On Key Assignment Schemes and Cryptographic Enforcement Mechanisms for Information Flow Policies

Naomi Farley

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
School of Mathematics and Information Security
Royal Holloway, University of London

2018

Declaration

These doctoral studies were conducted under the supervision of Professor Jason Crampton and Professor Gregory Gutin.

The work presented in this thesis is the result of original research I conducted, in collaboration with others, whilst enrolled in the School of Mathematics and Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Naomi Farley

November 11, 2018

Abstract

Access control policies specify permissible interactions between users and system resources, and are typically enforced by trusted components. Third parties (e.g. cloud servers) may not be trusted to correctly enforce a policy, in which case cryptographic enforcement schemes (CESs) may be used.

In this thesis, we consider the cryptographic enforcement of (read-only) information flow policies, which model hierarchies of security labels. For example, a symmetric key can be associated with each security label and used to encrypt associated objects. Users authorised for many labels may need to be issued many keys, which may be undesirable, particularly when user storage is limited. A key assignment scheme (KAS) allows a trusted entity to generate a 'small' secret for each user, from which all required keys can be derived. Key derivation may also rely on additional public information, which can be large and expensive to maintain.

In this thesis, we propose three symmetric KASs that eliminate public derivation information. Our first KAS is based on partitioning the policy hierarchy into chains, which permits very efficient key derivation. We show how to construct a chain partition that minimises the cryptographic material required both in total and by any one user. We then show that working with trees, rather than chains, further reduces the material distributed to users and that tree partitions are quicker to find than chain partitions. We then design a space-efficient KAS that imposes a logarithmic bound on derivation cost. In the worst case, user material may be larger than in prior schemes; we therefore design heuristic approaches and provide experimental evidence that the resulting schemes compare favourably to existing schemes.

Finally, we provide a definitional framework for CESs for read-only information flow policies, using which CESs can be proven correct and secure, and which helps identify limitations of primitives in CESs.

Statement of Contribution

This thesis is based on the following published papers:

- J. Crampton, N. Farley, M. Jones and G. Gutin, *Optimal Constructions for Chain-Based Cryptographic Enforcement of Information Flow Policies*, DBSec 2015.
- J. Crampton, N. Farley, M. Jones, G. Gutin and B. Poettering, *Cryptographic Enforcement of Information Flow Policies without Public Information*, ACNS 2015.
- J. Crampton, N. Farley, M. Jones, G. Gutin and B. Poettering, *Cryptographic Enforcement of Information Flow Policies without Public Information via Tree Partitions*, Journal of Computer Security 25(6): 511-535 (2017).
- J. Alderman, N. Farley and J. Crampton, *Tree-based Cryptographic Access Control*, ESORICS, 2017.
- J. Alderman, J. Crampton and N. Farley, *A Framework for the Cryptographic Enforcement of Information Flow Policies*, SACMAT, 2017.

For Jade Schuhmacher, who was like a sister to me during my time at Royal Holloway in undergrad, and who sadly passed away during my PhD.

Acknowledgements

Firstly, I'd like to thank my supervisors Jason Crampton and Gregory Gutin for encouraging me to undertake a PhD. In particular, I'd like to thank Gregory for putting me forward for the opportunity, and Jason for taking me under his wing. Thank you also for your support throughout my PhD. I'd also like to thank my co-authors Mark and Bertram for being awesome to work with. Thank you also to the EPSRC through the Centre for Doctoral Training at Royal Holloway for their financial support and making my PhD possible, and to Thales UK for my internship opportunity. I am also very grateful to Keith Martin and Bogdan Warinschi for giving me the thumbs up in my PhD viva.

I've met many great people during my PhD at Royal Holloway, who I'd like to thank for all the social outings, the games of squash, and meetups at the Happy Man. In particular, I'd like to thank Christian, Dan and Rachel for their words of encouragement and reassurance through the PhD years, and for making the escape room games more intense and exciting!

I'd also like to thank my family for their constant support. In particular, my Mum and Dad who have always pushed me to do my best and have always let me know of their love and support. Thank you for giving me opportunities in life that have led to me to be able to undertake a PhD.

Lastly, I'd like to thank James for always being there, for listening and for putting up with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This chapter highlights the motivation for the research into key assignment schemes and other cryptographic enforcement mechanisms for information flow policies and outlines the structure of this thesis.

1.1 Motivation

Access control is an essential security service in most multi-user computing systems and restricts users' interactions with system resources. Typically, those interactions that are authorised between users and resources of a system are defined within an *access policy*. One general form of access policy is an *information flow* policy. Information flow policies are useful for defining access for users in schemes in which access is defined in terms of a hierarchy. Such policies have been widely studied in the literature [3, 6, 26, 30, 32, 36, 37, 47, 68] and encompass many forms of policy that are useful in practice, for example temporal [8, 29], geo-spatial [7], role- and attribute-based access control policies [30] (RBAC and ABAC respectively).

Traditionally, an access control policy is enforced using a low-level trusted software component (e.g. a *policy decision point*); users send access requests, specifying the resource

and access permission (e.g. read, write, execute) they desire, to the policy decision point. Upon receiving such a request, the decision point would compare the access request to the underlying access policy to determine if the request is authorised or not. Such a mechanism for enforcing access is only appropriate when the access policy is enforced in a trusted environment (e.g. the policy is enforced by the same organisation that defined it).

Increasingly, however, organisations are outsourcing data storage to untrusted third party (e.g. cloud) servers. The use of third party (i.e. decentralised and untrusted) storage means that our assumption that our policy will be enforced in a trusted environment no longer holds. In particular, we may not trust the third party to correctly enforce the access policy, or to not read the data that we outsource to it. We only trust the third party to ensure that the data is always available and not to tamper with the data that it provides.

Thus, in such situations, we may consider enforcing our access policy using a *cryptographic enforcement scheme* (CES). In such a scheme, data must be protected (e.g. encrypted) before it is outsourced in order to prevent the untrusted server from learning the contents of the data; this therefore protects the *confidentiality* of the data. However, the problem is now how to ensure that authorised users retain access to the data. Since we do not trust the storage provider to enforce the policy correctly, and we do not want to provide it with decryption keys so that it can decrypt data for users (since this would enable the third party server to also read the data), we must distribute appropriate cryptographic material (via a trusted party) to users such that they can only decrypt and read data for which they are authorised. Thus, an access control policy may be enforced through the careful protection of resources and careful distribution of cryptographic material (e.g. keys) to authorised users.

When choosing an appropriate encryption scheme to protect outsourced data in this way, symmetric cryptography may be preferred over public key techniques (e.g. attribute-based encryption [18, 58, 85]) due to its better efficiency and smaller ciphertext and key sizes. For example, attribute-based encryption often uses expensive bilinear pairings [18, 58, 85, 86] and keys which are several orders of magnitude larger than symmetric keys for AES, for example (see [2] for comparisons of recommended key sizes for different cryptographic schemes).

Informally, when a symmetric encryption scheme is chosen, each class of resource may

be associated with a unique symmetric key. Thus, a user authorised for many resources may require many decryption keys. This may be expensive, both in terms of the cost of transmitting all such keys to users over secure channels, and in terms of user storage since users will be required to store all their necessary keys simultaneously using secure storage.

Instead, it would be desirable to give each user a (small) secret, from which they can derive all keys required. A *key assignment scheme* (KAS) is a cryptographic tool that enables a trusted party (e.g. data owner) to generate user secrets, and provides a key derivation mechanism to allow users to derive all their respective keys from their given secret. As a consequence of reducing the amount of cryptographic material a user may be required to store, public information may be introduced in order to aid the derivation of keys, and multiple computations may be required in order to derive any key from a given user secret.

Thus a key assignment scheme may be characterised by:

- the size of the secret each user has to store;
- the total amount of cryptographic material distributed to users;
- the amount of auxiliary (public) information required for key derivation;
- the amount of computation/time required for key derivation.

Thus when designing such schemes, we ideally want to minimise each of these characteristics. Unfortunately, it is not possible to minimise all such characteristics simultaneously; for example, decreasing the size of user secrets may increase the amount of public information used to support key derivation and/or derivation time, and thus a trade-off must be sought. Additionally, whilst key derivation is often cheap (e.g. only requires the use of cheap functions, such as pseudorandom functions etc.), publishing and maintaining public information may be (relatively) expensive. Moreover, such public information may be expensive to provide, since an administrator must generate this information and ensure that it is always online and up-to-date. In addition, some form of public key infrastructure (PKI) may be required, for example, to allow users to verify that the information provided is authentic. Furthermore, it may not be possible to provide an on-line server to store public information (e.g. in military ad-hoc networks).

Thus, we choose to focus on key assignment schemes that require little or no public

information. We propose three different types of key assignment schemes that minimise the amount of public information and compare their respective trade-offs. In particular, we propose a chain, tree and binary tree-based KAS which enable keys to be iteratively derived down paths in the respective graph representations of the policy being enforced. The chain and tree-based KASs do not require public information (other than a graph representation of the policy poset) to support key derivation, and the binary tree-based KAS enables keys to be derived using only knowledge of labels associated with keys; thus the policy structure need not be public.

Whilst chain-based KASs have been proposed in the literature, such schemes typically assume that the policy to be enforced is a total order or is already represented as a chain partition. Thus our main contribution is to show how any information flow policy poset can be partitioned into chains as to minimise the total amount of secret material distributed to the user population, and the total number of secrets required by any user.

We then identify that information flow policy posets can be represented as trees rather than chains, whilst still minimising the amount of public information. We thus propose how information flow policy posets can be represented as trees as to further minimise the total number of secrets required to be distributed to the user population.

We then design a KAS to minimise the amount of key derivation required by each user. Representing our information flow policy poset as a binary tree enables us to logarithmically bound key derivation. We then propose heuristics for minimising the average number of secrets required per user. Such schemes may be important in devices with limited computational power, or at times in which key derivation is time-critical (e.g. in an authorisation protocol, as described in [4]).

Having proposed a number of KASs in this thesis, we then turn our attention to considering how KASs could be used within a cryptographic enforcement scheme. We show that whilst key assignment schemes are useful cryptographic tools for reducing the amount of cryptographic material required by users, these alone are not sufficient for enforcing access control. Thus we provide a rigorous framework for cryptographic enforcement schemes for read-only information flow policies, and show how one could enforce a policy using a KAS as a building block. We show that the current definition of a KAS only allows us to construct a basic CES with limited functionality. We thus identify how the definition of a

KAS should be modified in order to support more dynamic CESs.

1.2 Structure

We provide relevant definitions and background material in Chapter 2. In Chapters 3-5, we describe three different key assignment schemes which require little or no public derivation information. We begin by discussing chain-based key assignment schemes in Chapter 3, and tree-based key assignment schemes in Chapter 4. In Chapter 5, we introduce an alternative key assignment scheme based on binary trees. A comparison of the KASs proposed in this thesis is given in Section 5.6. We provide a definitional framework of a cryptographic enforcement scheme for information flow policies in Chapter 6. We conclude this thesis with a summary of the contribution of this thesis in Chapter 7.

Chapter 2

Background and Related Work

Contents

In this chapter, we provide some useful notation and definitions and discuss related work.

2.1 Notation and Definitions

We denote the empty string by ϵ and denote the concatenation of strings x and y as $x \parallel y$. We define a bit string $b \in \{0,1\}^\star$ to be a bit string of arbitrary length. For a bit string $b = b_0 b_1 \dots b_t$, we define b_i to be the i^{th} bit of b, where $i \in \{0, \dots, t\}$. We define $|b|$ to be the length of the bit string b.

A *security parameter* ρ is used within a cryptographic scheme to determine system parameter sizes such as key size etc. Typically, it is argued that the larger the security parameter, the greater the security of the cryptographic primitive [65]. We use 1^ρ to denote the unary

representation of ρ. This is useful since the run-time of algorithms within cryptographic schemes are generally quantified in terms of the length of the inputs (i.e. the length of 1^ρ).

An algorithm B runs in polynomial time if there exists a polynomial function p such that, for every input $x \in \{0,1\}^\star$, $B(x)$ takes at most $p(|x|)$ steps [65]. We define a *Probabilistic Polynomial Time* (PPT) adversary/algorithm to be one which has access to random coins (i.e. a source of randomness) and runs in polynomial time [65].

We write $a \leftarrow x$ to denote the assignment of value x to variable a, whilst $a \xleftarrow{\$} X$ denotes a being assigned a value sampled uniformly at random from the set X. We write $a \leftarrow p(c)$ to denote a polynomial time algorithm or function p being run on input c and the output being assigned to a, and write $a \xleftarrow{\$} p(c)$ if p is probabilistic polynomial time (PPT). We use $p(c) \rightarrow a$ to denote a polynomial time algorithm or function p that on input c outputs value a. A function negl is *negligible* if, for every polynomial p, there exists an $N \in \mathbb{N}$ such that for all integers $n > N$, $\mathsf{negl}(n) < \frac{1}{p(n)}$ [65].

We use the symbol $\perp$ to denote: (i) failure when output by an algorithm; and (ii) a null value when assigned to a variable. We denote the elements of a list or array A of n elements by $A[0], \ldots, A[n-1]$.

An *undirected graph* $G = (V, E)$ is defined by a vertex set V and undirected edge set E. 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.

A *directed graph* (*digraph*) $D = (V, A)$ is defined by a vertex (or node) set V and directed edge (arc) set A. For vertices $x, y \in V$, $xy \in A$ denotes a directed edge from x to y in D. A *directed path* is a sequence of arcs $v_1v_2, v_2v_3, \ldots, v_{p-1}v_p$, which we may write as a sequence of vertices $v_1v_2 \ldots v_{p-1}v_p$ through which the path passes. We write $x \rightsquigarrow_D y$ if there exists a path from x to y in D, and $x \not\rightsquigarrow_D y$ if no such path exists. For all $x \in V$, we define $x \rightsquigarrow_D x$. A directed *acyclic* graph is a directed graph that contains no directed cycles (i.e. there exists no directed path from x to x that passes through some vertex not equal to x). We say that x is an *ancestor* of y (and y is a *descendant* of x) if $x \rightsquigarrow_D y$.

$X \subseteq V$ is an *independent set* if for all $x, y \in X$ where $x \neq y$, $x \not\rightsquigarrow_D y$ and $y \not\rightsquigarrow_D x$. An independent set X is maximal if, for any independent set Y in D, $|X| \geqslant |Y|$.

The *in-degree* of a vertex $y \in V$ is defined to be the number of arcs of the form xy in A. A directed acyclic graph (DAG) $D = (V, A)$ is an *out-tree* if a single vertex $r \in V$ (the root) has in-degree 0, and all other vertices in V have in-degree 1. A directed acyclic graph D is an *out-forest* if every vertex of D has in-degree less than or equal to 1. If a directed path exists between a pair of distinct vertices in an out-tree or out-forest, it is unique. $D' = (V', A')$ is a *subgraph* of D if $V' \subseteq V$ and $A' \subseteq A$. D' is a *spanning* subgraph if $V' = V$ and *non-spanning* if $V' \subset V$. A *spanning out-tree* (*out-forest*) is a spanning subgraph that is an out-tree (*out-forest*).

A *partially ordered set* (poset) $\mathcal{P}$ [34] is a pair $(L, \leqslant)$ where L is a (non-empty) set of elements and $\leqslant$ is a binary, reflexive, anti-symmetric, transitive order relation on L. We say $\mathcal{P}$ is a *chain* or *total order* if, for all $x, y \in \mathcal{P}$, $x \leqslant y$ or $y \leqslant x$. For $x, y \in L$, we may write $x \geqslant y$ if $y \leqslant x$, and $y < x$ if $y \leqslant x, x \neq y$. Given a poset $\mathcal{P}$, for each label $l \in L$, we may define its *order filter* to be $\uparrow_{\mathcal{P}} l = \{x \in L : x \geqslant l\}$ and the *order ideal* of l to be $\downarrow_{\mathcal{P}} l = \{x \in L : x \leqslant l\}$. When $\mathcal{P}$ is clear from the context, we will simply write $\uparrow l$ and $\downarrow l$. We say that x *covers* y, denoted $y \lessdot x$, if and only if $y < x$ and there exists no $z \in L$ such that $y < z < x$. If $y \lessdot x$, we say y is the *child* of x, and x is the *parent* of y. We say that $x, y \in L$ are *incomparable*, denoted $x \shortparallel y$, if $x \not\leqslant y$, and $y \not\leqslant x$. An *antichain* $A^\star \subseteq L$ of $(L, \leqslant)$ is a set of *incomparable* elements; that is, for each $x, y \in A^\star$ where $x \neq y$, $x \shortparallel y$. $A^\star$ is a *maximum* antichain if, for all other antichains $Z \subseteq L$, $|A^\star| \geqslant |Z|$. The *width* of a poset is the cardinality of a maximum antichain.

The Hasse diagram of a poset $\mathcal{P} = (L, \leqslant)$ is the directed acyclic graph $H(\mathcal{P}) = (L, A_{\min})$ where $xy \in A_{\min}$ if and only if $y \lessdot x$ in $\mathcal{P}$. We define the directed graph $H^\star(\mathcal{P}) = (L, A_{\max})$ to be the *transitive closure* of $H(\mathcal{P})$, where $A_{\max} = \{xy : y < x\}$. We say $\mathcal{P}$ is an out-tree (out-forest) if $H(\mathcal{P})$ is an out-tree (out-forest).

$\{(C_1, \leqslant_{C_1}), \ldots, (C_t, \leqslant_{C_t})\}$ is a *chain partition* of $(L, \leqslant)$ if, for each $i, j \in \{1, \ldots, t\}$ where $i \neq j$, $C_i \subseteq L$, $C_i \cap C_j = \emptyset$, $C_1 \cup \cdots \cup C_t = L$ and $x \leqslant_{C_i} y$ if and only if $x \leqslant y$ for all $x, y \in C_i$. To simplify the expression, we will define a chain partition as $\{C_1, \ldots, C_t\}$. An example chain partition of a poset is shown in Figure 2.1. A *linear extension* of $\mathcal{P}$ is a chain $(L, \preccurlyeq)$ such that if $x \leqslant y$ then $x \preccurlyeq y$. Every (finite) partial order has at least one

linear extension, which may be computed, in linear time, by representing the partial order as a directed acyclic graph and using a topological sort [27, §22.3].

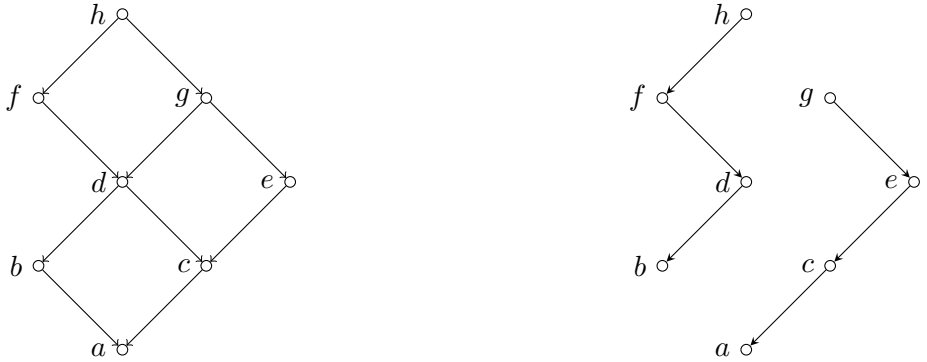


Figure 2.1: Hasse diagram of a simple poset and a chain partition of the poset.

The *power set* of a set X, denoted 2^X, is the set of all subsets of X.

Let $\mathcal{U} = \{A_1, \ldots, A_n\}$ be a set of attributes. A collection $\mathbb{A} \subseteq 2^{\mathcal{U}}$ is *monotone* if, for all B, C, if $B \in \mathbb{A}$ and $B \subseteq C$, then $C \in \mathbb{A}$. An *access structure* (respectively, monotone access structure) is a collection (respectively, monotone collection) $\mathbb{A}$ of non-empty subsets of $\mathcal{U}$ i.e. $\mathbb{A} \in 2^{2^{\mathcal{U}}}$. The sets in $\mathbb{A}$ are called the authorised sets, and the sets not in $\mathbb{A}$ are called the unauthorised sets.[1] In the context of attribute-based encryption (which we discuss later), access control policies can be described in different ways, including as access structures, and as Boolean formulas (or trees). In subsequent chapters, we may slightly abuse notation by referring to a policy as both a Boolean formula and as an access structure, i.e. we may write $A \in \mathbb{A}$ to denote A being a set of attributes that satisfy a Boolean formula represented by $\mathbb{A}$ (more formally, A belongs to the access structure comprising all satisfying sets of the Boolean formula).

Experiments. Security (or correctness) of a cryptographic scheme may be captured through the use of *experiments* (or *games*). The aim of an experiment is to capture the essence of a security notion and specify the expected behaviour such that a particular instantiation can be proven to meet these requirements. Typically, an experiment is played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. (We assume that all data sent amongst entities in an experiment is done so via confidential, integrity-preserving, authenticated channels.)

Within an experiment, the challenger sets up the cryptographic scheme/system and sets

[1] This definition is taken from [13] but the notation is adapted to refer to attributes rather than parties in a linear secret sharing scheme.

a challenge for the adversary (for example, the challenger may choose at random a course of action which the adversary will have to guess) along with some winning conditions (e.g. the adversary wins if he guesses the challenge value correctly).

The challenger will typically provide the adversary with public system parameters. In addition, the adversary may be given access to oracles. An adversary $\mathcal{A}$ given oracle access is denoted $\mathcal{A}^{\mathcal{O}}$, where the $\mathcal{O}$ denotes the set of oracles to which the adversary has access. Informally, oracles allow the adversary to influence the system by triggering the execution of algorithms, without necessarily knowing all inputs to each algorithm. This mechanism allows the adversary, to some degree, to 'embed' information of its choosing into the system and to control its execution; the resulting knowledge of the system represents any prior knowledge an adversary may have about a real system. Furthermore, oracles model an adversary taking 'real-world' actions that result in an algorithm being run by a trusted entity (e.g. system manager).

Most oracles include a call to a system algorithm and take as input a subset of the inputs to that algorithm (those which the adversary can choose). Oracles are not provided for any algorithms which the adversary can run itself. An oracle may also perform some validation of the inputs to ensure that the adversary does not provide inputs that could permit a 'trivial win'. The only information the adversary may learn is that which is explicitly given to it as input and that which is output from oracles (together this should be chosen to reflect all possible leakage in the real system).

After interacting with oracles (if permitted), the adversary must submit a response to his challenge (for example, his guess of the challenge value). We define the advantage $\mathcal{A}$ of an adversary to be the probability that the adversary wins the experiment (e.g. guesses the challenge value correctly) minus the probability of the adversary winning by 'randomly guessing' a valid challenge response (for experiments where the challenge value is either 0 or 1, this probability is $\frac{1}{2}$). Intuitively, the advantage of an adversary captures in some essence how useful the information leaked by the system is in enabling one to 'break' the system. Informally, we say that a scheme is *secure* (*correct*) if the advantage of a (PPT) adversary in the corresponding experiment is negligible in the security parameter.

2.2 Access Control Policies

As mentioned in Section 1.1, access control policies are useful for defining interactions that are authorised between users and resources of a system. Generally, a policy defines a set of users U, whose access is to be defined within the system, along with a set of data objects O, which are to be protected under the policy. A set of permissions P may also be defined, which are typically tuples of the form (o, a), where $o \in O$ and a is an access permission (e.g. read, write, execute) permitted under the policy. Users are typically assigned permissions, which specify the types of access that they are permitted to perform on specific objects. If only one form of access is defined within a policy (e.g. read access), then we may abuse notation and authorise users for objects directly. For example, if only read access was enforced, then a user u being authorised for data object o means u is authorised to *read* the contents of o.

