend $\mathcal{F}$ to include two entries for each normal file f (that is, one that isn't a directory): f_r for read access to the file, and f_w for write access. We define $f_w \prec f_r$, which means that for any consistent policy λ, $\lambda(f_w) \geqslant \lambda(f_r)$. In other words, any user with security label $\lambda(f_w)$ can write to f, and can also deduce the key for $\lambda(f_r)$. In contrast, a user with label $\lambda(f_r)$ can only read the file.

In a practical implementation, any user with read access would be able to change f in memory, but would need to compute a checksum before the file is written back to disk. This would be implemented using $\kappa(\lambda(f_w))$ as input to a keyed hash function in order to compute the checksum. The file system can verify the correctness of the checksum and only write the file to disk if it has been computed correctly (that is, with the correct key). This is the approach adopted in the cryptographic storage file system (CSFS) [12], for example. However, CSFS only has a single key associated with a file, meaning that any user who can read the file can also write to it.

5.1.4 Comparison with related work

We have already noted that we can quite easily offer discriminated read and read-write access to files, unlike existing approaches. In addition, we can significantly reduce the number of keys in use. Existing approaches partition the file system and associate a group key with each set of files in the partition [4, 12]. A Unix-like access control policy is used to determine which group the file is associated with and which users should have access to the file. A group key is encrypted with the public key of each member of the group.

The hierarchical access control policy allows us to simplify the access control policy and to deal with hierarchical file systems in a very natural way. The disadvantage of our approach is that all files are "owned" by the system, not individual users. However, this is a feature of mandatory and role-based access control policies.

5.2 KAS for temporal access control

Temporal access control adds a further dimension to access control policies, by limiting the time for which certain rights are authorized. This has been explored in the context of role-based access control [6] and access control mechanisms for broadcast XML data [7], and is supported in XACML [21]. Typically, some unit of time is split into smaller intervals (days into hours, years into months, etc) and users are authorized for some continuous period of time comprising one or more smaller intervals. A user might be authorized to access files only between the hours of 9am and 6pm, for example.

One application of these schemes is to implement temporal access control to published XML data. Different parts of the content are encrypted with different keys. Subscribers are given the keys that will enable them to decrypt the content during the time intervals for which they have paid subscription fees [7].

So called "time-bound" KASs have attempted to add this temporal dimension to KASs. Such schemes have been widely studied [3, 7, 9, 17, 23], but many of these schemes have been shown to be insecure [16, 24, 25], and are often rather complicated. We now show that the schemes defined in Section 4 to support lazy updates can be used, either directly or with some minor modification, to construct KASs for temporal hierarchical access control policies.

We assume that there are m time intervals and that a user will be given access for i consecutive time intervals $1 \leqslant i \leqslant m$. (Hence, there are $m+1$ time points, $t_0, \ldots, t_m$, which denote the beginning/end points of time intervals.) We write $[i, j]$ to denote the time period between t_{i-1} and t_j, $j \geqslant i$. There are $\frac{1}{2}m(m+1)$ such intervals. The set of intervals, which we denote $\mathcal{I}_m$ has a natural partial ordering that respects the semantics of temporal access control: $[i, j] \leqslant [i', j']$ if and only if $i \geqslant i'$ and $j \leqslant j'$. We write $\kappa_i(x)$, $1 \leqslant i \leqslant m$, to denote the key for security label x during interval i.

5.2.1 Naïve approach

The simplest way to implement a temporal KAS is to construct a normal KAS for the poset $L \times \mathcal{I}_m$. This is analogous to Scheme 0 in Section 4. Of course, the scheme will require a considerable amount of public storage.

5.2.2 NBKAS

The second possibility is to simply use the NBKAS from Section 4.2. The trusted center initially:

- runs the RSA key generator to obtain n;
- chooses a system secret $s_1 \in \mathbb{Z}_n^*$ at random;

- defines a mapping $\phi : L \to \mathbb{N}^*$ such that $\phi(x) \mid \phi(y)$ if and only if $y \leqslant x$;
- publishes $\{n\} \cup \{\phi(x) : x \in L\}$;
- defines $\kappa_1(x) = s_1^{\phi(x)} \mod n$;
- distributes $\kappa_1(x)$ to every user with security label x authorized in period 1.