In order to group together access permissions that should be assigned to the same user, or set of users, a set of 'security labels' may be defined. Users and objects/permissions are each assigned to some subset of these security labels; users then inherit the permissions associated to their set of assigned security labels. For example, if the permission (o, read) is assigned to a security label x, then we say that any user assigned to security label x is authorised to *read* object o. In some cases, such security labels may be ordered in a hierarchy, represented as a partial ordering $(L, \leqslant)$, where L is the set of labels and $\leqslant$ is an ordering on such labels. Then we may define *hierarchical* access control over $(L, \leqslant)$, such that if a user is assigned to some security label $l \in L$, then they inherit the access permissions assigned to all labels $l' \leqslant l$.

We now introduce three well-studied forms of access control policy: *information flow*, *role-based* and *attribute-based* access control policies.

2.2.1 Information Flow Policies

An *information flow policy* is a type of access control policy in which the set of security labels L are ordered in a hierarchy, represented as the poset $(L, \leqslant)$. Each user $u \in U$ and data object $o \in O$ is assigned to a *single* security label in $(L, \leqslant)$. More formally:

Definition 1. *An* information flow policy *[15] is a tuple* $((L, \leqslant), U, O, \lambda)$ *where:*

- $(L, \leqslant)$ *is a partially ordered set of security labels;*
- U *is a set of users;*
- O *is a set of data objects;*
- $\lambda : U \cup O \rightarrow L$ *is a security function mapping each user in* U *and data object in* O *to a security label in* L *(referred to as their clearance and classification level, respectively).*

The security property of an information flow policy states that a user $u \in U$ is authorised to *read* a data object $o \in O$ if and only if $\lambda(o) \leqslant \lambda(u)$ [15]. We write $U(l) = \{u \in U : \lambda(u) = l\}$ and $O(l) = \{o \in O : \lambda(o) = l\}$ to represent the sets of users and objects assigned to security label l.

The Bell-LaPadula [14, 15] and Biba [19] Models may be used to implement an information flow policy. The Bell-LaPadula Model enforces security whilst the Biba Model is used to enforce data integrity. Informally, the Bell-LaPadula model enforces a 'read down, write up' policy, which enables a user u to *read* an object o if $\lambda(u) \geqslant \lambda(o)$ and *write* to an object o if $\lambda(o) \geqslant \lambda(u)$.

2.2.2 Role-based Access Control Policies

Role-based access control was introduced [43, 80] in order to better model access control requirements (than more traditional mandatory and discretionary access control, MAC and DAC, policies [20]), and to support administrative tasks within the enforcement of access control policies (for example, the granting and revoking of access permissions).

In these policies, a set of *roles* is defined (where each role reflects a 'job title' or task within an organisation). Users are assigned to roles via a *user-assignment* operation, and permissions are also assigned to roles via a *permission-assignment* operation. An access request is *authorised* if both the user and the requested permission are assigned to some common role. Note that in contrast to information flow policies in which users and objects

are each assigned to a single security label, users and permissions in an RBAC policy may be assigned to many roles.

As argued by Ferraiolo *et al.* [43], such policies are suitable for modelling organisations in which:

- the set of roles are static, or change slowly over time;
- the 'function' or roles of its employees change frequently over time (i.e. user-assignment to roles changes frequently).

In such cases, the design of RBAC strongly supports administrative changes to the access rights of its users. For example, when a user joins an organisation, they can simply be assigned to roles which reflect their job title or tasks. If a user's role within an organisation changes, they can be deassigned from the roles to which they no longer are authorised, and be assigned to new roles. Thus, because such roles are typically so closely linked to job roles within an organisation, it should be clear, from an administrator's perspective, to identify which access permissions each user should/should not be authorised for.

There are four types of RBAC policy: Core RBAC (RBAC_0), Hierarchical RBAC (RBAC_1), Constrained RBAC (RBAC_2) and Symmetric RBAC (RBAC_3).

A *Core* role-based access control policy (RBAC_0) [78] is defined by the tuple (U, R, P, UA, PA) where:

- U is a set of users;
- R is a set of roles;
- $P \subseteq O \times A$ is a set of permissions, where O is a set of data objects and A is a set of access rights;
- $UA \subseteq U \times R$ (user-role assignment);
- $PA \subseteq P \times R$ (permission-role assignment).

Informally, R may be a set of job titles, security levels etc. We say that a user $u \in U$ is

authorised for a permission $p \in P$ if and only if there exists an $r \in R$ such that $(u, r) \in UA$ and $(p, r) \in PA$.

If we only consider *read* access, then we can instead define our RBAC_0 policy as the tuple (U, R, O, UA, OA) where O is our set of data objects and $OA \subseteq O \times R$ (object-role assignment). Then we say that a user $u \in U$ is authorised to *read* an object $o \in O$ if and only if there exists $r \in R$ such that $(u, r) \in UA$ and $(o, r) \in OA$.

Hierarchical RBAC (RBAC_1) enables role hierarchies. Namely, it enables a role r to inherit the permissions assigned to any role r' which is lower than r in the role hierarchy.

There exist two other types of role-based access control policies that are not required in this thesis, namely RBAC_2, and RBAC_3, which add functionality to RBAC_0 [78]. Informally:

- RBAC_2 (or *Constrained* RBAC) introduces role constraints (e.g. one can define mutually exclusive roles, such that a user can only be assigned to one of such roles);
- RBAC_3 (or *Symmetric* RBAC) combines RBAC_1 and RBAC_2 together to enable role constraints in hierarchical RBAC.

2.2.3 Attribute-based Access Control Policies

In role-based access control policies, we determine whether a user is authorised for a certain permission by seeing whether both the user and access permission have a role in common. However, we may want to add other restrictions on what permissions a user has and/or who can access specific data objects. For example, we may wish to restrict access to an object according to time, a user's location, department etc. Thus attribute-based access control was introduced in order to enable a more fine-grained level of access control. As mentioned by Ferraiolo *et al.* [43]:

"ABAC enables precise access control, which allows for a higher number of discrete inputs into an access control decision, providing a bigger set of possible combinations of those variables to reflect a larger and more definitive set of possible rules to express policies."

Informally, a set of attributes is defined over which access policies will be defined, which

can be represented as Boolean formulae over the set of attributes. We use $\wedge$ to denote the boolean operator 'AND' and $\vee$ the boolean operator 'OR'. Typically, an ABAC policy falls into one of two categories: *object-centric* and *user-centric*.

In an *object-centric* policy, each user is associated with a set of attributes and each data object is associated with an access policy. Then, a user u is authorised to access an object o if the set of attributes which u possesses satisfies the policy associated with object o. As an example, suppose user u has attributes $\{$*Student*, *Maths*, *Royal Holloway University*$\}$ and the object 'Maths lecture notes' is associated with the access policy *Maths* $\wedge$ *Royal Holloway University*. The, user u is authorised for 'Maths lecture notes' because u has both attributes *Maths* and *Royal Holloway University*. Suppose the object 'Maths exam results' is associated with the policy *Professor* $\wedge$ *Maths* $\wedge$ *Royal Holloway University*. Then u is not authorised for 'Maths exam results' since their set of attributes does not satisfy the policy associated to 'Maths exam results'.

In a *user-centric* policy, each data object is associated with a set of attributes, and each user is associated with an access policy. Then a user u is authorised for object o if the attributes associated to o satisfy u's access policy. For example, we could define the object 'Math exam results' as having the attributes *Professor*, *Maths*, *Royal Holloway University* and a user u as having the policy *Maths* $\wedge$ *Student* $\wedge$ *Royal Holloway University*. Then u is not authorised for 'Maths exam results' since the attributes associated to 'Maths exam results' do not satisfy u's policy (for this to be true, 'Maths exam results' must also possess the attribute *Student*, for example).

More formally, an attribute-based access control (ABAC) policy can be defined by the tuple $(U, P, \mathcal{U}, \lambda, \phi)$ where:

- U is a set of users;
- $P \subseteq O \times A$ is a set of permissions, where O is a set of data objects and A is a set of access rights;
- $\mathcal{U}$ is a set of attributes;
- $\lambda : U \cup P \to 2^{\mathcal{U}}$ is a function mapping users or permissions to a set of attributes;
- $\phi : U \cup P \to 2^{2^{\mathcal{U}}}$ is a function mapping users or permissions to access structures over $\mathcal{U}$.

In a *user-centric* policy, each user is associated with a set of attributes $\overline{A} \subseteq \mathcal{U}$, and each permission is associated with an access structure $\mathbb{A} \in 2^{2^{\mathcal{U}}}$. A user $u \in U$ is authorised for a permission p if their attributes satisfy the policy associated to p. Thus we define $\lambda : U \to 2^{\mathcal{U}}$ and $\phi : P \to 2^{2^{\mathcal{U}}}$.

In an *object-centric* policy, each object (permission) is associated with a set of attributes $\overline{A} \subseteq \mathcal{U}$, and each user is associated with an access structure; a user u is authorised for permission p if $\overline{A}$ satisfies the access structure associated with u. Thus, for object-centric policies, $\lambda : P \to 2^{\mathcal{U}}$ and $\phi : U \to 2^{2^{\mathcal{U}}}$.

2.3 Encryption Schemes

We now give formal definitions for symmetric and asymmetric (public key) encryption schemes. We then also provide a formal definition of large attribute-universe key policy attribute-based encryption (KP-ABE) schemes, which we will use in Chapter 6. Whilst alternative attribute-based encryption schemes exist, for example small attribute-universe KP-ABE schemes [58] and ciphertext policy attribute-based encryption (CP-ABE) schemes [18], such schemes will not be required in this thesis.

2.3.1 Symmetric Encryption Schemes

A symmetric encryption scheme $\mathcal{SE}$ [65] with key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is a triple of polynomial time algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where:

- $\kappa \xleftarrow{\$} \mathsf{Gen}(1^\rho)$ is a randomised key generation algorithm that takes as input a security parameter 1^ρ and outputs a key $\kappa \in \mathcal{K}$;
- $c \xleftarrow{\$} \mathsf{Enc}_\kappa(m)$ is a randomised encryption algorithm that takes as input a key $\kappa \in \mathcal{K}$ and a message $m \in \mathcal{M}$ and outputs a ciphertext $c \in \mathcal{C}$;
- $(m \cup \perp) \leftarrow \mathsf{Dec}_\kappa(c)$ is a deterministic algorithm that takes as input a key $\kappa \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ and outputs a message $m \in \mathcal{M}$ or a special reject symbol $\perp$.

$\mathbf{Exp}^{\text{lor-cpa-b}}_{\mathcal{SE},\mathcal{A}}(1^\rho)$	Oracle LOR(m_0, m_1)
$k \xleftarrow{\$} \mathsf{Gen}(1^\rho)$	**if** $\lvert m_0\rvert \neq \lvert m_1\rvert$:
$b' \leftarrow \mathcal{A}^{\mathcal{O}}(1^\rho)$	**return** $\perp$
return $b' = b$	**return** $\mathsf{Enc}_k(m_b)$

Figure 2.2: LOR IND-CPA experiment for symmetric encryption scheme $\mathcal{SE}$.

Correctness [65]. A symmetric encryption scheme $\mathcal{SE}$ is said to be *correct* if for every $\rho \in \mathbb{N}$, every $\kappa \in \mathcal{K}$ output by $\mathsf{Gen}(1^\rho)$ and every $m \in \mathcal{M}$,

$$\Pr[\mathsf{Dec}_\kappa(\mathsf{Enc}_\kappa(m)) = m] = 1.$$

Security [16]. The security of a symmetric encryption scheme $\mathcal{SE}$ can be formalised through the indistinguishability against a chosen-plaintext attack (IND-CPA) experiment shown in Figure 2.2. Here, the experiment is presented in the left-or-right (LOR) format. A find-then-guess (FTG) version of the experiment also exists, however the adversary is allowed multiple challenges in the LOR version and only one in the FTG version, thus we opt for the LOR version here. The experiment is played between a challenger and an adversary $\mathcal{A}$. In this experiment, the challenger randomly generates a key κ via Gen and gives $\mathcal{A}$ access to a left-or-right (LOR) encryption oracle, denoted $\mathcal{O}$. Each time $\mathcal{A}$ makes a query to the LOR oracle with inputs (m_0, m_1), where m_0, m_1 are two messages in $\mathcal{M}$, the challenger will check that m_0 and m_1 are of the same length and, if so, returns $\mathsf{Enc}_\kappa(m_b)$ to the adversary. After polynomially many (in the security parameter) queries to the LOR oracle, $\mathcal{A}$ submits his guess b' of b (i.e. whether m_0 or m_1 was encrypted).

The objective of the adversary $\mathcal{A}$ in this experiment is to try to learn something about the plaintext contents of a ciphertext c for which he does not possess the (decryption) key. If he is able to distinguish which of two messages m_0, m_1 have been encrypted to c, then $\mathcal{A}$ must have been able to learn something from the ciphertext c that enabled him to distinguish such messages.

Informally, we say that a symmetric encryption scheme $\mathcal{SE}$ is IND-CPA secure if an adversary cannot distinguish which of the messages was encrypted with probability significantly better than guessing ($\frac{1}{2}$).

We define the advantage of an adversary $\mathcal{A}$ in the experiment $\mathbf{Exp}^{\mathsf{lor-cpa-b}}_{\mathcal{SE},\mathcal{A}}(1^\rho)$ above as:

$$\mathrm{Adv}^{\mathsf{lor-cpa}}_{\mathcal{SE},\mathcal{A}}(1^\rho) = \left|\Pr\left[\mathbf{Exp}^{\mathsf{lor-cpa-1}}_{\mathcal{SE},\mathcal{A}}(1^\rho) \to 1\right] - \Pr\left[\mathbf{Exp}^{\mathsf{lor-cpa-0}}_{\mathcal{SE},\mathcal{A}}(1^\rho) \to 1\right]\right|.$$

We say that $\mathcal{SE}$ is IND-CPA secure (in the LOR version of the experiment) if:

$$\mathrm{Adv}^{\mathsf{lor-cpa}}_{\mathcal{SE},\mathcal{A}}(1^\rho) \leqslant \mathsf{negl}(\rho).$$

2.3.2 Asymmetric Encryption Schemes

An asymmetric (public key) encryption scheme $\mathcal{AE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with keyspace $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ consists of the following three algorithms [65]:

- $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\rho)$ is a randomised key generation algorithm that takes as input a security parameter 1^ρ and returns a public-private key pair (pk, sk) where $pk, sk \in \mathcal{K}$;
- $c \xleftarrow{\$} \mathsf{Enc}_{pk}(m)$ takes as input a public key $pk \in \mathcal{K}$ and plaintext message $m \in \mathcal{M}$ and outputs a ciphertext $c \in \mathcal{C}$;
- $(m \cup \perp) \leftarrow \mathsf{Dec}_{sk}(c)$ is a deterministic decryption algorithm that takes as input a secret key $sk \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ and returns a message $m \in \mathcal{M}$ or a distinguished failure symbol $\perp$ otherwise.

Correctness [65]. A public key encryption scheme $\mathcal{AE}$ is said to be *correct* if, for all $\rho \in \mathbb{N}$, all (pk, sk) output by $\mathsf{Gen}(1^\rho)$ and for all messages $m \in \mathcal{M}$:

$$\Pr\left[\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m\right] = 1.$$

Security [65]. Security of a public key encryption scheme $\mathcal{AE}$ may be formalised through the LOR IND-CPA experiment $\mathbf{Exp}^{\mathsf{lor-cpa-b}}_{\mathcal{AE},\mathcal{A}}(1^\rho)$ shown in Figure 2.3. This security game is similar to the LOR IND-CPA experiment for symmetric encryption schemes shown in Figure 2.2, except that the keygen algorithm Gen for $\mathcal{AE}$ produces a public-private key pair (pk, sk), the adversary $\mathcal{A}$ gets the public key pk (and not the secret key sk), and the LOR encryption oracle (shown in Figure 2.3) returns the message m_b encrypted under pk

(provided $|m_0| = |m_1|$).

$\mathbf{Exp}^{\mathrm{lor-cpa-b}}_{\mathcal{AE},\mathcal{A}}(1^\rho)$	Oracle LOR(m_0, m_1)
$(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\rho)$	**if** $\lvert m_0\rvert \neq \lvert m_1\rvert$:
$b' \leftarrow \mathcal{A}^{\mathcal{O}}(1^\rho, pk)$	**return** $\perp$
return $b' = b$	**return** $\mathsf{Enc}_{pk}(m_b)$

Figure 2.3: LOR IND-CPA experiment for public key encryption scheme $\mathcal{AE}$ [65].

We define the advantage of an adversary $\mathcal{A}$ in the experiment $\mathbf{Exp}^{\mathsf{lor-cpa-b}}_{\mathcal{AE},\mathcal{A}}(1^\rho)$ as:

$$\mathrm{Adv}^{\mathsf{lor-cpa}}_{\mathcal{AE},\mathcal{A}}(1^\rho) = \left|\Pr\left[\mathbf{Exp}^{\mathsf{lor-cpa-1}}_{\mathcal{AE},\mathcal{A}}(1^\rho) \to 1\right] - \Pr\left[\mathbf{Exp}^{\mathsf{lor-cpa-0}}_{\mathcal{AE},\mathcal{A}}(1^\rho) \to 1\right]\right|.$$

We say that $\mathcal{AE}$ is (LOR) IND-CPA secure if:

$$\mathrm{Adv}^{\mathsf{lor-cpa}}_{\mathcal{AE},\mathcal{A}}(1^\rho) \leqslant \mathsf{negl}(\rho).$$

2.3.3 Key Policy Attribute-based Encryption

A key policy attribute-based encryption (KP-ABE) [58, 71] scheme $\mathcal{ABE}$ comprises the algorithms $\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{KeyGen}$ and $\mathsf{Decrypt}$ with attribute-universe $\mathcal{U}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ and key space $\mathcal{K}$. A large attribute-universe scheme permits arbitrary strings to be used as attributes i.e. $\mathcal{U} = \{0,1\}^\star$. The algorithms are defined as follows:

- $(MK, PP) \xleftarrow{\$} \mathsf{Setup}(1^\rho)$ takes as input a security parameter 1^ρ and outputs a master secret $MK \in \mathcal{K}$ and public parameters PP;
- $c \xleftarrow{\$} \mathsf{Encrypt}(m, \gamma, PP)$ takes as input a message $m \in \mathcal{M}$, a set of attributes $\gamma \subseteq \mathcal{U}$ and public parameters PP output by Setup, and outputs a ciphertext $c \in \mathcal{C}$;
- $\kappa_{\mathbb{A}} \xleftarrow{\$} \mathsf{KeyGen}(\mathbb{A}, MK, PP)$ takes as input an access structure (policy) $\mathbb{A} \in 2^{2^{\mathcal{U}}}$, the master secret key $MK \in \mathcal{K}$ and public parameters PP, and outputs a (decryption) key $\kappa_{\mathbb{A}} \in \mathcal{K}$ for the policy $\mathbb{A}$; and
- $(m \cup \perp) \leftarrow \mathsf{Decrypt}(c, \kappa_{\mathbb{A}}, PP)$ takes as input a ciphertext $c \in \mathcal{C}$ (an encryption of $m \in \mathcal{M}$ under attribute set $\gamma \subseteq \mathcal{U}$), a decryption key $\kappa_{\mathbb{A}} \in \mathcal{K}$ for policy $\mathbb{A}$ and public parameters PP output by Setup. It outputs a message $m \in \mathcal{M}$ if $\gamma \in \mathbb{A}$ (the set of attributes γ satisfies the policy $\mathbb{A}$) or $\perp$ otherwise.

Correctness. We say a (large attribute-universe) KP-ABE scheme $\mathcal{ABE}$ is *correct* if for all $\rho \in \mathbb{N}$, all $MK \in \mathcal{K}$ and PP output by $\mathsf{Setup}(1^\rho)$, all messages $m \in \mathcal{M}$, all access structures $\mathbb{A} \in 2^{2^{\mathcal{U}}}$, all attribute sets $\gamma \subseteq \mathcal{U}$ that satisfy $\mathbb{A}$ and all keys $\kappa_{\mathbb{A}}$ output by $\mathsf{KeyGen}(\mathbb{A}, MK, PP)$:

$$\Pr\left[\mathsf{Decrypt}(\mathsf{Encrypt}(m, \gamma, PP), \kappa_{\mathbb{A}}, PP) = m\right] = 1.$$

Security. The security of a (large attribute-universe) KP-ABE scheme $\mathcal{ABE}$ can be formalised through an indistinguishability against a chosen plaintext attack (IND-CPA) game. The *fully secure* [69, 70] find-then-guess (FTG) IND-CPA experiment for KP-ABE schemes is shown in Figure 2.4 [2].

$\mathbf{Exp}^{\mathrm{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}}(1^\rho)$

$(MK, PP) \xleftarrow{\$} \mathcal{ABE}.\mathsf{Setup}(1^\rho)$
$X_{ftg} \leftarrow \emptyset$
$A^\star \leftarrow \emptyset$
$(m_0, m_1, A') \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\rho, PP)$
if $|m_0| \neq |m_1|$:
  return False
for $\mathbb{A} \in X_{ftg}$:
  if $A' \in \mathbb{A}$:
    return False
$A^\star \leftarrow A'$
$c \xleftarrow{\$} \mathcal{ABE}.\mathsf{Encrypt}(m_b, A^\star, PP)$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(c)$
return $b' = b$

Oracle KeyGen($\mathbb{A}$)

if $A^\star \in \mathbb{A}$:
  return $\perp$
$X_{ftg} \leftarrow X_{ftg} \cup \{\mathbb{A}\}$
return $\mathcal{ABE}.\mathsf{KeyGen}(\mathbb{A}, MK, PP)$

Figure 2.4: Fully secure find-then-guess IND-CPA security experiment for (large attribute-universe) KP-ABE Schemes.

The experiment is played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. Informally, in this experiment, the adversary $\mathcal{A}$ may encrypt messages of his choosing (since it has access to the public parameters PP) and is given oracle access granting the ability to query for decryption keys associated to certain access structures of its choosing. $\mathcal{A}$ asks the challenger $\mathcal{C}$ to (randomly) encrypt one of two messages m_0, m_1 under a set of attributes A' (which $\mathcal{A}$ also chooses) *provided* that A' does not satisfy the access structure associated

[2] Another version of the FTG IND-CPA experiment is the *selective* version in which the adversary must select their challenge prior to playing the game. We, however, opt for the *fully* secure version which gives the adversary more power and allows them to dynamically choose their challenge. Furthermore, a scheme proven *fully* secure is argued to be more secure than one which is *selectively* secure.

to any decryption key which $\mathcal{A}$ possesses. The aim of the adversary in this game is to try to guess which message m_b was encrypted by the challenger. Informally, we say that if the adversary wins the game with non-negligible advantage then he was able to learn something about the contents of data for which he is not authorised, from observing ciphertexts and other keys. Since we desire that unauthorised users cannot learn anything about data for which they do not possess a valid decryption key, we say that a KP-ABE scheme is IND-CPA secure if the advantage of an adversary in the experiment shown in Figure 2.4 is negligible.

In this fully secure find-then-guess (FTG) experiment, the adversary $\mathcal{A}$ runs in two phases: the *find* and *guess* phases. During the *find* phase, the adversary has not yet chosen a challenge (i.e. chosen m_0, m_1, A') and can query the KeyGen oracle (polynomially many times) for a decryption key associated to an access structure $\mathbb{A}$ of his choosing. For each such query, the challenger $\mathcal{C}$ adds $\mathbb{A}$ to a list X_{ftg}, which helps $\mathcal{C}$ track which access structures the adversary currently has a decryption key for. After the adversary has sent his challenge messages m_0, m_1 and attribute set A' to $\mathcal{C}$, we now say the adversary is in his *guess* phase. During this phase, the adversary can continue to query the KeyGen oracle (polynomially many times) for decryption keys; however, such queries are only valid if A' does not satisfy the access structure for which the adversary is requesting the key (else this would lead to a trivial win). The adversary then submits his guess b' of b. Note that the game fails if the adversary sends a challenge with attribute set A' to $\mathcal{C}$, where A' satisfies an access structure in X_{ftg} (since $\mathcal{A}$ would already possess a valid decryption key for this challenge, leading to a trivial win).

The advantage $\mathrm{Adv}^{\mathrm{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho)$ of an adversary $\mathcal{A}$ in the fully secure find-then-guess IND-CPA experiment shown in Figure 2.4 is defined as [69]:

$$\mathrm{Adv}^{\mathrm{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) = \left|\Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa}-1}_{\mathcal{ABE},\mathcal{A}}(1^\rho) \to 1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa}-0}_{\mathcal{ABE},\mathcal{A}}(1^\rho) \to 1\right]\right|.$$

Definition 2. *A (large attribute-universe) key-policy attribute-based encryption scheme $\mathcal{ABE}$ is (fully) secure (in the* find-then-guess *case) if for all PPT adversaries $\mathcal{A}$, for all $\rho \in \mathbb{N}$:*

$$\mathrm{Adv}^{\mathrm{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) \leqslant \mathsf{negl}(\rho).$$

2.4 Pseudorandom functions

In order to define a pseudorandom function, we will first define a random function.

Let $\langle \mathcal{D} \to \mathcal{R} \rangle$ be the set of all functions with domain $\mathcal{D}$ and range $\mathcal{R}$. Then, given $\langle \mathcal{D} \to \mathcal{R} \rangle$, a *random function* is a function φ chosen uniformly at random from $\langle \mathcal{D} \to \mathcal{R} \rangle$ [17]. We denote choosing a random function φ from $\langle \mathcal{D} \to \mathcal{R} \rangle$ as $\varphi \xleftarrow{\$} \langle \mathcal{D} \to \mathcal{R} \rangle$.

Definition 3. *Consider a family of efficiently computable functions $\mathcal{F}\colon \mathcal{K} \times \{0,1\}^* \to \mathcal{R}$ indexed by a keyspace $\mathcal{K}$ and each with range $\mathcal{R}$. We write $\mathcal{F}_\kappa(x)$ to denote $\mathcal{F}(\kappa, x)$, where $\mathcal{F}_\kappa(\cdot)$ is an* instance *of $\mathcal{F}$ keyed under κ. We say that $\mathcal{F}$ is a* pseudorandom function *(PRF) if the* advantage *of all PPT adversaries $\mathcal{A}$ in distinguishing $\mathcal{F}$ from a random function (chosen uniformly at random from the set of all functions with the same domain $\mathcal{D} = \{0,1\}^\rho$ and range $\mathcal{R}$) is negligible. More formally, let us define the distinguishing advantage of an adversary $\mathcal{A}$ to be:*

$$\mathrm{Adv}^{\mathsf{ind-prf}}_{\mathcal{F},\mathcal{A}}(1^\rho) = \left| \Pr[\kappa \xleftarrow{\$} \{0,1\}^\rho ; \mathcal{A}^{\mathcal{F}_\kappa} \to 1] - \Pr[\varphi \xleftarrow{\$} \langle \{0,1\}^\rho \to \mathcal{R} \rangle ; \mathcal{A}^{\varphi} \to 1] \right| .$$

We say that $\mathcal{F}$ is indistinguishable from a random function if, for all PPT adversaries $\mathcal{A}$,

$$\mathrm{Adv}^{\mathsf{ind-prf}}_{\mathcal{F},\mathcal{A}}(1^\rho) \leqslant \mathsf{negl}(\rho).$$

In the definition above, writing "$\mathcal{A}^f \to 1$" for a function f means that adversary $\mathcal{A}$ has oracle access to f and terminates outputting value 1. In Definition 3, an adversary either has access to a keyed PRF instance $\mathcal{F}_\kappa$ or a completely random function φ.

In this thesis, we only consider PRFs whose keyspace and range are the same set, i.e. $\mathcal{R} = \mathcal{K} = \{0,1\}^\rho$.

2.5 Key Assignment Schemes

A natural way to enforce a read-only information flow policy $((L, \leqslant), U, O, \lambda)$ is to define a symmetric cryptographic key κ_l for each $l \in L$, encrypt each data object $o \in O$ with $\kappa_{\lambda(o)}$ and give each user $u \in U$ all keys κ_l such that $l \leqslant \lambda(u)$. Unfortunately, the number

of keys that each user requires may be large, and thus we desire some method of reducing the amount of cryptographic material a user is required to store.

In such situations, a (symmetric) key assignment scheme (KAS) [6, 36, 83] may be used in order to reduce the amount of key material required by each user. Informally, a KAS provides functionality to enable a trusted authority to generate a key κ_l and a small amount of secret material σ_l for each label $l \in L$, some public information Pub, and a mechanism by which keys can be derived from secret and public material. The cryptographic keys produced in a KAS can then be used within a cryptographic enforcement scheme to protect sensitive data objects.

KASs typically represent the policy's underlying poset as a directed acyclic graph [6, 32, 33, 34, 36, 82, 83] and enable iterative key derivation along paths in that graph. Users are issued (small) user secrets and users with security label x can derive the key associated to a security label $y \leqslant x$ using the user secret associated with x and public information associated with arcs in a path from x to y. As an example, Figure 2.5 shows the Hasse diagram of a poset $(L, \leqslant)$ where each arc xy is labelled with a piece of public information which enables a user with knowledge of κ_x to derive κ_y (the iterative scheme described in Section 2.5.3 [36]). Then, considering the policy $((L, \leqslant), U, O, \lambda)$, $u \in U$ is given $\kappa_{\lambda(u)}$ and can iteratively derive keys for which they are authorised.

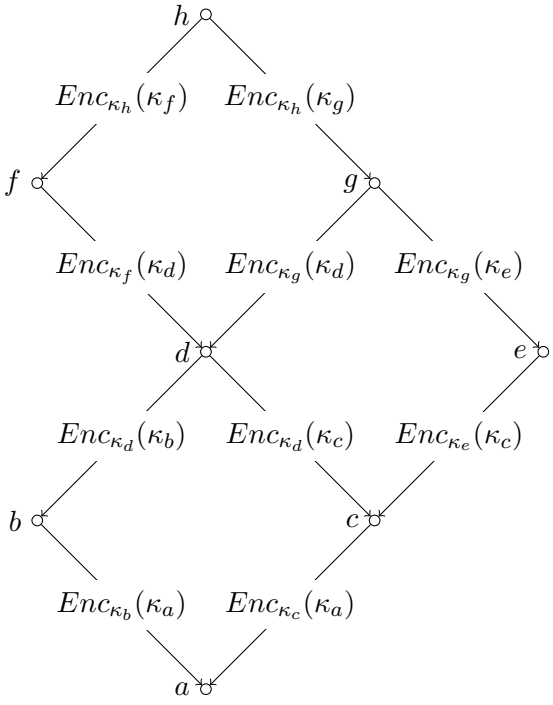


Figure 2.5: Example of key derivation.

Research into the use of cryptographic key assignment schemes for dealing with key management issues when enforcing hierarchical access control began with the seminal work of Akl and Taylor [3]. Since this work, numerous KASs have been proposed in the literature [6, 8, 31, 32, 33, 34, 47, 48, 68, 81, 83]. For example, Ferrara *et al.* [6] introduced a key assignment scheme where keys are derived iteratively using public information. Ateniese *et al.* provided a framework for (and provided instantations of) time-based key assignment schemes. The intention of such schemes is to only authorise users to access certain files during a specific time period. Key assignment schemes for geo-spatial systems have also been proposed enable users to derive keys associated to locations for which they were authorised for. Crampton *et al.* [36] provide a comparison of various KASs in the literature.

2.5.1 KAS Definition

A KAS comprises of two algorithms: Setup and Derive [6]. Given the information flow policy poset $(L, \leqslant)$, a setup authority calls Setup to generate, for each label $l \in L$, a unique key κ_l and secret material σ_l, along with public information Pub to help users to derive keys for which they are authorised. As an example, Section 2.5.3 describes a scheme (see the example direct scheme) in which Pub contains $Enc_{\kappa_x}(\kappa_y)$ for each pair of labels $x, y \in L$ such that $y < x$ in the policy poset; thus a user can derive κ_y only if they possess κ_x for some label $x > y$.

We now provide a formal definition of a KAS.

Definition 4. *A (symmetric)* key assignment scheme *(KAS) for a poset* $\mathcal{P} = (L, \leqslant)$ *comprises the algorithms* $(\mathsf{Setup}, \mathsf{Derive})$ *where:*

- $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, (L, \leqslant))$ *is a probabilistic polynomial-time algorithm run by a setup authority that takes as input a security parameter* 1^ρ *and poset* $(L, \leqslant)$ *and outputs a user secret* σ_l *and symmetric key* κ_l *for each* $l \in L$*, along with a set of public information* Pub*;*
- $(\kappa_y \cup \perp) \leftarrow \mathsf{Derive}((L, \leqslant), x, y, \sigma_x, Pub)$ *is a deterministic polynomial-time algorithm run by a user to derive* κ_y *from the user secret* σ_x*. It takes as input* $(L, \leqslant)$*, labels* $x, y \in L$*, the secret* σ_x*, and public information* Pub*, and outputs the derived key* κ_y

assigned to label y if $y \leqslant x$, and outputs $\perp$ otherwise (i.e. when the derivation is unauthorised).

A representation of the policy poset is required as input to the Derive algorithm [6]. Hence in schemes in which users must derive their set of keys, the data owner may need to publish the policy poset (or distribute it to every user). The size of the policy is typically proportional to the number of arcs in the graphical representation of the poset (each arc representing a piece of public information) used to define key/secret derivation; that is $\mathcal{O}(n^2)$, where $n = |L|$ (the number of security labels). In the case of edge-based schemes (see Section 2.5.3), the data owner may also publish (or otherwise distribute) a piece of public information for every arc in the graphical representation of the policy poset in order to support derivation along such arcs. We will use Pub_{der} to denote this set of public derivation information. Note that the size of Pub_{der} will be several orders of magnitude bigger than the policy representation (due to the relative sizes of each datum of information).

We will define Pub to contain Pub_{der} used to support key derivation, and potentially the poset and global parameters. Thus when we discuss schemes that eliminate public information in Chapters 3 and 4, we actually mean that such schemes eliminate Pub_{der}, since both schemes assume that the policy poset is public.

Definition 5 (Correctness [47])**.** *A KAS is* correct *if for all $\rho \in \mathbb{N}$, all $(L, \leqslant)$, all $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub)$ output by $\mathsf{Setup}(1^\rho, (L, \leqslant))$, and all $x, y \in L$ such that $y \leqslant x$:*

$$\Pr[\kappa'_y \leftarrow \mathsf{Derive}((L, \leqslant), x, y, \sigma_x, Pub) : \kappa'_y = \kappa_y] = 1.$$

2.5.2 Security

Atallah *et al.* [6] introduced two formal security notions for key assignment schemes: *key recovery* (KR) and *key indistinguishability* (KI). Intuitively, a scheme is secure against *key recovery* if an adversary, representing a group of colluding users, cannot learn a key for which none of the colluding users were authorised. In the interests of integrating a KAS with other cryptographic schemes that require keys to be uniformly distributed (e.g. IND-CPA secure symmetric encryption schemes, described in Section 2.3.1), the stronger notion of *key indistinguishability* was introduced [6, 48]. Key indistinguishability is a

stronger security notion and specifies that the adversary cannot learn anything about a key for which none of the colluding users were authorised (other than its length). D'Arco *et al.* [39] show that the scheme proposed by Akl and Taylor [3] is secure against key recovery. Furthermore, they describe how to transform a KR secure KAS into one that is KI secure.

Freire *et al.* introduced another security notion, *strong key indistinguishability* (SKI) [48], which is similar to KI except that, in the security game, the adversary is given more information than in the KI security game. In the KI game, the adversary is allowed to query for polynomially (in the security parameter) many secrets/keys for labels which are not higher than (or equal to) a challenge label chosen by the adversary. In the SKI game however, the adversary is given the keys for *all* non-challenge labels (including keys for labels above the challenge label), and secrets for all security labels that do not enable the adversary to derive the challenge key (i.e. for all labels not greater than or equal to the challenge label). Whilst it seems that SKI is a stronger notion than KI, because the adversary is given more information, Castiglione *et al.* [25] showed that SKI and KI are *polynomially* equivalent (i.e. one could move from the SKI to KI game with tightness loss $n = |L|$).

In this thesis, we show that all our proposed key assignment schemes satisfy strong key indistinguishability. Not only is this security notion the strongest for KASs, but we believe that it is the most natural one [35] since it also models keys for higher labels being leaked through exposure [48], for example. It can also be more useful to use the SKI game in comparison to the KI game in reductive proofs (see Chapter 6 for example). We thus define the security experiment for strong key indistinguishability, $\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$ in Figure 2.6. Note that we provide the static notion of security in Figure 2.6 in which we consider a static setup and the challenge label is fixed a priori. A variant of Definition 6 would consider dynamic adversaries: such an adversary is able to choose the challenge label x *during* the experiment, rather than having it fixed as one of the experiment's parameters. However, it has been shown that static and dynamic definitions of strong key indistinguishability are polynomially equivalent [48]; corresponding results for (plain) key indistinguishability have also been obtained [9]. To simplify the exposition, therefore, we restrict our attention to the static case. Before we formally introduce the experiment, we

need to first introduce some notation. Given a finite poset $(L, \leqslant)$ and $x \in L$, we define:

$$Corrupt_x = \{(l, \sigma_l) : l \in L, x \not\leqslant l\},$$
$$Keys_x = \{(l, \kappa_l) : l \in L \setminus \{x\}\}.$$

In the experiment we assume that the adversary receives the information flow policy poset $(L, \leqslant)$ in the same format as the Setup algorithm does. In the experiment, the adversary $\mathcal{A}$ selects a challenge label $x \in L$. The challenger then calls SetUp on the security parameter 1^ρ and poset $(L, \leqslant)$ to generate a secret and key for each $l \in L$, and Pub to support key derivation. The challenger randomly selects a string from the keyspace and assigns it to $\kappa_0^\star$, and assigns $\kappa_1^\star \leftarrow \kappa_x$. The challenger gives the adversary the security parameter 1^ρ, poset $(L, \leqslant)$, $Corrupt_x, Keys_x$, Pub and either the random string $\kappa_0^\star$, if $b = 0$ or the key for label x, $\kappa_1^\star = \kappa_x$ if $b = 1$. The adversary, with access to the Derive algorithm, must then submit his guess b' of the value of b. The adversary wins if $b' = b$.

Intuitively, the adversary represents a group of colluding users trying to learn something about a key for which none of the users are authorised; this is modelled by giving the adversary the secret and key for all labels $l \in L$ where $l \not\geqslant x$ and x is the challenge label, since in practice such secrets and keys could be given to the adversary by colluding users. Additionally, the leakage of keys over time is also modelled in the SKI experiment by giving the adversary the keys for *all* non-challenge labels. Then, informally, if the adversary is able to learn something about the key for his challenge label x (for which none of the colluding users whom he represents are authorised), then he should be able to distinguish whether he was given κ_x or a random string in the experiment. Thus, we argue that if the adversary cannot do better than to 'guess' whether he was given the real or random key, then he cannot learn anything about κ_x.

$\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$:

$(\{\sigma_l, \kappa_l\}_{l \in L}, Pub) \xleftarrow{\$} \mathsf{SetUp}(1^\rho, (L, \leqslant))$
$\kappa_0^\star \xleftarrow{\$} \mathcal{K}, \kappa_1^\star \leftarrow \kappa_x$
$Corrupt_x \xleftarrow{\$} \{(l, \sigma_l) : l \in L, x \not\leqslant l\}$
$Keys_x \xleftarrow{\$} \{(l, \kappa_l) : l \in L \setminus \{x\}\}$
$b' \xleftarrow{\$} \mathcal{A}(1^\rho, (L, \leqslant), x, \kappa_b^\star, Corrupt_x, Keys_x, Pub)$
return $b' = b$

Figure 2.6: Security experiment for strong key indistinguishability.

Definition 6. *Let $(L, \leqslant)$ be an arbitrary poset. A KAS $\mathcal{KAS}$ for $(L, \leqslant)$ is* strongly key

indistinguishable with respect to static adversaries *[48] if, for all $x \in L$, the advantage of all PPT adversaries $\mathcal{A}$ that interact in experiment* $\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$ *is negligible in ρ, where we define* $\mathrm{Adv}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$ *to be*

$$\left|\Pr\left[\mathbf{Exp}^{\mathsf{SKI-1}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x) \to 1\right] - \Pr\left[\mathbf{Exp}^{\mathsf{SKI-0}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x) \to 1\right]\right|.$$

Observe that in this definition the adversary obtains, in principle, all secrets embedded in the system (that is, all σ_x and κ_x values), excluding only those that would allow distinguishing the challenge key by trivial means (e.g. by invoking the Derive algorithm).

In order for schemes to achieve KI (SKI) security, there must be a separation between the secret material used to derive keys, and the keys themselves. In other words, keys for labels lower in the policy should not be derivable from keys for higher security labels. If this was not the case, then an adversary can trivially win the KI (SKI) game. To see why, suppose there exists a security label l such that $l < x$, where x is the challenge label and $\kappa_l = F(\kappa_x)$, where F is some public function. Then, in the security game, the adversary is given either κ_x or some random bit string in the keyspace $\mathcal{K}$, and also obtains κ_l as part of $Keys_x$. Then, the adversary can trivially win the security game by computing $F(\kappa_x)$; if this is equal to κ_l then the adversary knows (with high probability) that they were given κ_x and thus returns 1. If $F(\kappa_x) \neq \kappa_l$, then the adversary knows that they were given some randomly generated bit string and thus returns 0. Hence if keys for lower labels can be derived from keys for higher labels, the scheme can not be KI secure.

Thus in order to make a KAS KI secure, an *intermediate secret* s_l is typically defined for each label $l \in L$, which can be used to derive secrets for labels lower in the access hierarchy and derive the key κ_l.[3] In traditional KI schemes, such as the KI scheme by Atallah *et al.* [6], σ_l for each $l \in L$ is simply the value s_l. However, in more recent schemes such as the schemes proposed by Crampton *et al.* [34, 33] and Freire *et al.* [48], each secret σ_l may comprise of multiple intermediate secrets, i.e. $\sigma_l \subseteq \{s_l\}_{l \in L}$. Thus, when comparing schemes in terms of the size of the user secrets, we define the size of a user secret for label $l \in L$, $|\sigma_l|$ to be the number of intermediate values s_l and keys κ_l it contains. We also make the assumption that each key and intermediate secret in a user secret is labelled with its corresponding security label. Thus when we write $\sigma_x \subseteq \{s_l\}_{l \in L}$ for example, we

[3] In contrast, in non-KI schemes, keys can be derived from other keys and thus intermediate secrets are not required.

actually mean $\sigma_x \subseteq \{(l, s_l)\}_{l \in L}$.

2.5.3 Classes of KAS

We provide an overview and comparison of different types of key assignment schemes that have been proposed in the literature. An extensive comparison of types of key assignment schemes is provided in a survey paper by Crampton *et al.* [36].

Trivial KAS. In a trivial KAS [36, 46], each user $u \in U$ is given all keys for which they are authorised i.e. $\sigma_{\lambda(u)} = \{\kappa_l : l \leqslant \lambda(u)\}$. Considering the poset shown in Figure 2.7, for example, a user assigned label f would be given $\sigma_f = \{\kappa_a, \kappa_b, \kappa_c, \kappa_d, \kappa_f\}$. Key derivation is trivial and no public information is required since each user is given all the keys for which they are authorised.

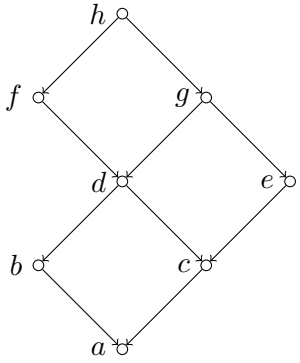


Figure 2.7: Hasse diagram of a simple poset.

Direct KAS. A *direct* KAS enables users to derive any of their authorised keys in *at most one* derivation step. In the direct scheme described by Crampton *et al.* [36], a piece of public information is associated with every arc in the transitive closure of the Hasse diagram of the poset underlying the policy, e.g. $Pub_{der} = \{Enc_{\kappa_x}(\kappa_y) : x, y \in L, y < x\}$ and each user $u \in U$ is given $\kappa_{\lambda(u)}$. Figure 2.8 shows a subset of the arcs in the transitive closure of the Hasse diagram in Figure 2.7, namely those that start at node h (we omit all other arcs in the transitive closure for clarity). Then a user assigned to label h can derive the key for a label y in the set $\{a, b, c, d, e, f, g\}$ by using κ_h to decrypt the piece of public information $Enc_{\kappa_h}(\kappa_y)$ associated to the arc from h to y. The scheme proposed by Gude [59] is another example of a direct scheme.

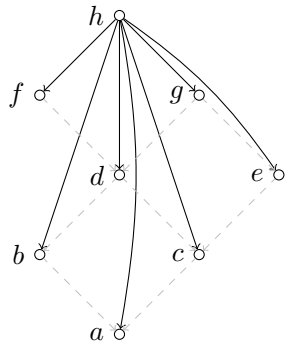


Figure 2.8: Subset of arcs in the transitive closure of Figure 2.7.

Iterative KAS. An *iterative* KAS enables a user to iteratively derive keys from their given user secret along paths in a defined graphical representation of the policy poset (e.g. the poset's Hasse diagram). In the scheme described by Crampton *et al.* [36], each user u is given a single secret $\sigma_{\lambda(u)} = \kappa_{\lambda(u)}$ and each arc xy in the Hasse diagram is associated with a piece of public information $Enc_{\kappa_x}(\kappa_y)$, i.e. $Pub_{der} = \{Enc_{\kappa_x}(\kappa_y) : x, y \in L, y \lessdot x\}$. Consider again the Hasse diagram in Figure 2.7. Then a user assigned label e can iteratively derive the key for label a by first deriving κ_c (by decrypting $Enc_{\kappa_e}(\kappa_c) \in Pub$ using κ_e) and then using κ_c to derive κ_a (by decrypting $Enc_{\kappa_c}(\kappa_a) \in Pub$ using κ_c). De Santis *et al.* [83] tweak this scheme to make it KI secure by introducing an intermediate secret value s_l for each label $l \in L$, from which the key (and intermediate secret) for all labels $l' \leqslant l$ can be derived using public information.

Several iterative schemes have been proposed in the literature [6, 47, 48, 79]. For example, Atallah *et al.* [6] propose two iterative schemes, the first of which is secure against key recovery (the *base* scheme), and the latter of which (the *extended scheme*) is also secure against key indistinguishability. Both schemes require users to only store one piece of secret material for their assigned label (κ_l for the base scheme and σ_l for the extended scheme), but require $\mathcal{O}(n^2)$ pieces of public derivation information since each arc in the graphical representation of the policy poset (which is, at least, the Hasse diagram) is associated with a piece of public information.