As before, a user with security label x can use $\kappa_1(x)$ to compute $\kappa_1(y)$ for all $y \leqslant x$.

At the beginning of the ith time period, the trusted center:

- chooses a system secret $s_i \in \mathbb{Z}_n^*$ at random, where $(s_i, s_j) = 1$, $j < i$;
- defines $\kappa_i(x) = s_i^{\phi(x)} \mod n$;
- distributes $\kappa_i(x)$ to every user with security label x authorized in period i.

It is clear that a user with security label x can use the same public information, the same derivation method, and the new key to derive $\kappa_i(y)$ for all $y \leqslant x$.

In fact, this scheme can be employed even if users are not authorized for a contiguous time period. The trusted center simply distributes keys if the user is authorized for a particular interval at the beginning of that interval. A further advantage of this scheme is that keys can be "drip-fed" to users. Hence, it is not necessary to update keys if a user is removed from the scheme. (This is based on the assumption that the applications for which temporal access control is envisaged would write and encrypt the content only once.)

5.2.3 Comparison with related work

Bertino *et al* proposed the use of a KAS to implement temporal hierarchical access control policies [7]. However, the KAS they use [23] has been shown to be insecure [25].

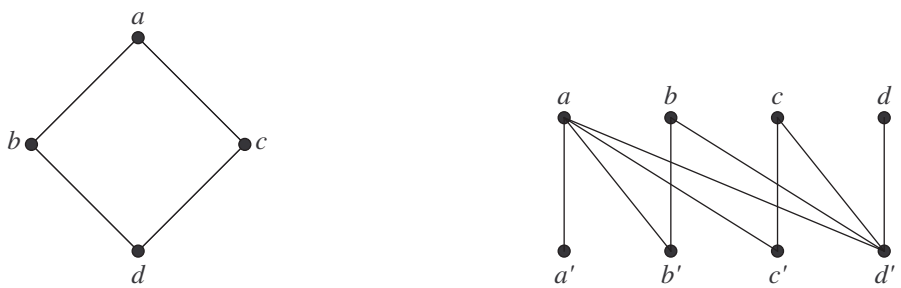
Ateniese *et al* propose two schemes for temporal access control. Given L and m, both schemes require the construction of a new poset. The first scheme is an IKEKAS defined for the poset $L' = (L \times \mathcal{I}_m) \cup (L \times [m])$, where $(x, y) \leqslant (x', y')$ if and only if $x \leqslant x'$ (in L), $y \in [m]$, $y' \in \mathcal{I}_m$ and $y \in y'$. It can be shown that the number of edges in L' is $\mathcal{O}(pm^3)$, where p is the cardinality of the partial order relation in L. Hence, the amount of public information that needs to be stored (which is proportional to the cardinality of the number of edges in an IKEKAS) may be large.

The second scheme is a direct key encrypting key assignment scheme, but it also requires that a user store secret values, one for each of the time intervals for which he is authorized to have access. Keys are obtained and derived (in one step) by computing bilinear mappings. This scheme is defined for the poset $L_r \cup L_l$, where $L = L_r = L_l$ (as sets) and $x \leqslant y$ if and only if $x \in L_l$, $y \in L_r$ and $x \leqslant y$ (in L). (An illustration of this construction is shown in Figure 3.) In this case, the number of edges in L' is $\mathcal{O}(|L|^2)$. Although the amount of public information that needs to be stored is independent of m, it is still proportional to the square of $|L|$.

The use of bilinear mappings seems somewhat arbitrary: any other encryption scheme with appropriate security properties would appear to work just as well; indeed, the same poset transformation and an Akl-Taylor scheme for the transformed poset has been described previously [22]. The motivation for using bilinear mappings appears to be that

the size of each datum of public information is constant (and occupies 171 bits). However, we note that the scheme proposed by Atallah *et al* [2], and illustrated in Appendix A.1, employs hash functions, and each datum occupies only 160 bits in a typical implementation.