Castiglione *et al.* [26] propose an alternative iterative scheme based on a secret sharing scheme and symmetric encryption scheme in which users have to decrypt multiple pieces of public information in order to obtain 'shares' of a secret of a label l, which can then be used to decrypt a piece of public information associated with label l in order to obtain κ_l.

Edge-based KAS. Let $D = (V, A)$ be the graphical representation of the partially ordered set $(L, \leqslant)$ of an information flow policy, where $V = L$ and $A \subseteq \{xy : x, y \in L, y < x\}$. In an edge-based KAS, each arc $xy \in A$ is associated with a piece of public derivation material p_{xy} which enables a user with knowledge of σ_x to derive κ_y [35]. Both the direct and iterative scheme described above are examples of *edge-based* schemes, in which public information is associated with edges in the graphical representation of the policy poset.

Node-based KAS. In a node-based KAS, each *node* (or vertex) $l \in L$ is associated with a piece of public information n_l, which is used to assist in the derivation of κ_l [36]. The scheme proposed by Akl and Taylor [3] is an example of a node-based KAS in which keys are iteratively derived using node labels, and whose security is based on the hardness of factoring composite integers modulo n, where n is a product of two large primes.

2.5.4 Scheme Comparison

We may evaluate different schemes by considering a number of parameters. Let $|\sigma_l|$ be the size of σ_l where $l \in L$ (recall that in Section 2.5.2 we defined this to be the number of keys and intermediate secrets contained in σ_l). Then we write $\mathsf{k}_{\max}$ to denote the maximum size of σ_l taken over all $l \in L$ and K to denote $\sum_{l \in L} |\sigma_l|$. We write p to denote the number of items of public derivation information in Pub_{der} and d to denote the number of key derivation operations a user may be required to perform to derive a key. Let n denote the cardinality of L. Recall that, given a poset $(L, \leqslant)$, $A_{\min}$ is the set of arcs in the Hasse diagram of $(L, \leqslant)$ and $A_{\max}$ is the set of arcs in the transitive closure of the Hasse Diagram of $(L, \leqslant)$. Then the characteristics of the trivial, direct and iterative schemes described in Section 2.5.3 are summarised in Table 2.1 (note that here we compare the representatives of these types of schemes, as described in [36]).

Scheme	$\sigma_{\lambda(u)}$	Pub_{der}	K	$\mathsf{k}_{\max}$	p	d
Trivial	$\{\kappa(x) : x \leqslant \lambda(u)\}$	$-$	$n + \lvert A_{\max}\rvert$	$O(n)$	0	0
Direct	$\{\kappa(\lambda(u))\}$	$\{Enc_{\kappa_x}(\kappa_y) : y < x\}$	n	1	$\lvert A_{\min}\rvert$	$O(n)$
Iterative	$\{\kappa(\lambda(u))\}$	$\{Enc_{\kappa_x}(\kappa_y) : y \lessdot x\}$	n	1	$\lvert A_{\max}\rvert$	1

Table 2.1: How the parameters of various key assignment schemes vary.

Naturally, there is a trade-off between the amount of public information the trusted au-

thority needs to compute and make available, and the number of key derivation operations that are required by users to derive any of their respective keys. The direct scheme, for example, minimises the cost of key derivation at the expense of an increase in public information. Consider the example in Figure 2.7; the Hasse diagram of the poset has 10 arcs and the graph of the transitive closure has 23 arcs. Thus in the iterative scheme for this poset, $k = 1$, $p = 10$ and $d = 4$, compared to $k = 1$, $p = 23$ and $d = 1$ in the direct scheme [36]. Thus reducing d results in an increase in the amount of public information to support key derivation.

Even within the broad classes of KASs that we have identified, there is flexibility in designing a KAS. Informally, we may enforce a policy using a KAS in any way we see fit. We may, for example, increase the number of arcs (by including some transitive arcs), thereby decreasing the lengths of the directed paths in the graph and the number of key derivations that are required. For example, by adding the set of arcs $\{hd, gc, da\}$ to the Hasse diagram in Figure 2.5, as shown in Figure 2.9, a user u given $\kappa_{\lambda(u)}$ can derive any of their authorised keys in at most two derivation steps [6].

More complex schemes have been devised to reduce the number of derivation operations by increasing the number of arcs $|A|$ in the graphical representation of the poset [8, 31, 40]. Atallah *et al.* [6] propose n-hop schemes in which the key κ_y for any label $y \leqslant x$ can be derived in at most n derivation steps from κ_x, at the expense of an increase in the amount of public information. For instance, the 2-hop scheme for total orders requires $\mathcal{O}(n \log n)$ additional pieces of public information to support key derivation (one additional piece of public information for each new arc added to the Hasse diagram of the total order). Crampton extended these ideas to arbitrary interval-based access control policies [31].

Alternatively, we may increase the number of intermediate secrets (or keys) given to each user $u \in U$ (i.e. the size of the user secret $\sigma_{\lambda(u)}$) and reduce the derivation time (keeping the number of arcs constant). This corresponds to allowing the user to start from multiple points in the graph. For example, considering Figure 2.5, we may give users assigned to label h the keys κ_h and κ_d in order to reduce the number of derivation steps required to derive any authorised key from four (if we had just given κ_h) to two. In practice, there may be constraints that will dictate what kind of cryptographic enforcement schemes will be appropriate. There may be constraints, for example, on the computational power and/or storage of the end-user devices (e.g. smart cards); or it may not be possible to provide an

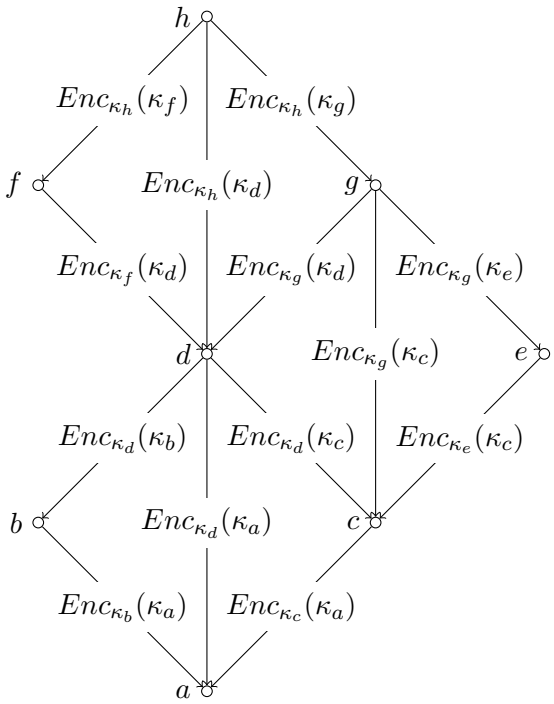


Figure 2.9: Adding arcs to Figure 2.5 to reduce the number of key derivation steps.

on-line server to store public information (e.g. in military ad-hoc networks).

Many KASs in the literature aim to provide each user with a user secret of fixed size (i.e. $\mathcal{O}(1)$), which is simply the intermediate secret or key for their assigned label, i.e. $\sigma_{\lambda(u)} = s_{\lambda(u)}$ or $\kappa_{\lambda(u)}$ [6, 36, 83], from which they can derive all their respective keys (see iterative and direct schemes in Section 2.5.4, for example). The trade-off, however, is that the amount of public information required to support key derivation and/or the time it takes to derive a key may be substantial. For example, public information may be significantly large in situations where security labels are defined in terms of (subsets of) attributes, as may be the case for attribute-based schemes [18, 58, 74].

Chapter 3

Chain-based Key Assignment Schemes

Contents

In this chapter, we describe a type of key assignment scheme for information flow policies that does not require public derivation information and is based on chain partitions. We describe how to best partition the policy's underlying poset into chains in order to reduce the total number of distributed intermediate secrets and maximum number of intermediate secrets required by each user. This chapter is based on the following published works:

- *J. Crampton, N. Farley, M. Jones and G. Gutin, Optimal Constructions for Chain-Based Cryptographic Enforcement of Information Flow Policies, DBSec 2015.*
- *J. Crampton, N. Farley, M. Jones, G. Gutin and B. Poettering, Cryptographic En-*

forcement of Information Flow Policies without Public Information via Tree Partitions, Journal of Computer Security 25(6): 511-535 (2017).

3.1 Introduction

In this chapter, we consider the cryptographic enforcement of read-only information flow policies using KASs based on chain partitions. Key Assignment Schemes instantiated upon a poset of chains (or total orders) may not require public derivation information to support key derivation. By definition, given a chain C, there is a *unique* directed path from x to y (in the Hasse diagram of C) whenever $y < x$. Thus, informally, one could derive keys for labels lower in the chain by iteratively applying a PRF directly to intermediate secrets for labels higher in the chain, without requiring additional public information. (This is not possible when the Hasse diagram of the poset being enforced contains security label vertices with more than one incoming arc; hence public information is typically required in such schemes to enable users assigned from different, incomparable security labels to derive the key for their shared descendant security labels.). Thus, given that a primary motivation for this thesis is to reduce the amount of public derivation information required within KASs, a natural starting point is to consider KASs based on chain partitions.

This observation has led to the development of chain-based KASs [32, 47, 48] for arbitrary information flow policies. Recalling from Section 2.1[1], $\overline{C} = \{C_1, \ldots, C_t\}$ is a *chain partition* of a poset $(L, \leqslant)$ if, for each $i, j \in \{1, \ldots, t\}$ where $i \neq j$, $C_i \subseteq L$ is a chain, $C_i \cap C_j = \emptyset$ and $C_1 \cup \cdots \cup C_t = L$. The basic idea is to partition the information flow policy poset $(L, \leqslant)$ into disjoint chains[2] and instantiate a separate KAS for each chain.

Several KASs have been proposed for enforcing policies represented as chain partitions. For example, Crampton *et al.* [32] proposed two chain-based key assignment schemes; one based on the use of hash functions and the other on the RSA scheme. Unfortunately, such schemes are, at best, only secure against key recovery, and the RSA-based scheme is relatively expensive in terms of storage and computation since it requires users to store relatively large RSA keys, and derivation is based on exponentiation computations.

[1] A more formal definition of a chain partition is provided in Section 2.1.

[2] Dilworth [42] showed that any poset can be partitioned into $l \geqslant w$ chains where w is the width of the poset.

Furthermore, as suggested by Freire *et al.* [47], the RSA scheme uses multiple RSA moduli; one modulus for each chain in the partition, and thus can be expensive in terms of public information since each such moduli (which may be large) has to be publicly available.

Freire *et al.* [47] propose a KI-secure scheme for total orders, which can be used to instantiate a KAS on each chain in a given chain partition. The security of the scheme is based on the hardness of factoring Blum integers (namely, finding the prime roots p, q of a Blum integer N where $p = q = 3 \bmod 4$.). Similarly to the scheme by Crampton *et al.* [32], key derivation is expensive in such schemes since it requires expensive exponentiation operations. In order to reduce the derivation costs, Freire *et al.* [48] later introduced another KAS for total orders which can similarly be used to enforce a policy represented as a chain partition. In this scheme, keys are derived from intermediate secrets using pseudorandom functions which are particularly cheap to compute. The scheme achieves the strongest security notion for KASs, namely *strong key indistinguishability* (see Section 2.5.2 for definition).

Contributions. Whilst the schemes proposed by Crampton *et al.* [36] and Freire *et al.* [47, 48] remove the need for public derivation information, each user may require large user secrets [32, 47, 48] (informally, one intermediate secret/key per chain in the partition). Furthermore, existing work on chain-based KASs assume that either the policy to be enforced is a total order, or that a chain partition of the underlying poset has *already* been found and simply generates the required secrets and keys for this partition [32, 47, 48]. This approach ignores the fact that there will be multiple ways to partition the partially ordered set that defines a policy into chains, each of which will have a different impact on the characteristics of the resulting KAS. Figure 3.2 shows three different chain partitions of the Hasse diagram of a poset in Figure 3.1. As we will show (in Section 3.3), the chain partition chosen affects the number of intermediate secrets a user may require, the maximum number of key derivations, and so on. Recall that the existing trade-offs in KASs means that by eliminating public derivation information, the size of user secrets may increase as a result. Thus, it is important, if we are to make best use of existing chain-based schemes, that we know how to *efficiently* find an 'optimal' chain partition (in terms of KAS characteristics) to use for a given information flow policy. More precisely, we show how to construct a chain partition for any given poset such that the total number of intermediate secrets required by any user, and by the entire user population, is minimised.

Minimising the total number of intermediate secrets required by any user and in total is desirable in situations in which user devices have limited storage, and in scenarios in which it is expensive to transmit intermediate secrets to users (e.g. in military ad-hoc networks).

Our first contribution (Theorem 2) is to show how $\widehat{\mathsf{K}}(\overline{C})$, the (total) number of intermediate secrets distributed to users for a chain partition $\overline{C}$, is related to the set of arcs in the representation of $\overline{C}$ as a directed acyclic graph. We then prove that $\widehat{\mathsf{K}}(\overline{C})$ is determined by the end-points of the chains in $\overline{C}$ (Lemma 2). This, in turn, allows us to prove there exists a chain partition that simultaneously minimises the number of intermediate secrets required and the number of chains in the partition (Theorem 3). The result is also of practical importance, since the number of chains in $\overline{C}$ provides a tight upper bound on the number of intermediate secrets required by any one user; namely a user requires at most one intermediate secret per chain in the partition. Our main contribution (Theorem 1 and Section 3.5) is to develop a polynomial-time algorithm (in the number of security labels n) to find a chain partition $\overline{C}$ such that $\widehat{\mathsf{K}}(\overline{C})$ and the number of chains is minimised (with respect to all chain partitions). Our algorithm is based on finding an optimal feasible flow in a network and makes use of the characterisation of the number of intermediate secrets in terms of the set of arcs (established in Theorem 2) to define the capacities of the arcs in the network. We thereby provide rigorous foundations for the development of efficient chain-based key assignment schemes.

3.2 Chain-based Enforcement

In this chapter, we are interested in the use of chain-based KASs to enforce information flow policies since such schemes eliminate the need for public derivation information. As mentioned previously, if $(L, \leqslant)$ is a chain, then (by definition) there is a unique directed path from x to y (in the Hasse diagram of $(L, \leqslant)$) whenever $y < x$. Then, informally, we may derive intermediate secrets and keys iteratively down each chain. For example, one could assign a random intermediate secret s_t to the top label t in the chain and then, for all $y \lessdot x$ in the chain, define:

$$s_y = \mathcal{F}_{s_x}(0)$$
$$\kappa_y = \mathcal{F}_{s_y}(1),$$

where $\mathcal{F}$ is a pseudorandom function. Thus, if $y < x$, there exist $z_1, \ldots, z_l \in L$ with $y = z_1 < z_2 < \cdots < z_l = x$; κ_y may be derived from s_x by iteratively deriving $s_{z_i} = \mathcal{F}_{s_{z_{i+1}}}(0)$ for $i = l-1, \ldots, 1$ and then deriving $\kappa_y = \mathcal{F}_{s_y}(1) = \mathcal{F}_{s_{z_1}}(1)$.

Of course, when the policy poset to be enforced is a chain, defining user secrets is easy; we simply define $\sigma_x = s_x$ for all $x \in L$. In most cases, however, the poset is not a total order, and thus in these situations a chain partition of the poset must first be found. Unfortunately, forming the partition breaks some of the 'connectivity' of the partial ordering. We may repair these breaks by issuing more than one intermediate secret to some users. Thus, given a chain partition of a poset, we need to determine which intermediate secrets each user should be given (and thus the intermediate secrets that each user secret σ_x for $x \in L$ should comprise).

Let $(L, \leqslant)$ be a poset and $C = x_1 > x_2 > \cdots > x_m$ be a chain in L. Then we say any chain of the form $x_i > x_{i+1} > \cdots > x_m$, $1 \leqslant i \leqslant m$, is a *suffix* of C; the empty chain is (vacuously) also a suffix of C.

Proposition 1. *For all $x \in L$ and any chain $C \subseteq L$, $\downarrow x \cap C$ is a suffix of C.*

The above result (due to Crampton *et al.* [32, Proposition 4]) enables us to define, for a given chain partition $\overline{C}$, the intermediate secrets that should form part of σ_x for each $x \in L$ and should thus be given to a user $u \in U(x)$, since $\downarrow x$ defines the labels for which u is authorised. Given a chain partition $\overline{C} = \{C_1, \ldots, C_\ell\}$ of $(L, \leqslant)$, $\{\downarrow x \cap C_1, \ldots, \downarrow x \cap C_\ell\}$ is a disjoint collection of chain suffixes. Hence, a user in $U(x)$ must be given the intermediate secrets for the maximal elements in the non-empty suffixes $\downarrow x \cap C_1, \ldots, \downarrow x \cap C_\ell$. Thus, any user requires at most ℓ intermediate secrets. Let $\phi(x, \overline{C}) \subseteq L$ denote this set of maximal elements. (Clearly, $x \in \phi(x, \overline{C})$ for all chain partitions $\overline{C}$ and all $x \in L$.) For example, consider the (Hasse diagram of the) chain partition $\overline{C}_1$ in Figure 3.2 of the poset $(L, \leqslant)$ shown in Figure 3.1. Table 3.1 shows $\phi(x, \overline{C}_1)$ for each security label x.

We now describe the SKI-secure KAS (see Section 2.5.2 and Freire *et al.* [48] for further details) for chain-based schemes proposed by Freire *et al.* [48] based on pseudorandom functions, which we will use to enforce our information flow policies once we have found a chain partition of its underlying poset. Freire *et al.* [48] provide a formal description of the Setup and Derive algorithms.

x	$\phi(x, \overline{C}_1)$
a	$\{a\}$
b	$\{b\}$
c	$\{a, c\}$
d	$\{b, c, d\}$
e	$\{a, e\}$
f	$\{b, c, d, f\}$
g	$\{b, e, g\}$
h	$\{b, e, g, h\}$

Table 3.1: $\phi(x, \overline{C}_1)$ for each $x \in L$ where $\overline{C}_1$ is a chain partition of Figure 3.1 shown in Figure 3.2a.

Given a policy poset $(L, \leqslant)$, let σ_x, s_x and κ_x be the user secret, intermediate secret and key associated to a label $x \in L$ respectively. Let $\overline{C}$ be a chain partition of $(L, \leqslant)$ and let $p(x)$ be the *parent* of label x in $\overline{C}$ (if such an element exists). Since $\overline{C}$ is a chain partition, $p(x)$ is unique. Let $\mathcal{F} : \{0,1\}^\rho \times D \to \{0,1\}^\rho$ be a PRF where D is the message space of the PRF and let $c_0, c_1 \in D$ where $c_0 \neq c_1$.

Informally, the Setup algorithm takes as input a chain partition $\overline{C}$ of $(L, \leqslant)$ and security parameter 1^ρ and performs the following steps:

1. for each chain C_i in $\overline{C}$, assign a random value in $\{0,1\}^\rho$ to the intermediate secret for the top element in C_i and generate an intermediate secret for each remaining element y in the chain by applying a PRF $\mathcal{F}$ keyed with the secret of its parent in C_i to c_0, i.e. $s_y = \mathcal{F}_{s_{p(y)}}(c_0)$;
2. for each element $x \in L$, generate κ_x by applying the PRF $\mathcal{F}$ keyed under s_x to c_1, that is, $\kappa_x = \mathcal{F}_{s_x}(c_1)$;
3. define the user secret, for each $x \in L$, to be $\sigma_x \stackrel{\text{def}}{=} \{(z, s_z) : z \in \phi(x, \overline{C})\}$;
4. set $Pub \leftarrow (\overline{C}, \{c_0, c_1\})$;[3]
5. return $(\{\sigma_x, \kappa_x\}_{x \in L}, Pub)$.

The Derive algorithm performs the following steps, given the chain partition $\overline{C}$ (contained in Pub), $x, y \in L$ (where it is assumed that $y \leqslant x$) and σ_x:

[3]Freire *et al.* [48] left this implicit and suggested that Pub could be replicated in all user secrets if required.

1. if $x = y$, then output $\mathcal{F}_{s_x}(c_1)$ (where $(x, s_x) \in \sigma_x$);
2. if $y < x$, then find $(z, s_z) \in \sigma_x$ such that $z \geqslant y$, so there exists $z = z_0 \succcurlyeq \cdots \succcurlyeq z_t = y$ in $\overline{C}$, and compute $\mathcal{F}_{s_{z_0}}(c_0) = s_{z_1}, \ldots, \mathcal{F}_{s_{z_{t-1}}}(c_0) = s_y$; output $\kappa_y = \mathcal{F}_{s_y}(c_1)$.

Remark 1. *Let w be the* width *(see Section 2.1) of a poset $(L, \leqslant)$. Clearly, $(L, \leqslant)$ cannot have a chain partition containing less than w chains (since no two elements in an antichain of size w can belong to the same chain, by definition). Dilworth's theorem asserts that there exists a chain partition of $(L, \leqslant)$ into w chains [42]. Thus, if we can find a chain partition of $(L, \leqslant)$ into w chains, no user will require more than w intermediate secrets. (If a user u were to have more intermediate secrets than there are chains in the partition, then there must exist a chain containing y and z for which u has intermediate secrets and one of the intermediate secrets may be derived from the other.)*

In the described chain-based KAS, a user in $U(x)$ will need to be given $\left|\phi(x, \overline{C})\right|$ intermediate secrets ($|\sigma_x| = \left|\phi(x, \overline{C})\right|$), in contrast to most KASs in the literature in which each user receives a single intermediate secret or key [6, 36]. However, chain-based KASs have substantial benefits: (i) they require no public derivation information [32][4]; (ii) they can use cryptographic primitives that are very easy to compute; and (iii) it is easy to construct such schemes with the strong key indistinguishability property [48].

3.3 Problem Statement

Certain aspects of chain-based KASs are not well understood. For example, it is not obvious how to best select a chain partition to reduce the number of intermediate secrets per user. As we have already noted, some users will require multiple intermediate secrets, each of which corresponds to a unique label in L. In particular, a user u in $U(x)$ will require an intermediate secret for each chain that contains an element y such that $y < x$. Three chain partitions of the poset in Figure 3.1 are shown in Figure 3.2. We have, for example, $\phi(g, \overline{C}_1) = \{b, e, g\}$, $\phi(g, \overline{C}_2) = \{b, d, g\}$, and $\phi(g, \overline{C}_3) = \{d, g\}$. Hence, the number of intermediate secrets required (by each user and in total across the entire user population) will vary, depending on the chain partition chosen. Thus, considering various chain partitions of L, we may ask:

[4] Atallah *et al.* [6] suggest the minimal amount of public information required to support key derivation is the poset itself, however we only require the chain partition of the policy poset (which is a subgraph of the policy poset) to be publicly available.

- How do we minimise $\mathsf{k}_{\max}$, the maximum number of intermediate secrets a user may require?
- How do we minimise K, the total number of intermediate secrets contained in all the user secrets?
- How do we minimise $\widehat{\mathsf{K}}$, the total number of intermediate secrets that need to be issued to users?

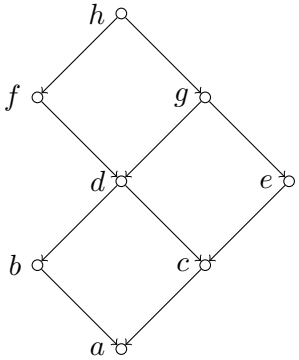
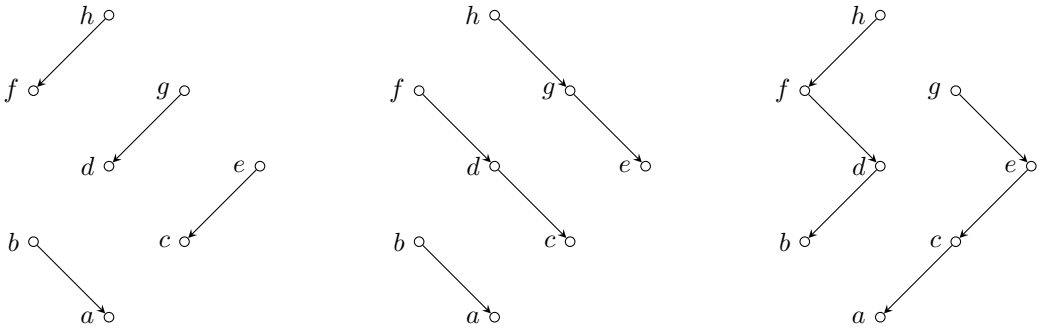


Figure 3.1: The Hasse diagram of a simple poset.



(a) $\overline{C}_1$ (b) $\overline{C}_2$ (c) $\overline{C}_3$

Figure 3.2: Three chain partitions of the poset in Figure 3.1.

More formally, given a chain partition $\overline{C}$ of $(L, \leqslant)$, we may regard ϕ as a function from L to 2^L that is completely determined by $\overline{C}$. Thus, given a chain partition $\overline{C}$, we can define the following values:

$$\begin{aligned}
\mathsf{k}_{\max}(\overline{C}) &\stackrel{\text{def}}{=} \max\left\{\left|\phi(x, \overline{C})\right| : x \in L\right\} \\
\mathsf{K}(\overline{C}) &\stackrel{\text{def}}{=} \sum_{x \in L} \left|\phi(x, \overline{C})\right| \\
\widehat{\mathsf{K}}(\overline{C}) &\stackrel{\text{def}}{=} \sum_{x \in L} |U(x)| \cdot \left|\phi(x, \overline{C})\right|.
\end{aligned}$$

Values of $\mathsf{k}_{\max}(\overline{C}_i)$ and $\mathsf{K}(\overline{C}_i)$ for each chain partition $\overline{C}_i$ in Figure 3.2 are shown in

Table 3.2; security label h is used for illustrative purposes.[5]

Partition	$\phi(h, \overline{C}_i)$	$\mathsf{k}_{\max}(\overline{C}_i)$	$\mathsf{K}(\overline{C}_i)$
$\overline{C}_1$	$\{b, e, g, h\}$	4	20
$\overline{C}_2$	$\{b, f, h\}$	3	17
$\overline{C}_3$	$\{g, h\}$	2	13

Table 3.2: $\phi(h, \overline{C}_i)$, $\mathsf{k}_{\max}(\overline{C}_i)$ and $\mathsf{K}(\overline{C}_i)$ for the chain partitions in Figure 3.2.

The important question is: can we minimise these parameters (over all choices of chain partition $\overline{C}$ for a given information flow policy)? In short, given an information flow policy $((L, \leqslant), U, O, \lambda)$, how do we determine the best choice of $\overline{C}$ of $(L, \leqslant)$ for use in a chain-based KAS such that $\mathsf{k}_{\max}(\overline{C})$ and $\widehat{\mathsf{K}}(\overline{C})$ are minimised?[6] It is this question we address in the remainder of the chapter. In particular, at the end of Section 3.5, we prove the following result.

Theorem 1. *Let $P = ((L, \leqslant), U, O, \lambda)$ be an information flow policy where $(L, \leqslant)$ is of width w and let $\widehat{\mathsf{K}}_{min}$ denote the minimum number of intermediate secrets required by a chain-based key assignment scheme for P. Then in $O(|L|^4\, w)$ time, we can find a chain partition $\overline{C}$ for which the corresponding chain-based key assignment scheme only requires $\widehat{\mathsf{K}}_{min}$ intermediate secrets and $\mathsf{k}_{max}(\overline{C}) \leqslant w$.*

Remark 2. *We assume throughout that our information flow policy poset has a maximum element. We may assume this without loss of generality: given an information flow policy poset $(L, \leqslant)$ without a maximum element, we simply add a maximum element r and define $r \geqslant m$ for all maximal elements m in L; no users are assigned to r. Observe that such a transformation does not affect the values of k_{max} and $\widehat{\mathsf{K}}$.*

3.4 Computing $\mathsf{k}_{\max}(\overline{C})$ and $\widehat{\mathsf{K}}(\overline{C})$

Informally, we take a policy poset $(L, \leqslant)$ and construct a second poset $(L, \leqslant')$, where $x <' y$ implies $x < y$ (but $x < y$ does not necessarily imply $x <' y$). We will say $\leqslant'$ is *contained* in $\leqslant$. In particular, any chain partition $\overline{C}$ of $(L, \leqslant)$ defines a second poset

[5] Note that we can deduce K from $\widehat{\mathsf{K}}$ by letting $|U(x)| = 1$ for all $x \in L$.

[6] Crampton *et al.* [32] observed that further research was needed to identify the best choice of chain partition for a given information flow policy. While subsequent research [47] has formalised and strengthened the security properties of chain-based KASs [48], we are not aware of any research that specifies how to select a chain partition.

$(L, \leqslant_{\overline{C}})$, where $x <_{\overline{C}} y$ if and only if x and y belong to the same chain and $x < y$; thus $\leqslant_{\overline{C}}$ is contained in $\leqslant$ for any $\overline{C}$. Note, however, that $x \lessdot_{\overline{C}} y$ does not necessarily imply $x \lessdot y$.[7] Intuitively, we have to 'break' relations in $(L, \leqslant)$ in order to form $\overline{C}$. In order to 'repair' such breaks, we may have to give users additional intermediate secrets, which allows them to start at various points in the Hasse Diagram of $\overline{C}$, and from these points, derive all their respective keys by iteratively deriving down chains.

Definition 7. *Given a poset $(L, \leqslant)$ and $z < y$, we define*

$$\gamma(yz) = \{x \in L : x \geqslant z, x \not\geqslant y\}.$$

Thus $z \in \gamma(yz)$ and $y \notin \gamma(yz)$. For the maximum element $r \in L$ and any $y, z \in L$ such that $z < y$, $r \notin \gamma(yz)$. Informally, the intuition behind γ is that its cardinality measures the 'damage' that would be done by creating a chain partition $\overline{C}$ such that $z \lessdot_{\overline{C}} y$, because having $z \lessdot_{\overline{C}} y$ means that $z \not\leqslant_{\overline{C}} x$ for any $x \in \gamma(yz)$. Thus, every user in $U(x)$ will require an extra intermediate secret, s_z, in order to derive κ_z. We will capture this intuition more precisely in Lemma 1.

Remark 3. *For maximum element r and any chain partition $\overline{C} = \{C_1, \ldots, C_\ell\}$, $\phi(r, \overline{C}) = \{t_1, \ldots, t_\ell\}$, where t_i is the maximum element in chain C_i. Moreover, $r = t_i$ for some i.*

Hence, we can construct a tree $T_{\overline{C}} = (L, \leqslant_{T_{\overline{C}}})$, where $y \lessdot_{T_{\overline{C}}} x$ if and only if one of the following conditions holds: (i) $y = t_j$ where $t_j \neq r$, and $x = r$; (ii) $y \lessdot_{\overline{C}} x$.

We shall define $T_{\overline{C}}$ to be the *tree representation* of $\overline{C}$.

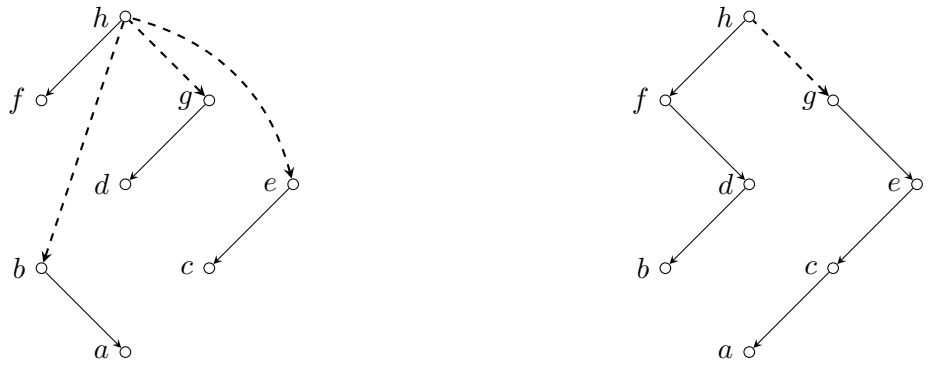
Figure 3.3 illustrates the construction of two such trees, using chain partitions from Figure 3.2; the arcs used to create the trees are shown as dashed lines.

Will will use the tree representation of $\overline{C}$ to prove the following result:

Lemma 1. *Let $\mathcal{P} = (L, \leqslant)$ be a poset and let $\overline{C}$ be a chain partition of $(L, \leqslant)$. Then, for all $x, y, z \in L$ such that $x \neq r$ and $z \lessdot_{T_{\overline{C}}} y$,*

$$z \in \phi(x, \overline{C}) \text{ if and only if } x \in \gamma(yz).$$

[7] To see this, consider the poset of four elements, in which $a \lessdot b \lessdot d$ and $a \lessdot c \lessdot d$ with $b \not\leqslant c, c \not\leqslant b$. Then $\{\{b\}, \{c\}, \{a, d\}\}$ is a chain partition and $a \lessdot_{\overline{C}} d$, but it does not hold that $a \lessdot d$.



(a) $T_{\overline{C}_1}$ (b) $T_{\overline{C}_3}$

Figure 3.3: Creating trees from partitions $\overline{C}_1$ and $\overline{C}_3$ in Figure 3.2.

Proof. Given $z \in \phi(x, \overline{C})$ and chain partition $\overline{C} = \{C_1, \ldots, C_\ell\}$, an element $y_i \in \phi(x, \overline{C})$ if and only if y_i is the maximum element in $C_i \cap {\downarrow_{\mathcal{P}}} x$ (where $C_i \cap {\downarrow_{\mathcal{P}}} x$ is non-empty) for some $C_i \in \overline{C}$ (Section 3.3). Thus, $z \in {\downarrow_{\mathcal{P}}} x$ and hence $z \leqslant x$. Moreover, $x \not\geqslant y$ (otherwise there would exist $t \in \phi(x, C)$ such that $y \leqslant_{T_{\overline{C}}} t$ and hence $z \lessdot_{T_{\overline{C}}} y \leqslant_{T_{\overline{C}}} t$, violating the condition that z is the maximum element in the suffix $C_i \cap {\downarrow} x$). Thus $x \in \gamma(yz)$.

Now suppose $x \in \gamma(yz)$. Then $x \not\geqslant y$, by definition, and hence y does not belong to ${\downarrow} x \cap C_i$ for any i. However, $x \geqslant z$; hence, there exists $t \in \phi(x, \overline{C})$ such that $z \leqslant_{T_{\overline{C}}} t$. Since $\overline{C}$ is a chain partition, the only parent of z in $T_{\overline{C}}$ is y. Hence it must be the case that $z = t$ (and thus $z \in \phi(x, \overline{C})$). □

Essentially, the above result means that given a chain partition $\overline{C} = (L, \leqslant_{\overline{C}})$ and its tree representation $T_{\overline{C}}$, a security label $x \in L$ must be associated with the intermediate secret s_z if the arc yz belongs to $T_{\overline{C}}$. Thus, every user $u \in U(x)$ must be given the additional intermediate secret s_z.

Let $(L, \leqslant)$ be an information flow policy poset and let $y, z \in L$ with $z < y$. Then, following Crampton *et al.* [34], we define

$$\omega(yz) \stackrel{\text{def}}{=} \sum_{x \in \gamma(yz)} |U(x)| \,.$$

We will be interested in minimising $\sum \omega(yz)$, where the sum is taken over all pairs (y, z) such that $z \lessdot_{T_{\overline{C}}} y$. The intuition behind this definition is that it captures, in some appropriate sense, the connectivity that is lost from $(L, \leqslant)$ by using $(L, \leqslant_{\overline{C}})$. Since every element in $(L, \leqslant_{\overline{C}})$ has at most one parent, $\gamma(yz)$ represents those elements in L that

become 'disconnected' from z by defining $z \lessdot_{\overline{C}} y$, and thus need to be assigned the intermediate secret s_z. $\omega(yz)$ captures the number of users who will require s_z if $z \lessdot_{\overline{C}} y$. The next result establishes an exact correspondence between $\phi(x, \overline{C})$ and $\gamma(yz)$, and enables us to use network flow techniques to compute a chain partition $\overline{C}$ that minimises $\widehat{\mathsf{K}}(\overline{C})$ (as we explain in Section 3.5).

Theorem 2. *Let $((L, \leqslant), U, O, \lambda)$ be an information flow policy and let $\overline{C}$ be a chain partition of $(L, \leqslant)$ with maximum element r. Then*

$$\widehat{\mathsf{K}}(\overline{C}) = \ell\,|U(r)| + \sum_{z \lessdot_{T_{\overline{C}}} y} \omega(yz),$$

where ℓ is the number of chains in $\overline{C}$.

Proof. By definition,

$$\widehat{\mathsf{K}}(\overline{C}) = \sum_{x \in L} |U(x)|\,\left|\phi(x, \overline{C})\right| = |U(r)|\,\left|\phi(r, \overline{C})\right| + \sum_{x \in L \setminus r} \sum_{z \in L} |U(x)|\,\delta(x, z),$$

where $\delta(x, z)$ equals 1 if $z \in \phi(x, \overline{C})$ and 0 otherwise. By Lemma 1, we have $\delta(x, z) = 1$ if and only if $x \in \gamma(yz)$ for $z \lessdot_{T_{\overline{C}}} y$. Moreover, y is unique, since $T_{\overline{C}}$ is a tree. Therefore

$$\sum_{x \in L \setminus r} \sum_{z \in L} |U(x)|\,\delta(x, z) = \sum_{z \lessdot_{T_{\overline{C}}} y} \sum_{x \in \gamma(yz)} |U(x)| = \sum_{z \lessdot_{T_{\overline{C}}} y} \omega(yz).$$

As $r \geqslant x$ for all $x \in L$, $\phi(r, \overline{C})$ must contain exactly one element from each chain in $\overline{C}$. Therefore $|U(r)| \cdot \left|\phi(r, \overline{C})\right| = \ell\,|U(r)|$, as required. □

Whilst the use of γ and ω help to provide a good intuition for the relationship between the choice of arcs for a chain partition $\overline{C}$ and $\widehat{\mathsf{K}}(\overline{C})$ ($\mathsf{K}(\overline{C})$), the following result shows that the number of intermediate secrets required by a chain partition can be computed by considering only the minimum elements in the chain partition.

Lemma 2. *Let $\overline{C} = \{C_1, \ldots, C_\ell\}$ be a chain partition of $\mathcal{P} = (L, \leqslant)$ and let chain C_i have bottom element b_i, $1 \leqslant i \leqslant \ell$. Let $\uparrow x = \uparrow_{\mathcal{P}} x$ and and $\downarrow x = \downarrow_{\mathcal{P}} x$ for all $x \in L$* [8]*. Then*

$$\mathsf{K}(\overline{C}) = \sum_{i=1}^{\ell} |\uparrow b_i| \qquad \textit{and} \qquad \widehat{\mathsf{K}}(\overline{C}) = \sum_{i=1}^{\ell} \sum_{x \in \uparrow b_i} |U(x)|\,.$$

[8] Unless otherwise stated, we will use to $\uparrow x$ to denote $\uparrow_{\mathcal{P}} x$ and and $\downarrow x = \downarrow_{\mathcal{P}} x$ for all $x \in L$ where $\mathcal{P} = (L, \leqslant)$.

Proof. We have, by definition,

$$\begin{aligned}
\widehat{\mathsf{K}}(\overline{C}) &= \sum_{x\in L} |U(x)|\,\big|\phi(x,\overline{C})\big| = \sum_{x\in L} |U(x)|\,|\{C_i : C_i \cap {\downarrow}x \neq \emptyset, 1 \leqslant i \leqslant \ell\}| \\
&= \sum_{x\in L} |U(x)|\,|\{b_i : x \geqslant b_i, 1 \leqslant i \leqslant \ell\}| \\
&= \sum_{x\in L}\sum_{i=1}^{\ell} |U(x)|\,\delta(x,b_i) \qquad \text{where } \delta(x,b_i)=1 \text{ if } x \geqslant b_i \text{ and 0 otherwise} \\
&= \sum_{i=1}^{\ell}\sum_{x\in L} |U(x)|\,\delta(x,b_i) = \sum_{i=1}^{\ell}\sum_{x\in{\uparrow}b_i} |U(x)| \ .
\end{aligned}$$

Clearly, we may prove the result for $\mathsf{K}(\overline{C})$ in an analogous fashion by setting $|U(x)| = 1$ for each $x \in L$. □

In the chain partition $\overline{C}_1$ in Figure 3.2a, for example, the bottom elements are a, c, d and f and $|{\uparrow}a| = 8$, $|{\uparrow}c| = 6$, $|{\uparrow}d| = 4$ and $|{\uparrow}f| = 2$. Thus, $\mathsf{K}(\overline{C}_1) = 20$ ($\widehat{\mathsf{K}}(\overline{C}_1) = 20$ when $|U(x)| = 1$ for all $x \in L$).

Theorem 3. *Let $P = (L, \leqslant), U, O, \lambda)$ be an information flow policy where $(L, \leqslant)$ has width w and let $\widehat{\mathsf{K}}_{min}$ denote the minimum number of intermediate secrets required by the user population for any chain-based key assignment scheme for P. Then there exists a chain partition containing w chains such that $\widehat{\mathsf{K}}(\overline{C}) = \widehat{\mathsf{K}}_{min}$.*

Proof. Let $\overline{C}$ be a chain partition of $(L, \leqslant)$ into $t \geqslant w$ chains such that $\widehat{\mathsf{K}}(\overline{C}) = \widehat{\mathsf{K}}_{min}$ and let B be the set of bottom vertices in the chains of $\overline{C}$. A result of Gallai and Milgram asserts that if a chain partition $\overline{C}$ of a poset $(L, \leqslant)$ contains t chains, where $t > w$, then there exists a chain partition $\overline{C}'$ into $t-1$ chains such that the set of bottom vertices in $\overline{C}'$ is a subset of B [51].[9] Hence, by iterated applications of the Gallai-Milgram result, there exists a chain partition $\overline{C}^*$ of width w such that the set of bottom vertices B^* in $\overline{C}^*$ is a subset of B. Moreover, by Lemma 2,

$$\widehat{\mathsf{K}}(\overline{C}^*) = \sum_{b\in B^*}\sum_{x\in{\uparrow}b} |U(x)| \leqslant \sum_{b\in B}\sum_{x\in{\uparrow}b} |U(x)| \ .$$

By the minimality of $\widehat{\mathsf{K}}_{min}$, we deduce that $\widehat{\mathsf{K}}(\overline{C}^*) = \widehat{\mathsf{K}}_{min}$. □

[9] The result is phrased in the language of digraphs, but every poset may be represented by an equivalent transitive acyclic digraph.

Corollary 1. *Let $(L, \leqslant)$ be the poset of an information flow policy P. There exists a chain partition $\overline{C}$ such that the total number of secrets $\widehat{\mathsf{K}}(\overline{C})$ is minimised and $\mathsf{k}_{max}(\overline{C}) \leqslant w$.*

Proof. The result follows immediately from Theorem 3, the definition of $\mathsf{k}_{\max}(\overline{C}) = \max\left\{\left|\phi(x, \overline{C})\right| : x \in L\right\}$, and the fact that $\left|\phi(x, \overline{C})\right|$ is bounded above by the number of chains in $\overline{C}$ for all $x \in L$. □

3.5 Finding a Chain Partition Requiring $\widehat{\mathsf{K}}_{min}$ Intermediate Secrets

Suppose $(L, \leqslant)$ is a poset of width w. In general, a chain partition of $(L, \leqslant)$ has $\ell \geqslant w$ chains. Theorem 3 asserts that there exists a partition of $(L, \leqslant)$ into w chains such that the corresponding KAS requires the minimum number of intermediate secrets. We now show how such a chain partition may be constructed. In particular, we show how to transform the problem of finding a chain partition $\overline{C}$ such that $\widehat{\mathsf{K}}(\overline{C})$ attains the minimum value into a problem of finding a minimum cost flow in a network.

Informally, a *network* is a directed graph in which each arc is associated with a *capacity*. A *network flow* associates each arc in a given network with a flow, which must not exceed the capacity of the arc. Networks are widely used to model systems in which some quantity passes through channels (arcs in the network) that meet at junctions (vertices); examples include traffic in a road system, fluids in pipes, or electrical current in circuits. In our setting, we model an information flow policy as a network in which the capacities are determined by the weights ω. Our definitions for networks and network flows follow the presentation of Bang-Jensen and Gutin [12].

Definition 8. *A* network *is a tuple $\mathcal{N} = (D, l, u, c, \beta)$, where:*

- *$D = (V, A)$ is a directed graph with vertex set V and arc set A;*
- *$l : V \times V \to \mathbb{N}$ such that $l(vv') = 0$ if $vv' \notin A$ and $l(vv') \geqslant 0$ otherwise;*
- *$u : V \times V \to \mathbb{N}$ such that $u(vv') = 0$ if $vv' \notin A$ and $u(vv') \geqslant l(vv') \geqslant 0$ otherwise;*
- *$c : V \times V \to \mathbb{R}$;*

- $\beta : V \to \mathbb{R}$ *such that* $\sum_{v \in V} \beta(v) = 0$.

Intuitively, l and u represent lower and upper bounds, respectively, on how much flow can pass through each arc, and c represents the cost associated with each unit of flow in each arc. The function β ('balance vector') represents how much flow should enter or leave the network at a given vertex. If $\beta(x) = 0$, then the flow going into vertex x should be equal to the flow going out of vetex x. If $\beta(x) > 0$, then there should be $\beta(x)$ more flow coming out of x than going into x. If $\beta(x) < 0$, there should be $\beta(x)$ more flow going into x than coming out of x.

Given a network $\mathcal{N} = ((V, A), l, u, c, \beta)$, we define a flow[10] f to be a function $f : V \times V \to \mathbb{N}$. Given a flow f in $\mathcal{N}$, define the function β_f to be [12]:

$$\beta_f(v) = \sum_{v' \in V} (f(vv') - f(v'v)),$$

for all $v \in V$.

Definition 9. *Given a network* $\mathcal{N} = ((V, A), l, u, c, \beta)$, $f : V \times V \to \mathcal{N}$ *is a* feasible flow *for* $\mathcal{N}$ *if the following conditions are satisfied:*

- $u(vv') \geqslant f(vv') \geqslant l(vv')$ *for every* $vv' \in V \times V$*;*
- $\beta_f(v) = \beta(v)$ *for every* $v \in V$*.*

The cost *of* f *is defined to be*

$$\sum_{vv' \in A} c(vv') f(vv').$$

Our aim is to find a chain partition $\overline{C} = (X, \leqslant_{\overline{C}})$ with precisely w chains that minimises $\widehat{\mathsf{K}}(\overline{C})$. To do this, we will construct a network $\mathcal{N}$ such that the minimum cost flow of $\mathcal{N}$ corresponds to the desired chain partition. We can then find the minimum cost flow of $\mathcal{N}$ in polynomial time.