In our NBKAS approach, the amount of public information is proportional to L. It has been shown that we can choose $|L|$ primes and a function ϕ such that, $\phi(x)$ occupies $|L|$ bits [11] (see also Appendix A.2). Additional storage is required to store $|L|$ primes, each of which occupies $\log |L|$ bits. In other words, the overall public storage requirement is $|L|(\log |L| + 1)$ bits. Given that the bilinear mapping scheme of Ateniese *et al* requires $\mathcal{O}(|L|^2)$ items of public information, our scheme compares rather favorably. Figure 3 illustrates a simple poset and the corresponding poset that needs to be constructed for the scheme of Ateniese *et al* based on bilinear mappings.



(a) Simple poset

(b) "Bilinear poset"

Figure 3: Constructing the poset for the bilinear mapping KAS

We also note that we can drop the requirement that m is known in advance in our approach, assuming that the number of intervals that will ever be used is very small compared to n. This may be useful if the temporal access control policy is cyclical. In other words, the same hierarchical access control policy applies to each block of intervals. (We might require that users with a particular security label can access content every Monday, for example, and the content published on Monday requires different encryption keys each week.) Policies of this nature have been used by Bertino *et al* in the setting of role-based access control. However, we are not aware of any previous work on the implementation of cyclical temporal hierarchical access control policies using KASs.

6 Conclusion

Updating keys is known to be an issue in situations where the same encryption key is used to encrypt many different data objects. This problem occurs in cryptographic storage file systems, where techniques such as key regression have been used so that re-encryption of data objects is not necessarily performed when the encryption key is refreshed. The problem is even more pressing in key assignment schemes for hierarchical access control, in which a number of different keys may need to change at the same point in time, meaning that even higher volumes of data may need to be re-encrypted at some point. We have

proposed a number of different generic key assignment schemes that support the idea of lazy re-encryption. These are the first such schemes in the literature, and represent a considerable advance on existing approaches to key updates in KASs. We have also shown that key regression techniques used in cryptographic file systems are also KASs. Finally we have demonstrated that KASs with lazy updates have a number of useful applications.

Nevertheless, a number of opportunities for further work exist. We have not include a formal security analysis for any of our schemes. There are a number of reasons for this. First, we felt the most important contribution of this paper was to introduce the idea of lazy updates for key assignment schemes and to define generic constructions to support lazy updates. Nearly all papers on key assignment schemes describe a scheme with particular cryptographic primitives and prove the scheme is secure. We feel it is more useful to identify the characteristics of a generic scheme; the security analysis can follow for instantiations of the generic scheme using particular cryptographic mechanisms. Second, the space limitations preclude any substantial analysis. Instead, we have noted that there do exist provably secure IKEKASs, NBKASs and KRSs in the literature. In future work, we hope to prove that our generic constructions to support lazy updates are as secure as the components from which they are constructed. In other words, the KRS/NBKAS construction, for example, is secure from in the sense of key recovery and key indistinguishability if the KRS and NBKAS are.

Finally, it would of considerable practical interest to adapt the architecture used in cryptographic file systems and implement a key assignment scheme for a hierarchical file system. Similarly, it would interesting to use XML encryption to implement hierarchical access control for XML data.

References

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

[2] M.J. Atallah, K.B. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *Proceedings of 12th ACM Conference on Computer and Communications Security*, pages 190–202, 2005.

[3] G. Ateniese, A. De Santis, A.L. Ferrara, and B. Masucci. Provably-secure time-bound hierarchical key assignment schemes. Cryptology ePrint Archive, Report 2006/225, 2006. `http://eprint.iacr.org/2006/225.pdf`.

[4] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *Proceedings of 11th European Symposium on Research in Computer Security*, pages 327–346, 2006.

[5] D.E. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, Mitre Corporation, Bedford, Massachusetts, 1976.

[6] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001.

[7] E. Bertino, B. Carminati, and E. Ferrari. A temporal key management scheme for secure broadcasting of XML documents. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 31–40, 2002.

[8] T.-S. Chen and J.-Y. Huang. A novel key management scheme for dynamic access control in a user hierarchy. *Applied Mathematics and Computation*, 162:339–351, 2005.