Every top vertex in (the Hasse diagram of) $\overline{C}$ must have one child and no parent in $\overline{C}$, every bottom vertex must have one parent and no child in $\overline{C}$, and all other vertices must

[10] We will only consider integer flows in this thesis.

have exactly one parent and one child. We cannot represent this requirement directly in a network (i.e. we cannot set lower bounds and upper bounds directly for vertices - only arcs in the network). However, we can use the *vertex splitting procedure* [12] to simulate it. Informally, we will represent the poset $(L, \leqslant)$ as a network in which each vertex is represented as an arc which has lower and upper bound of 1, such that any feasible flow in the network must go through all such arcs (i.e. any feasible flow must pass through each vertex in L exactly once). Specifically, given poset $(L, \leqslant)$, define first a directed graph $D = (V, A)$. Let $L_{\text{in}} = \{x_{\text{in}} : x \in L\}$ and $L_{\text{out}} = \{x_{\text{out}} : x \in L\}$ and define vertex set $V = L_{\text{in}} \cup L_{\text{out}} \cup \{s, t\}$ where $(L_{\text{in}} \cup L_{\text{out}}) \cap \{s, t\} = \emptyset$. Define the arc set A as follows: for $v, v' \in L_{\text{in}} \cup L_{\text{out}}$, $vv' \in A$ if and only if either:

- $v = x_{\text{in}}$ and $v' = x_{\text{out}}$ for some $x \in L$;
- $v = x_{\text{out}}$ and $v' = y_{\text{in}}$ for some $x, y \in L$ such that $y < x$.

For every $v \in L_{\text{in}}$ we have $sv \in A$; and for every $v \in L_{\text{out}}$ we have $vt \in A$.

Then, given a policy $P = (\mathcal{P}, U, O, \lambda)$ where $\mathcal{P} = (L, \leqslant)$, define a network $((V, A), l, u, c, \beta)$, where

$$l(vv') = \begin{cases} 1 & \text{if } v = x_{\text{in}}, v' = x_{\text{out}}, x \in L \\ 0 & \text{otherwise;} \end{cases}$$

$$u(vv') = \begin{cases} 1 & \text{if } vv' \in A \\ 0 & \text{otherwise;} \end{cases}$$

$$c(vv') = \begin{cases} \sum_{z \in \uparrow_{\mathcal{P}} x} |U(z)| & \text{if } v = x_{\text{out}}, v' = t, \text{ where } x \in L \\ 0 & \text{otherwise;} \end{cases}$$

$$\beta(v) = \begin{cases} w & \text{if } v = s \\ -w & \text{if } v = t \\ 0 & \text{otherwise.} \end{cases}$$

We call this network the *network chain-representation of* $(L, \leqslant)$. Note that any feasible flow f for this network must have $0 \leqslant f(xy) \leqslant 1$ for all $xy \in A$.

Lemma 3. *Let $\mathcal{N}$ be the network chain-representation of an information flow policy $P =$*

$(\mathcal{P}, U, O, \lambda)$. Then $\widehat{\mathsf{K}}_{min}$, the minimum number of intermediate secrets required by a chain-based KAS for P with w chains is equal to $\widehat{f}$, where $\widehat{f}$ is the minimum cost of a feasible flow in $\mathcal{N}$.

Proof. Suppose we are given a chain partition $\overline{C} = (L, \leqslant_{\overline{C}})$ with w chains. Consider the following flow:

$$\begin{aligned} f(x_{\text{in}}x_{\text{out}}) &= 1 && \text{for all } x \in L; \\ f(x_{\text{out}}y_{\text{in}}) &= 1 && \text{if } y \lessdot_{\overline{C}} x; \\ f(sx_{\text{in}}) &= 1 && \text{if } x \text{ is the top element in a chain in } \overline{C}; \\ f(x_{\text{out}}t) &= 1 && \text{if } x \text{ is a bottom element in a chain in } \overline{C}; \\ f &= 0 && \text{otherwise.} \end{aligned}$$

Observe that f is a feasible flow. Indeed, by construction all arcs xy satisfy $u(xy) \geqslant f(xy) \geqslant l(xy)$. In the graph formed by arcs xy with $f(xy) = 1$, it is clear that every vertex x has in-degree and out-degree 1, except for s and t. Also, s has in-degree 0 and out-degree w in this graph, and t has in-degree w and out-degree 0. As all arcs xy have $f(xy) = 1$ or $f(xy) = 0$, we have that

$$\sum_{v \in V} (f(xv) - f(vx)) = \beta(x),$$

for all x, as required. Moreover, the cost of f equals $\sum_{b \in B} \sum_{x \in \uparrow b} |U(x)|$, where B is the set of bottom elements of chains in $\overline{C}$, which, by Theorem 2, equals $\widehat{\mathsf{K}}(\overline{C})$. Then, if $\overline{C}$ is a chain partition such that $\widehat{\mathsf{K}}(\overline{C}) = \widehat{\mathsf{K}}_{min}$ (by Theorem 3, such a chain partition exists) then $\widehat{f} = \widehat{\mathsf{K}}_{min}$.

Conversely, suppose f is a feasible flow for $\mathcal{N}$. Then we define $y \lessdot_{\overline{C}} x$ if and only if $x, y \in L$ and $f(x_{\text{out}}, y_{\text{in}}) = 1$. By the construction of $\mathcal{N}$ and definition of f, its not hard to see that $\overline{C}$ is a chain partition of $\mathcal{P}$ with w chains. By construction of $\mathcal{N}$, the cost of f equals $\sum_{b \in B} \sum_{x \in \uparrow b} |U(x)|$ where B is the set of bottom elements of chains in $\overline{C}$, which, by Theorem 2, equals $\widehat{\mathsf{K}}(\overline{C})$. Thus, if the cost of f is $\widehat{f}$ (as required by the theorem), then $\widehat{\mathsf{K}}(\overline{C}) = \widehat{f}$. □

Lemma 4. *We can find a minimum cost flow for $\mathcal{N}$ in $O(|L|^4\, w)$ time.*

Proof. Garg [53] showed that a chain partition with w chains can be obtained in time $O(|L|^{2.5})$. Thus, in particular, we can compute w in time $O(|L|^{2.5})$. To compute $\sum_{x \in \uparrow_{\mathcal{P}} y} |U(x)|$ for each $y \in L$ requires time $O(|A_{\max}| + |L|)$ (where $A_{\max}$ is the arc set of $H^\star(L, \leqslant)$) using depth-first search from y in the digraph obtained from $H^\star(L, \leqslant)$ by changing orientation of every arc. Thus, to compute $\sum_{x \in \uparrow_{\mathcal{P}} y} |U(x)|$ for all $y \in X$ requires time $O(|L|\,(|A_{\max}| + |L|))$.

The well-known buildup algorithm (see [12, §4.10.2], for example) finds a minimum cost flow for a network with n vertices and m arcs in time $O(n^2 m M)$, where M denotes the maximum of all absolute values of balance demands on vertices. By construction of $\mathcal{N}$, we have that $n = 2|L| + 2 = O(|L|)$, $m = O(n^2) = O(|L|^2)$, and $M = w$. Thus we get the desired running time. □

Remark 4. *Strictly speaking, the buildup algorithm assumes that all lower bounds on arcs are* 0*. However, we can satisfy this assumption, given* $\mathcal{N} = (D = (V, A), l, u, c, \beta)$*, by defining the network* $\mathcal{N}' = (D, l', u', c, \beta')$*, where, for each arc* $xy \in A$*:*

$$\begin{aligned} l'(xy) &= 0 & \beta'(x) &= \beta(x) - l(xy) \\ u'(xy) &= u(xy) - l(xy) & \beta'(y) &= \beta(y) + l(xy). \end{aligned}$$

We write $l' \equiv 0$ *to denote that* $l'(xy) = 0$ *for each arc* $xy \in A$*. Then the minimum cost flow* f' *for* $\mathcal{N}'$ *will have cost exactly* $\sum_{xy \in A} l(xy)c(xy)$ *less than the minimum cost flow for* $\mathcal{N}$*, and* f' *can be transformed into a minimum cost feasible flow* f *for* $\mathcal{N}$ *by setting* $f(xy) = f'(xy) + l(xy)$*.*

An example of how to apply the buildup algorithm is provided in Appendix 3.7.

We are now able to prove our main result, which is, essentially, a corollary of Theorem 3 and Lemmas 3 and 4.

Theorem 1. *Let* $P = ((L, \leqslant), U, O, \lambda)$ *be an information flow policy where* $(L, \leqslant)$ *has width* w *and let* $\widehat{\mathsf{K}}_{min}$ *denote the minimum number of intermediate secrets required by a chain-based key assignment scheme for* P*. Then in* $O(|L|^4\, w)$ *time, we can find a chain partition* $\overline{C}$ *for which the corresponding chain-based key assignment scheme only requires* $\widehat{\mathsf{K}}_{min}$ *intermediate secrets and* $\mathsf{k}_{max}(\overline{C}) \leq w$*.*

Proof of Theorem 1. By Theorem 3, there exists a chain partition that has exactly w chains, for which the corresponding chain-based KAS only requires $\widehat{\mathsf{K}}_{min}$ intermediate secrets. Then by Lemma 3, $\widehat{\mathsf{K}}_{min}$ is equal to the minimum cost of a feasible flow in $\mathcal{N}$, the network chain-representation of P. By Lemma 4, such a flow can be found in $O(|L|^4\, w)$ time, and this flow can be easily transformed into the corresponding chain partition $\overline{C}$. Finally, by definition of $\phi(x, \overline{C})$, $|\phi(x, \overline{C})| \leq w$ for each $x \in L$ and therefore no user requires more than w intermediate secrets, i.e. $\mathsf{k}_{\max}(\overline{C}) \leq w$. □

3.6 Adapting Chain-based KASs for Arbitrary Posets

We have described how to construct a chain partition of any given policy poset in order to minimise the maximum number of intermediate secrets required by each user and in total. In order to find such an 'optimal' partition, we used not only information about the policy poset $(L, \leqslant)$, but also the number of users assigned to each label $l \in L$ (i.e. information contained in U and λ). More precisely, we used this information to assign some form of 'weighting' to arcs in the network flow representation of the poset, in order to identify how to best partition the poset into chains such that the number of additional intermediate secrets required by the user population is minimised.

Unfortunately, the current definition of a KAS only considers using the policy poset $(L, \leqslant)$ as input to the SetUp algorithm in order to produce user secrets, label keys and public information to support user derivation. We argue that the SetUp algorithm should therefore take as input the entire information flow policy, and not just the policy poset, such that information about the *entire* policy (e.g. including the distribution of users to labels) can help to optimise how σ_x, κ_x are defined for each $x \in L$.

In addition, in order to compute $\sigma_x = \phi(x, \overline{C})$ for a given chain partition $\overline{C}$ of $(L, \leqslant)$ and for each $x \in L$, as part of the SetUp algorithm defined by Freire *et al.* [48], we require both the partition $(L, \leqslant)$ and $\overline{C}$ to be input into SetUp, however, the current definition of a KAS permits only one of such partial orders to be an input to SetUp. We thus suggest modifying the KAS construction above for chain-based constructions such that the SetUp algorithm takes as input the original policy $((L, \leqslant), U, O, \lambda)$ instead of a chain partition of $(L, \leqslant)$, and an 'optimal' chain partition of the policy poset is found as part of the SetUp

algorithm.

3.7 Example

Given a policy $P = ((L, \leqslant), U, O, \lambda)$, will now provide an example as to how to find a minimum cost flow in the network chain-representation $\mathcal{N}$ of P using the buildup algorithm shown in Figure 3.4. The buildup algorithm requires the use of *residual capacities*. Given a flow f in a network $\mathcal{N} = (D = (V, A), l, u, c, \beta)$, the *residual* capacity $r(xy)$ from x to y (for all $x, y \in V$, $x \neq y$) is [12]:

$$r(xy) = (u(xy) - f(xy)) + (f(yx) - l(yx)).$$

for all $xy \in A$. Given a path $\overline{P} = x_1 x_2 \dots x_n$ in $\mathcal{N}$, we define $r(\overline{P}) = min\left\{r(xy) : xy \in \overline{P}\right\}$.

Intuitively, the residual capacity $r(xy)$ of an arc xy lets us know how many units of flow we can send along arc xy such that the resulting flow between x and y is still feasible. Every time we send a unit of flow along arc xy, we may send a unit of flow along arc yx. This essentially 'cancels' a unit of flow previously sent along xy. Thus, if we send t units of flow along arc xy, we add t to the residual capacity of $r(yx)$.

Suppose we have a network $\mathcal{N} = (V, A, l, u, c, \beta)$ where $xy \in A$, $l(xy) = 2, u(xy) = 5$ and for a feasible flow f in $\mathcal{N}$, $f(xy) = 4$. Then, informally, we can increase $f(xy)$ by up to 1 unit, or can decrease $f(xy)$ by up to 2 units (i.e. send 2 units of flow along arc yx) such that f is still a feasible flow along arc xy. The cost of decreasing the flow (or sending flow along the residual arc yx) is precisely $c(yx) = -c(xy)$ for each unit of flow along yx. Thus, it may be useful to consider finding a flow in the residual network instead, since it enables one to 'travel backwards' along some arcs, but still result in a feasible flow.

Given such residual capacities and network $\mathcal{N}$, a residual network $\mathcal{N}(f)$ with respect to flow f is defined as $((V, A(f)), l, r, c, \beta)$, where $A(f) = \{xy : r(xy) > 0\}$ and $l(xy) = 0$ for all arcs $xy \in A(f)$ [12].

Then, as Bang-Jensen *et al.* [12] state, one can find a feasible flow f in $\mathcal{N}$ and then work in the residual network of $\mathcal{N}(f)$ (see Definition 4.4.1 [12]). As mentioned by Bang-Jensen *et*

The buildup algorithm
Input: A network $\mathcal{N} = (D = (V, A), l \equiv 0, u, c, \beta)$
Output: A minimum cost feasible flow in $\mathcal{N}$ with respect to β or a proof that the problem is infeasible.
Set $\beta_f = \beta$ (the initial balance vector).
Define $Y_f = \{v \in V : \beta_f(v) < \beta(v)\}$ and $Z_f = \{v : \beta_f(v) > \beta(v)\}$

1. Let $a_{xy} = 0$ for every $xy \in A$
2. If $Y_f = \emptyset$ then go to step 8.
3. If there is no (Y_f, Z_f)-path in $\mathcal{N}(f)$, go to step 9
4. Let p and q be chosen such that $p \in Y_f$, $q \in Z_f$ and $\mathcal{N}(f)$ contains a (p, q) path (path from p to q).
5. Find a minimum cost (p, q)-path $\overline{P}$ in $\mathcal{N}(f)$
6. Let $\epsilon = min\left\{r(\overline{P}), \beta(p) - \beta_f(p), \beta_f(q) - \beta(q)\right\}$
7. Let $f = f \oplus \epsilon\overline{P}$; modify Y_f, Z_f and go to Step 2.
8. return f.
9. Return 'no feasible solution'.

Figure 3.4: The buildup algorithm [12].

al. [12]: *"if f is a feasible flow in $\mathcal{N}$ and $f^\star$ is a feasible flow in $\mathcal{N}(f)$ then one can add $f^\star$ to f to obtain a new feasible flow in $\mathcal{N}$".*

We will now give an example of applying the buildup algorithm to a given network $\mathcal{N}$. We will begin by finding an initial flow (path) f in $\mathcal{N}$, and then work in the residual network $\mathcal{N}(f)$.

Example 1. *Let $P = ((L, \leqslant), U, O, \lambda)$ be an information flow policy where $|U(x)| = 1$ for every $x \in L$ and whose Hasse diagram $H(L, \leqslant)$ is shown in Figure 3.1.*

Figure 3.5(a) shows the network chain-representation $\mathcal{N} = ((V, A), l, u, c, \beta)$ of P, where each arc xy in $\mathcal{N}$ is labelled with $(l(xy), u(xy), c(xy))$ (note that some arcs have been omitted for clarity). Vertices s, t are labelled with $\beta(s)$ and $\beta(t)$ respectively (note that $\beta(v) = 0$ for all other vertices $v \in V$). Since the width of $(L, \leqslant)$ is 2, it follows that $\beta(s) = 2$ and $\beta(t) = -2$, and $\beta(v) = 0$ for all other vertices $v \in V$.

In order to apply the buildup algorithm to this network, we must first translate the network into an equivalent network $\mathcal{N}' = ((V, A), l', u', c, \beta')$ where no arc has a non-zero lower bound (see Remark 4). Such a network is shown in Figure 3.5(b) (note again that some arcs have been omitted for clarity). Since $l'(v_{in}v_{out}) = u'(v_{in}v_{out}) = 0$ for all $v \in L$, we omit them from Figure 3.5(b). Then, since $l'(xy) = 0$ for all arcs xy, we simply label each

arc xy in Figure 3.5(b) with values $(u'(xy), c'(xy))$*. Each vertex* $v \in V$ *in Figure 3.5(b) is labelled by* $(v, \beta'(v))$*. Then, we may apply the buildup algorithm (Figure 3.4) to find a minimum cost flow* f *in* $\mathcal{N}'$*.*

In Figures 3.5(b)-3.5(g), vertices with white circles correspond to vertices in Y_f *and white squares correspond to vertices in* Z_f*, as required by the buildup algorithm [12]. Black circles denote vertices that have reached their required balance value (i.e. where* $\beta'_f(v) = \beta'(v)$ *for a given vertex* $v \in V$*). For clarity, we will only label a vertex* v *with* $\beta'_f(v)$ *if* $\beta'_f(v) \neq \beta(v)$*. We will use blue arcs to denote the arcs that are currently part of the flow* f *being constructed, and use dashed lines to denote arcs added in the residual network (i.e. arcs that belong to* $A(f)$ *but not* A*).*

Then, following the procedure of the buildup algorithm, if $Y_f \neq \emptyset$*, we find a path (if one exists) between an element* p *in* Y_f *and* q *in* Z_f *in* $\mathcal{N}'$ *of minimum cost. W.l.o.g. assume that we find the minimum cost path* sf_{in} *from* s *to* f_{in} *of cost 0. Then* $\epsilon = min\left\{r(sf_{in}), \beta'(s) - \beta'_f(s), \beta'_f(q) - \beta'(q)\right\} = min\{1, 1, 1\} = 1$ *and we define* $f = f \oplus 1 \cdot sf_{in}$*. Since* $c(sf_{in}) = 0$*, the current cost of* f *is* 0*. Figure 3.5(c) shows the residual network* $\mathcal{N}'(f)$ *after adding one unit of flow along path* $f_{out}t$ *to flow* f*. Figure 3.5(d) shows the residual network* $\mathcal{N}'(f)$ *after adding one unit of flow along path* $f_{out}t$ *to flow* f*. Since* $c(f_{out}t) = 2$*, the cost of* f *is* 2*. Figure 3.5(e) shows the residual network* $\mathcal{N}'(f)$ *after adding one unit of flow along path* $h_{out}g_{in}$ *to flow* f*. Since* $c(h_{out}g_{in}) = 0$*, the cost of* f *remains the same.*

We may continue to step through the buildup algorithm. The resulting flow f *after adding one unit of flow along paths:* sh_{in}*,* $g_{out}e_{in}$*,* $e_{out}c_{in}$*,* $d_{out}b_{in}$*,* $b_{out}a_{in}$ *and* $c_{out}t$ *is shown in Figure 3.5(f). The resulting residual network* $\mathcal{N}'(f)$ *is shown in Figure 3.5(g). The cost of flow* f *is currently* 8*.*

Note that $Y_f = \{a_{out}\}$ *and* $Z_f = \{d_{in}\}$*. (All other vertices in* $\mathcal{N}'$ *in Figure 3.5(g) have their required balance.) It remains to find a minimum cost path from* a_{out} *to* d_{in}*. Then, we next try to find a minimum cost path from* a_{out} *to* d_{in} *in the residual network of* $\mathcal{N}'(f)$ *(shown in Figure 3.5(g)). The path* $a_{out}tc_{out}a_{in}b_{out}tf_{out}d_{in}$ *(denoted by red arcs in Figure 3.5(d)) is a minimum cost path from* a_{out} *to* f_{out} *of cost 5. (Although an alternative path* $a_{out}f_{out}d_{in}$ *exists, sending one unit of flow along such a path has cost* 6 *and thus does not have minimal cost.) We add one unit of flow along path* $a_{out}c_{out}a_{in}b_{out}tf_{out}d_{in}$ *to* f*.*

Then, since $|Y_f| = 0$*, buildup algorithm terminates and outputs the flow* f *shown in Figure 3.5(h) (via blue arcs), where* $f(xy) = 1$ *for all such arcs. The total cost of* f *is* 13*. Then, by Remark 4,* f *can be transformed into a minimum cost feasible flow* f' *for* $\mathcal{N}$ *by setting* $f'(xy) = f(xy) + l(xy)$ *for all arcs* $xy \in A$*. The resulting flow* f' *is shown in Figure 3.5(i). Then by Remark 4, the cost of* f' *is (also) equal to 13 (we add* $\sum_{xy \in A} l(xy)c(xy)$ *to the cost of* f *to find the cost of* f'*, but since* $l(xy) = 0$ *for all arcs* $xy \in A$ *where* $c(xy) \neq 0$*, the cost of* f' *is also that of* f*).*

Figure 3.5(h) shows the resulting chain partition found when s *and* t *are removed from the minimum cost flow* f'*. By Theorem 1, the total number of intermediate secrets required in a chain-based KAS using the chain partition shown in Figure 3.5(h) for the information flow policy* P *is equal to the cost of* $f' = 13$*.*

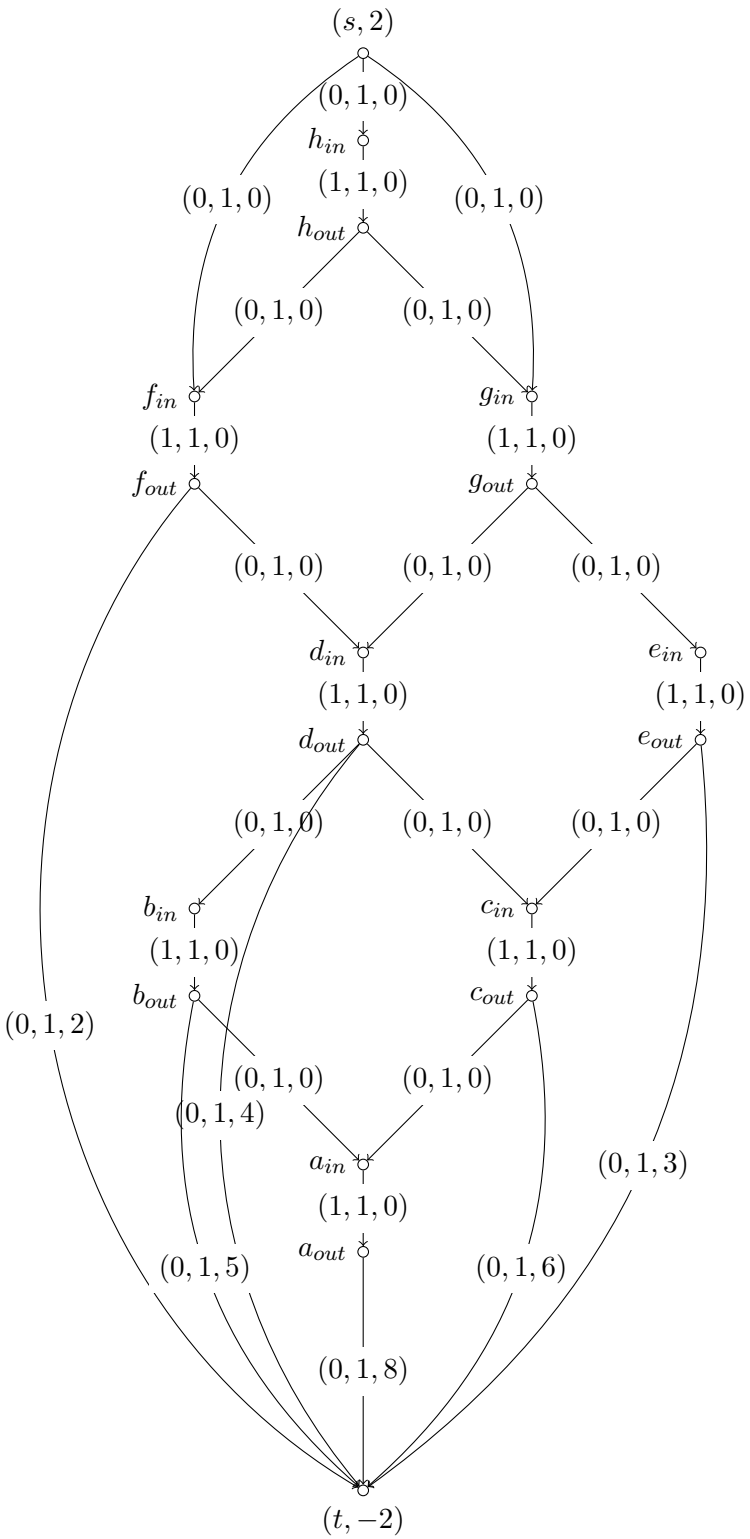


Figure 3.7 (a) Network chain-representation $\mathcal{N}$ of Hasse diagram in Figure 3.1 where each arc $xy \in A$ is labelled with $(l(xy), u(xy), c(xy))$ (some arcs have been omitted for clarity).

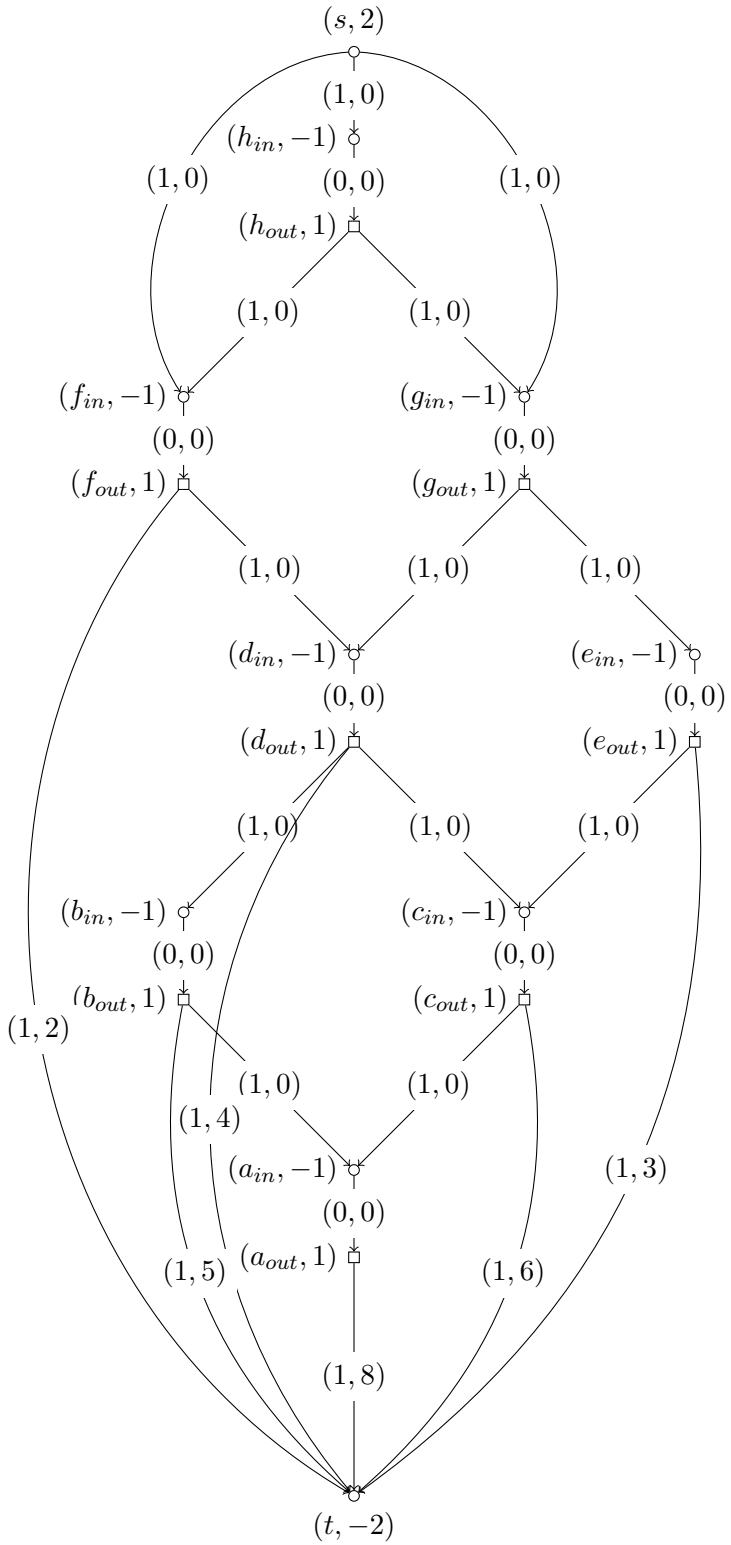


Figure 3.7 (b) Network chain-representation $\mathcal{N}'$ of Hasse diagram in Figure 3.1 where non-zero lower bounds on arcs have been removed (see Remark 4). Each arc xy is labelled by $(u(xy), c(xy))$ and each vertex z is labelled with $\beta'(z)$.

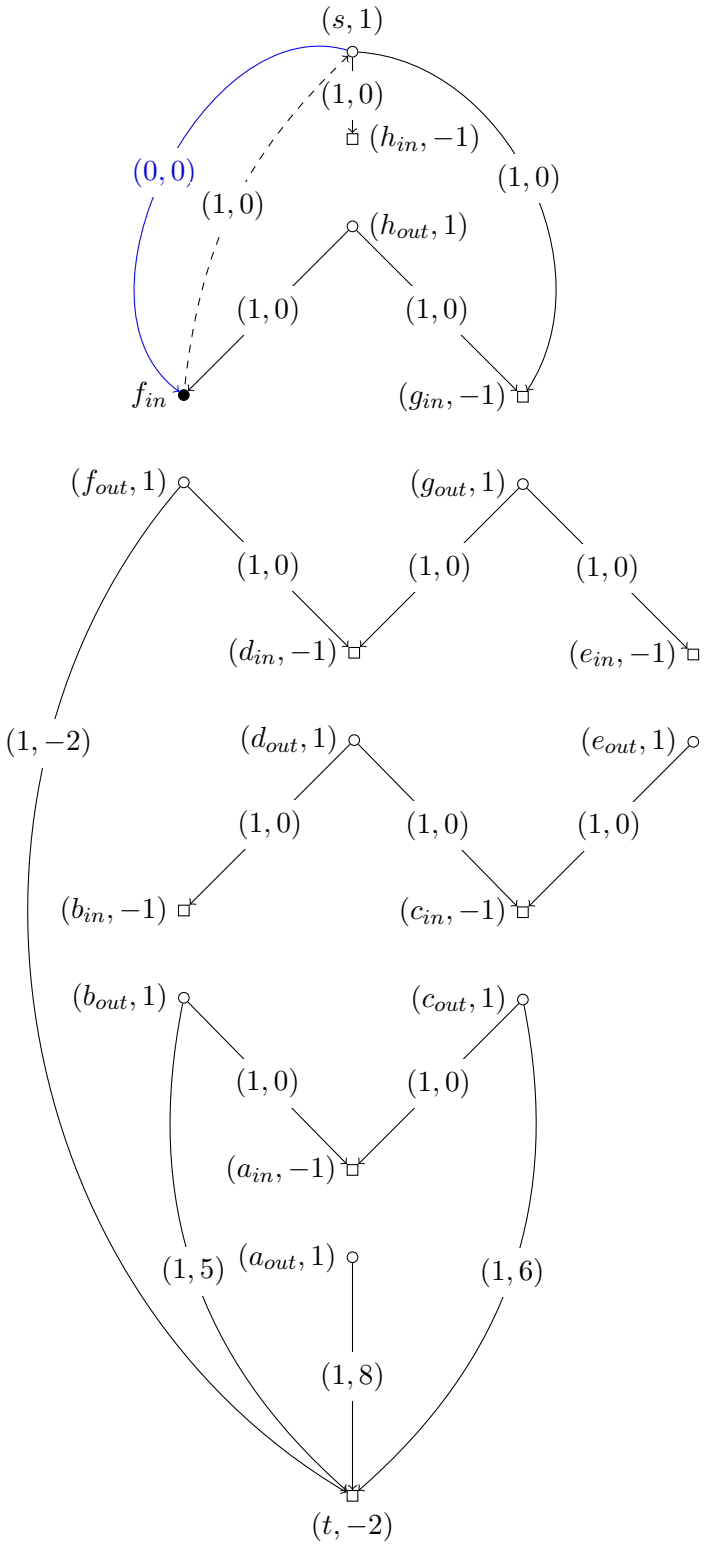


Figure 3.7 (c) Residual network after adding one unit of flow along sf_{in} to flow f. Each arc xy is labelled by $(r(xy), c(xy))$.

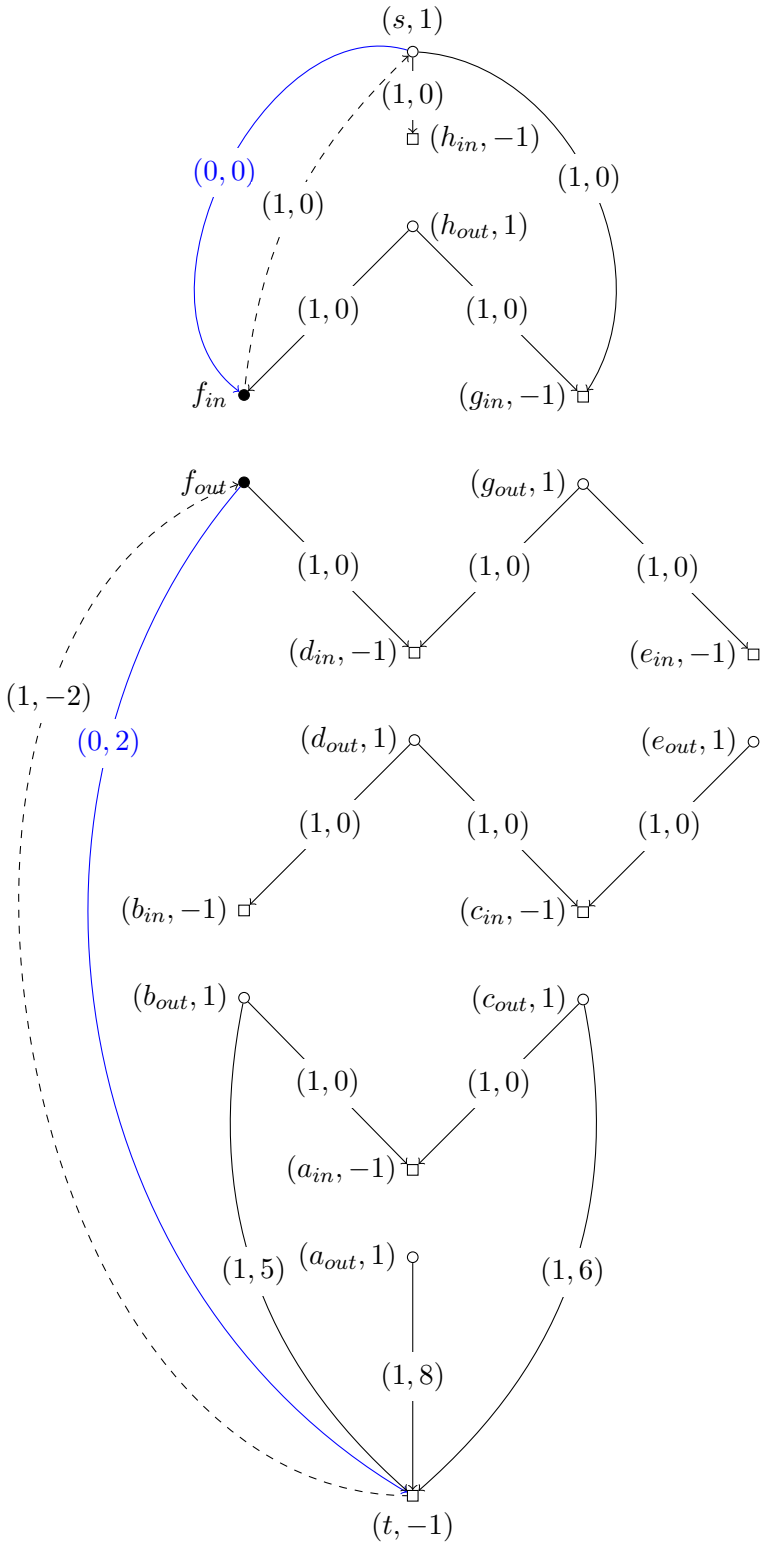


Figure 3.7 (d) Residual network after adding one unit of flow along $f_{out}t$ to flow f.

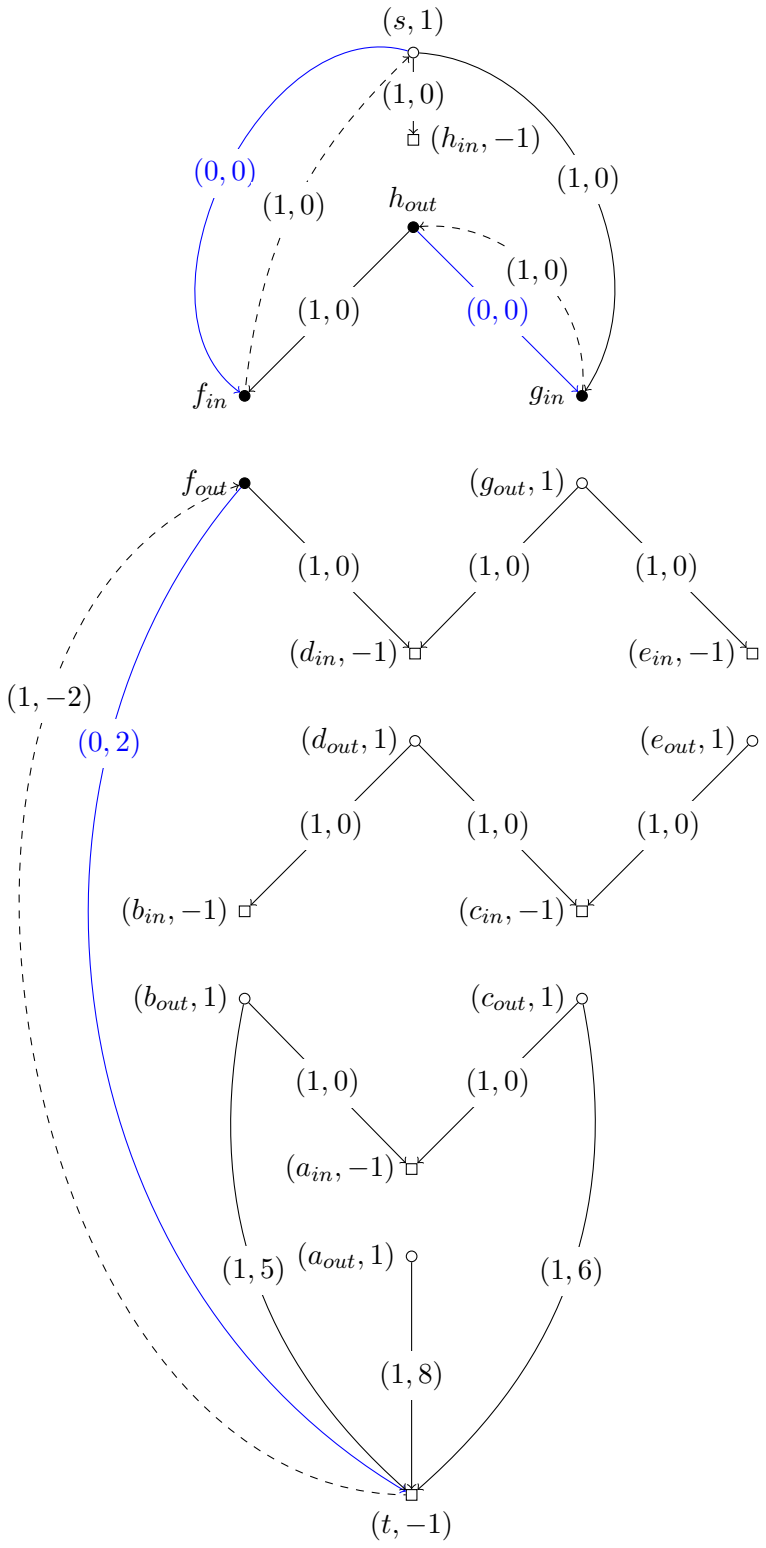


Figure 3.7 (e) Residual network after adding one unit of flow along $h_{out}g_{in}$ to flow f.

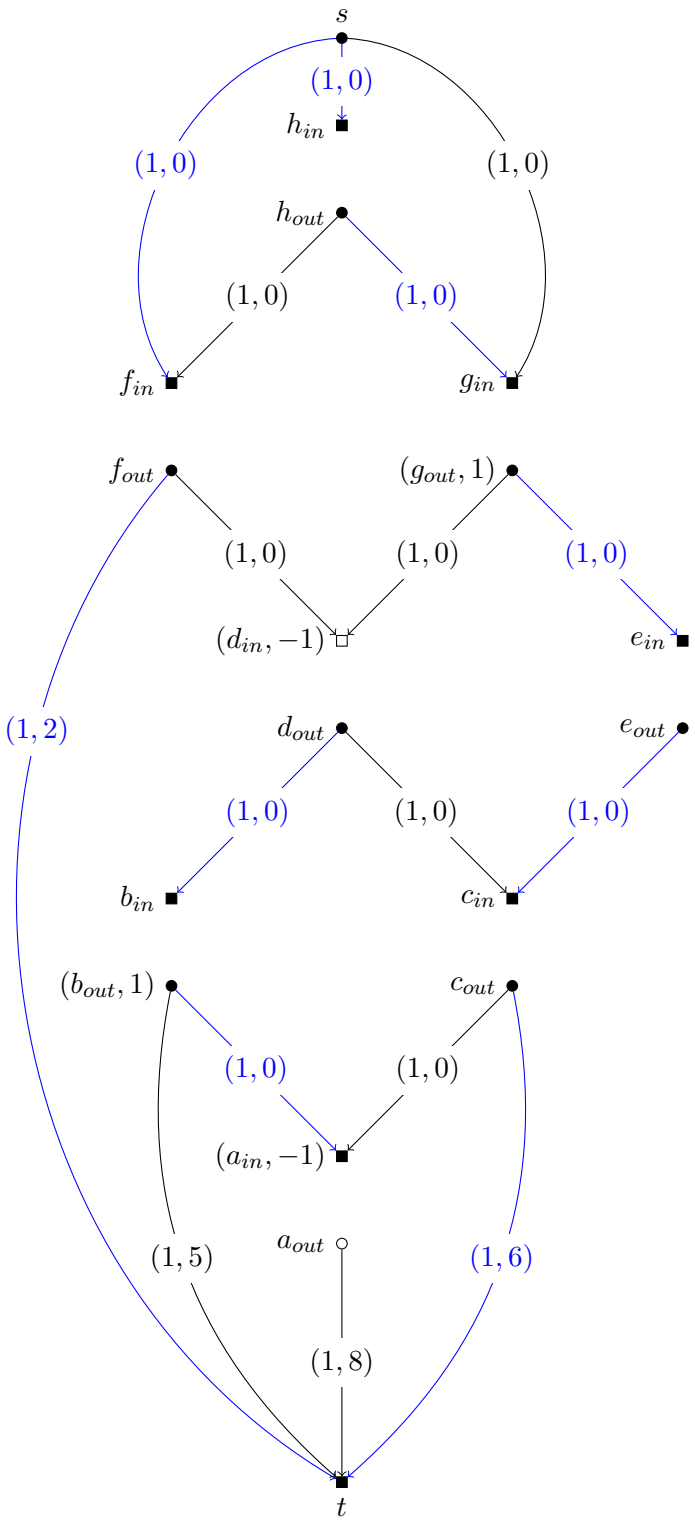


Figure 3.7 (f) Flow f after adding one unit of flow along paths sf_{in}, $f_{out}t$, $h_{out}g_{in}$, $g_{out}e_{in}$, $e_{out}c_{in}$, $d_{out}b_{in}$, $b_{out}a_{in}$ and $c_{out}t$.

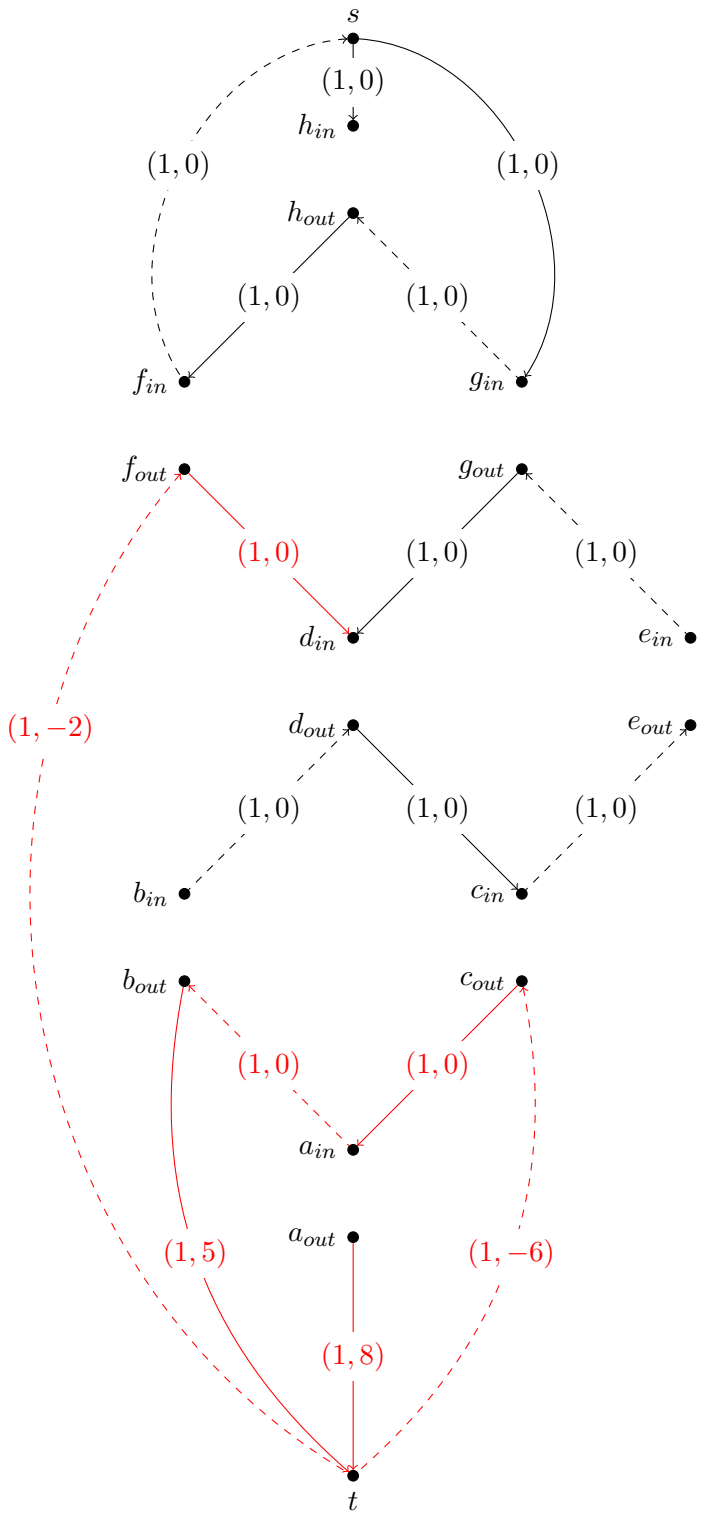


Figure 3.7 (g) Residual network $\mathcal{N}'(f)$ where f is shown in Figure 3.5(f). The path of minimum cost from a_{out} to d_{in}, $a_{out}tc_{out}a_{in}b_{out}tf_{out}d_{in}$ is shown via red arcs.