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

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

[11] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *Proceedings of 19th Computer Security Foundations Workshop*, pages 98–111, 2006.

[12] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999. `http://prisms.cs.umass.edu/~kevinfu/papers/fu-masters.pdf`.

[13] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. Cryptology ePrint Archive, Report 2005/303, 2005. `http://eprint.iacr.org/2005/303.pdf`.

[14] N.M. Haller. The S/KEY one-time password system. In *Proceedings of the 1994 Symposium on Network and Distributed System Security*, pages 151–157, 1994.

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

[16] C.-L. Hsu and T.-S. Wu. Cryptanalyses and improvements of two cryptographic key assignment schemes for dynamic access control in a user hierarchy. *Computers & Security*, 22(5):453–456, 2003.

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

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

[19] C.-H. Lin. Dynamic key management schemes for access control in a hierarchy. *Computer Communications*, 20:1381–1385, 1997.

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

[21] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005. OASIS Committee Specification (T. Moses, editor): `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

[22] A. De Santis, A.L. Ferrara, and B. Masucci. Cryptographic key assignment schemes for any access control policy. *Information Processing Letters*, 92:199–2005, 2004.

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

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

[25] 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*, 15(4):1054–1055, 2003.

[26] Y. Zheng, T. Hardjono, and J. Seberry. New solutions to the problem of access control in a hierarchy. Technical Report 93-2, Department of Computer Science, University of Wollongong, 1993.

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

A Some concrete key assignment schemes

In the interests of concreteness, we describe an IKEKAS and a NBKAS for the simple poset shown in Figure 4.

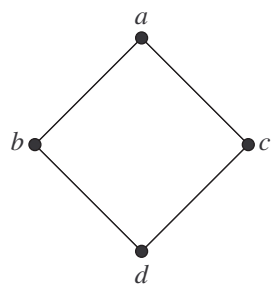


Figure 4: A simple poset

A.1 An IKEKAS based on the scheme of Atallah *et al*

We choose an integer l and a hash function $h : \{0,1\}^* \to \{0,1\}^l$ and define the key space to be $\{0,1\}^l$. We then choose four keys $\kappa(a), \ldots, \kappa(d)$ at random from the key space. Finally, we publish $\{\kappa(y) - h(\kappa(x) \parallel y) : y \lessdot x\}$, which equals

$$\{\kappa(b) - h(\kappa(a) \parallel b), \kappa(c) - h(\kappa(a) \parallel c), \kappa(d) - h(\kappa(b) \parallel d), \kappa(d) - h(\kappa(c) \parallel d)\}.$$

A user with $\kappa(a)$ can derive $\kappa(d)$, for example: he first computes $h(\kappa(a) \parallel b)$ and hence derives $\kappa(b)$ from the relevant public information; and then computes $\kappa(d)$ using a similar method and $\kappa(b)$.

A.2 An Akl-Taylor scheme

We first define a mapping $\chi : L \to 2^L$, where $\chi(x) = L \setminus \downarrow x$. We also define a mapping $\pi : L \to \{2,3,5,7\}$, where $\pi(a) = 2$, $\pi(b) = 3$, $\pi(c) = 5$ and $\pi(d) = 7$; we extend this to a mapping $\pi : 2^L \to \mathbb{N}^*$, where $\pi(X) = \prod_{x \in X} \pi(x)$. Finally, we define the mapping $\phi : L \to \mathbb{N}^*$, where $\phi(x) = \pi(\chi(x))$. Hence, we have $\phi(a) = 1$, $\phi(b) = 2.5$, $\phi(c) = 2.3$ and $\phi(d) = 2.3.5$. By construction, $\phi(x) \mid \phi(y)$ if and only if $y \leqslant x$.

We use an RSA key generator to produce $n = pq$, and then select a system secret $s \in \mathbb{Z}_n^*$. We define $\kappa(x) = s^{\phi(x)} \mod n$. Finally, we publish n and the mappings π and χ. A user with $\kappa(a)$ can derive $\kappa(d)$, by computing (mod n) $\kappa(a)^{\frac{\phi(d)}{\phi(a)}} = \kappa(a)^{\phi(d)} = s^{\phi(d)} = \kappa(d)$.