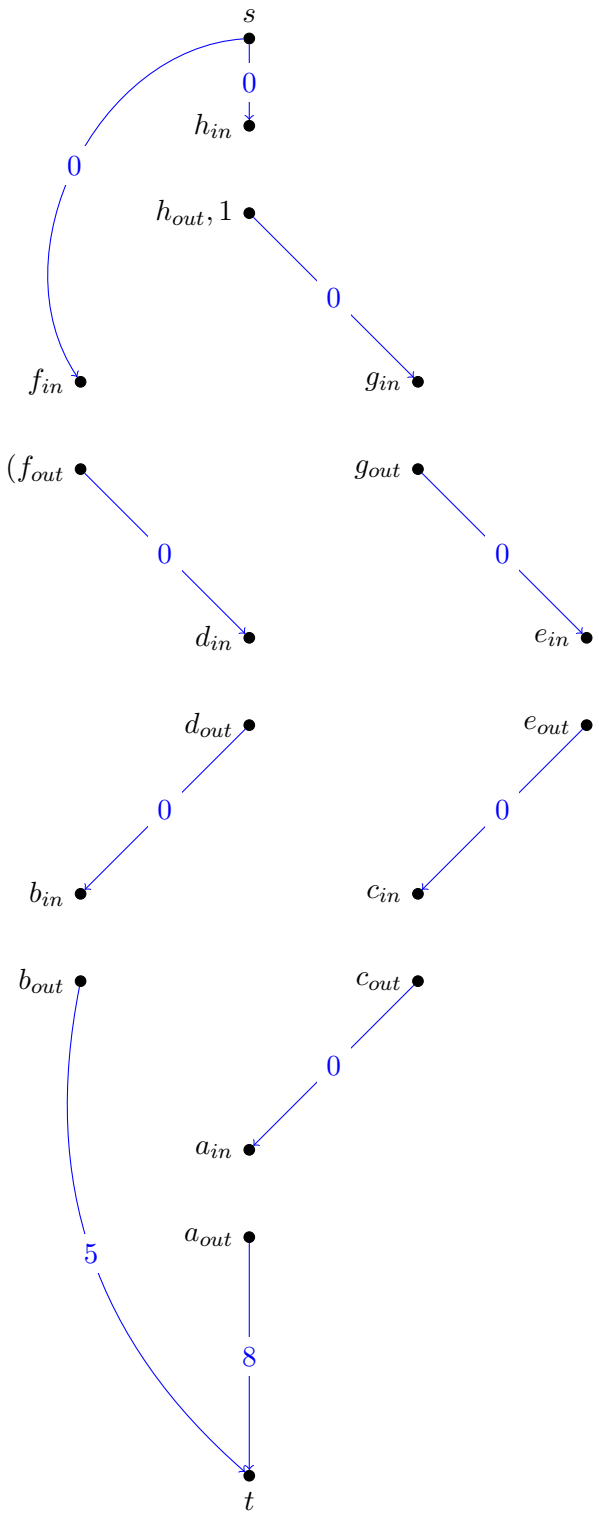


Figure 3.7 (h) Minimum cost flow f found for $\mathcal{N}'$ with total cost 13. Each arc xy is labelled by $c(xy)$.

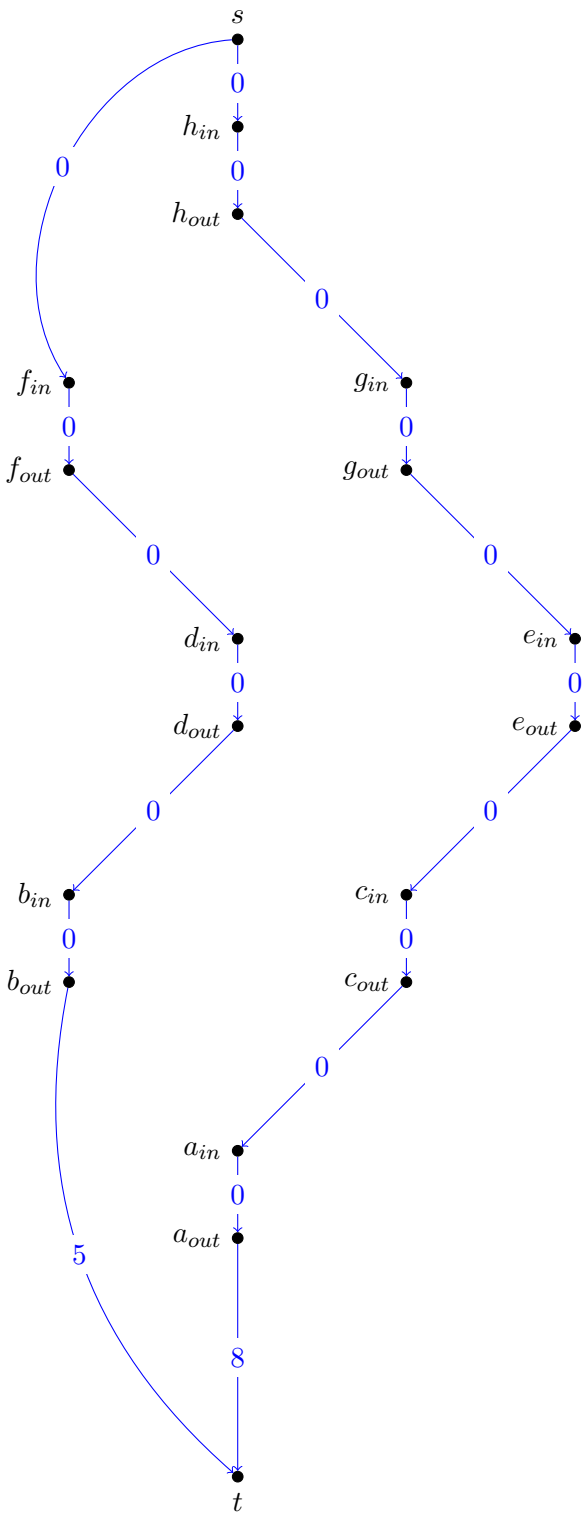


Figure 3.7 (i) Minimum cost flow f' of $\mathcal{N}$ with total cost 13. Each arc xy is labelled by $c(xy)$.

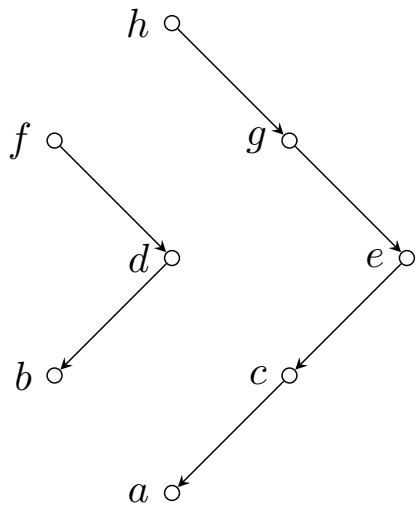


Figure 3.7 (j) Resulting chain partition.

Figure 3.5: Finding an optimal chain partition of Figure 3.1.

3.8 Conclusion

Until recently, symmetric KASs for information flow policies have assumed each user would be given a single intermediate secret, from which other intermediate secrets and decryption keys would be derived using public information generated by the scheme administrator (see, for example, [6, 36]). In this setting, there is a considerable literature on the trade-offs that are possible by reducing the number of steps required for the derivation of secrets, at the cost of increasing the amount of public information (see, for example, [8, 31, 38]).

A drawback of these types of KASs is that the amount of public information required may be substantial and may require a considerable amount of involvement from a trusted party to both provide such information and to also maintain it. Chain-based KASs obviate the requirement for public derivation information, the trade-off being that each user may require several intermediate secrets. The chain-based approach may well be much more practical, particularly if the poset is large and its Hasse diagram contains many arcs (consider the poset being a powerset, for example). Moreover, chain-based KASs may be easily implemented using pseudorandom functions, typically the fastest of cryptographic primitives in practice.

Although chain-based KASs existed before this work, it was not known which choice of chain partition was most appropriate for a given information flow policy. Our work provides formal and practical methods for constructing a chain partition with the smallest number of intermediate secrets in total, with the additional property that no user is required to have more than w intermediate secrets, where w is the width of the information flow policy poset.

Chapter 4

Tree-based Key Assignment Schemes

Contents

In this chapter, we introduce another key assignment scheme for read-only information flow policies that does not require public derivation material. In contrast to the prior chapter that used chain partitions, our KAS in this chapter is built from a tree *partition of the poset. We define a strongly key indistinguishable tree-based KAS and provide a construction for such schemes for a given information flow policy. Such schemes typically require fewer intermediate secrets to be distributed to the entire user population, and can be constructed in less time, than chain-based schemes.*

This chapter is based on the following published works:

- *J. Crampton, N. Farley, M. Jones, G. Gutin and B. Poettering, Cryptographic Enforcement of Information Flow Policies without Public Information, ACNS 2015.*
- *J. Crampton, N. Farley, M. Jones, G. Gutin and B. Poettering, Cryptographic En-*

forcement of Information Flow Policies without Public Information via Tree Partitions, Journal of Computer Security 25(6): 511-535 (2017).

4.1 Introduction

In this chapter, we show that it is possible to work with trees, rather than chains, without reintroducing the need for public derivation information. Informally, we represent the Hasse diagram of the policy poset as an *out-tree* (or tree partition), and define a secure construction for such structures. By definition of an out-tree, there exists a unique root vertex, and each non-root vertex has at most one parent. Thus, we may enforce a poset represented as an out-tree by assigning a random intermediate secret to the root vertex and, similarly to chain-based schemes in Chapter 3, iteratively derive secrets (and keys) down paths in the tree using a PRF without the need for public derivation information.

Intuitively, in tree-based KASs, we are not required to create as many 'breaks' in the policy poset in order to form a tree partition of the poset, compared to that which is required in order to form a chain partition. Since these 'breaks' result in the distribution of additional intermediate secrets, tree-based schemes generally require fewer intermediate secrets to be distributed to the user population than in chain-based schemes. Although Sandhu [79] proposed a KAS for policies whose poset is represented as a tree hierarchy, such a scheme is, at best, only KR secure[1]. Furthermore, such a scheme can only be used to enforce policies already represented as trees.

Contributions. We define a tree-based, SKI-secure KAS and provide a rigorous construction for such schemes for a given information flow policy $((L, \leqslant), U, O, \lambda)$. We identify a number of different parameters that may be important in the context of a tree-based KAS. In particular, we consider the total number of intermediate secrets that may be required in such a scheme and prove that an optimal tree partition and intermediate secret allocation function can be computed in time $\mathcal{O}(|L|^2)$, in comparison to $\mathcal{O}(|L|^4)$ time required to find an optimal chain-based partition (where L is the set of security labels

[1]Informally, the scheme by Sandhu [79] uses a one-way function to define keys iteratively down paths in the tree, however one needs to choose such a function carefully in order to ensure security (the keys output by the function should be indistinguishable from random, and thus a PRF should potentially be used instead).

defined in the given policy). In Section 5.6, we will show that a tree-based KAS for a given information flow policy will often require fewer intermediate secrets than a chain-based KAS. Our approach is based on constructing a weighted directed acyclic graph from the policy poset $(L, \leqslant)$ and then constructing a minimum weight spanning out-tree (see Section 2.1) from the graph. We establish a number of results about this out-tree that are likely to provide the foundation for further study of tree-based KASs.

In Section 4.2, we define a tree-based KAS, provide a method for constructing such schemes for a given information flow policy and prove that the resulting schemes have the property of strong key indistinguishability. In Section 4.3, we address the problem of constructing a tree partition of the policy poset, and propose an associated intermediate secret allocation function, which together minimise the total number of intermediate secrets required to enforce a given policy and can be computed in polynomial time. We conclude this chapter with a summary of our contributions.

4.2 Tree-based Key Assignment Schemes

Let $((L, \leqslant), U, O, \lambda)$ be an information flow policy. Recall from Section 2.1 that a directed acyclic graph $D = (V, A)$ is an *out-tree* if a single vertex $r \in V$ (the root) has in-degree 0, and all other vertices in V have in-degree 1. In the special case that the Hasse diagram $H(L, \leqslant)$ of the policy poset $(L, \leqslant)$ is an *out-tree*, we may use simple cryptographic primitives to enforce an information flow policy. Specifically, we know there is a unique directed path from x to y whenever $y < x$. Hence informally, we may define intermediate secrets and keys for each label in L as follows: we may set s_r (the intermediate secret associated to the root vertex of $H(L, \leqslant)$) to be a random binary string; for all $x, y \in L$ such that $y \lessdot x$, we may define $s_y = \mathcal{F}_{s_x}(y)$; and for all $l \in L$, define $\kappa_l = \mathcal{F}_{s_l}(l)$ and $\sigma_l = s_l$ where $\mathcal{F}$ is an appropriate pseudorandom function [79]. Thus intermediate secrets may be determined by the vertices, rather than the arcs, through which a directed path passes. In this case, we require no public information (apart from a description of the poset), because intermediate secrets and keys are derived only from intermediate secrets and (public) vertex labels.

In general, of course, $H(L, \leqslant)$ is not an out-tree. In the case that $H(L, \leqslant)$ is not an out-

tree, we need to remove arcs from $H^\star(L, \leqslant) = (L, A_{\max})$, the transitive closure of $H(L, \leqslant)$, such that the Hasse diagram of the resulting graph is a spanning out-tree of $H^\star(L, \leqslant)$. Then, similarly to chain-based schemes, we must repair such breaks by including additional intermediate secrets in user secrets. Similarly to Chapter 3, we may assume, without loss of generality, that our policy poset has a maximum element. Thus, we may assume that $H^*(L, \leqslant)$ has only one vertex of in-degree zero and so has a spanning out-tree [12, Prop. 1.7.1] (see Section 2.1 for the definition of a spanning out-tree).

4.2.1 Constructing a Key Assignment Scheme

In this chapter, we investigate ways of constructing a spanning out-tree from $H^*(L, \leqslant) = (L, A_{\max})$ to eliminate the need for public derivation information by selecting an arc set that is a suitable subset of $A_{\max}$. However, removing arcs means that some labels (vertices) are no longer reachable from others, and thus we have to 'repair' these breaks by allocating some users more than one intermediate secret. (Informally, we desire a spanning out-tree instead of a non-spanning out-tree since the latter would involve unneccessarily deleting additional arcs from $H^\star(L, \leqslant)$ for which additional intermediate secrets would have to be distributed in order to fix such breaks.) Our aim is to thus find a set of arcs for deletion such that the total number of intermediate secrets required by the user population is minimised.

Figure 4.1 illustrates three spanning out-trees derived from the poset in Figure 3.1. Removing arcs to create an out-tree inevitably means that certain paths are broken. The out-tree in Figure 4.1a, for example, means that a user associated with vertex h only requires a single intermediate secret and derivation requires no more than one hop. However, every other vertex (except a) requires additional intermediate secrets in order to bridge the gaps. Consider now the out-tree in Figure 4.1b. Users assigned to vertex d and given s_d are authorised for the key for labels a, b, c and d, but cannot derive the keys for labels a and c since there is no path from d to a or c. Thus, in order to repair such breaks, we could define σ_d to contain s_d *and* s_c, from which s_a can also be derived. Then, using s_c and s_a, a user assigned label d can derive the keys κ_c and κ_a respectively. The above observations motivate the following definition.

Definition 10. *Given an information flow policy poset* $(L, \leqslant)$*,* $A_T \subseteq L \times L$ *defines a* derivation out-tree $T = (L, A_T)$ *if:*

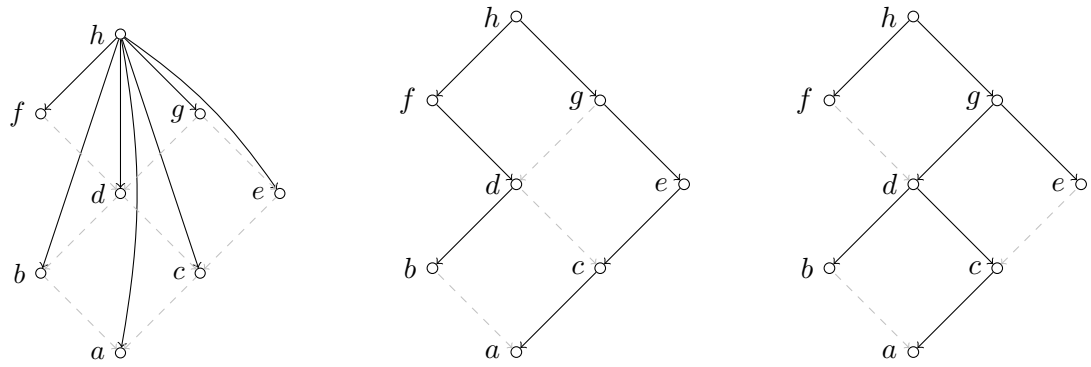


Figure 4.1: Spanning out-trees derived from the poset in Figure 3.1 by arc deletion.

- *T is an out-tree;*
- *$xy \in A_T$ implies $y < x$.*

Since the vertex set of a derivation out-tree T is defined to be L, T is a *spanning* out-tree of $H^\star(L, \leqslant)$.

Lemma 5. *Let $D = (V, A)$ be an acyclic digraph with only one vertex r of in-degree zero. Then by selecting one in-bound arc for each vertex $x \neq r$ we obtain a spanning out-tree T of D. Furthermore, any spanning out-tree of D can be constructed in this way.*

Proof. First, let us prove that T is a spanning out-tree. Clearly, T has no directed cycle and every vertex of $x \neq r$ has in-degree 1. It remains to show that T is connected and contains r. Consider a vertex $y_1 \neq r$ and a longest directed path of T terminating at y_1: $\overline{P} = y_t y_{t-1} \dots y_1$. Since T has no directed cycles all vertices of $\overline{P}$ are distinct and since $\overline{P}$ is longest, it must be that $y_t = r$, else if $y_t \neq r$, then y_t must have an in-bound arc, and thus $\overline{P}$ would not be maximal. Thus, every vertex of T is reachable from r showing that T is connected and contains r. Now let T be a spanning out-tree. Note that for every vertex $x \neq r$ there is exactly one arc to x. Thus, T can be constructed by the procedure of the lemma. □

If $T = (L, A_T)$ is a derivation out-tree of $(L, \leqslant)$ and $x \not> y$, then there is no directed path from x to y, i.e. $x \not\rightarrow_T y$. However, we may have $y < x$ but $x \not\rightarrow_T y$. Thus, the problem with a derivation out-tree, in the context of key assignment schemes, is that some authorised labels will no longer be reachable.

Intuitively then, we want our tree-based KAS's SetUp algorithm to take as input any given information flow policy $((L, \leqslant), U, O, \lambda)$, construct a tree partition T of $H^\star(L, \leqslant)$ and use such a partition to define intermediate secrets and keys for each $x \in L$. Since forming T typically causes breaks in the poset $(L, \leqslant)$, we then desire some function ϕ to determine, for each $x \in L$, what intermediate secrets should be contained in σ_x. We will now discuss ϕ in more detail.

Definition 11. *Given an information flow policy poset $(L, \leqslant)$ and a derivation out-tree T of $(L, \leqslant)$, $\phi : L \to 2^L$ is an* intermediate secret allocation function *if, for all $x \in L$:*

- $x \in \phi(x)$;
- *if $v \leqslant x$ then there exists $z \in \phi(x)$ such that $z \rightsquigarrow_T v$;*
- *if $v \not\leqslant x$ then for all $z \in \phi(x)$, $z \not\rightsquigarrow_T v$.*

Given a derivation out-tree $T = (L, A_T)$ of $(L, \leqslant)$, directed paths in T are used to derive intermediate secrets (and hence keys): A_T determines the paths and ϕ determines the starting points of those paths (and hence the set of intermediate secrets that should be given to each user). In particular, $\phi(x) \setminus \{x\}$ is a set of vertices that were reachable from x in $H(L, \leqslant)$ that are no longer reachable in T. Thus, informally, $\phi(x)$ identifies a set of starting places in T from which all (and only those) vertices that were accessible in $(L, \leqslant)$ from x remain accessible in T, and $|\phi(x)| - 1$ is the number of *additional* intermediate secrets that will be required by a user with security label x.

Ideally then, in order to reduce the number of intermediate secrets each user is assigned, we want to define a 'minimal' secret allocation function ϕ such that, for all $x \in L$, $\phi(x)$ contains the minimal number of intermediate secrets to allow a user assigned to label x to derive all their necessary keys. We define the following function ϕ_{min} to be such a minimal secret allocation function.

Definition 12. *Given a policy poset $(L, \leqslant)$ with maximum element r and a derivation out-tree $T = (L, A_T)$, define $\phi_{min} : L \to 2^L$, where*

$$\phi_{min}(x) = \begin{cases} \{x\} & \text{if } x = r, \\ \{z \in L : \exists y \in L \text{ such that } yz \in A_T, x \geqslant z, x \not\geqslant y\} & \text{otherwise.} \end{cases}$$

We show that: (i) ϕ_{min} is indeed an intermediate secret allocation function; and (ii) for any intermediate secret allocation function ϕ for T, and any $x \in L$, $\phi(x) \supseteq \phi_{min}(x)$.

Lemma 6. *For any poset $(L, \leqslant)$ and any derivation out-tree $T = (L, A_T)$, ϕ_{min} is an intermediate secret allocation function.*

Proof. Following Definition 11, we first show that $x \in \phi_{min}(x)$. This is trivially the case for $x = r$. If x is not the root vertex, there exists $y \in L$ such that $yx \in A_T$ (since T is a derivation out-tree). Then $x \geqslant x$ and $x \not\geqslant y$ (since $yx \in A_T$ implies $x < y$). Hence, by definition 12, $x \in \phi_{min}(x)$.

Now consider the case $v < x$. Since T is a derivation out-tree, there exists a path $z_t z_{t-1} \dots z_0$ in T, with $r = z_t$, $v = z_0$ and $t > 0$. If $z_i = x$ for some i then we are done (since $x \rightsquigarrow_T v$ and $x \in \phi_{min}(x)$). Otherwise, when $z_i \neq x$ for all i, there exists an integer $m < t$ such that $x \geqslant z_m$ and $x \not\geqslant z_{m+1}$. By definition, $z_m \in \phi_{min}(x)$ and also $z_m \rightsquigarrow_T v$.

Finally, consider the case $v \not\leqslant x$ and suppose (in order to obtain a contradiction) there exists $z \in \phi_{min}(x)$ such that $z \rightsquigarrow_T v$. Then $v \leqslant z$ (by definition of a derivation out-tree and $\rightsquigarrow_T$) and $z \leqslant x$ (by definition of $\phi_{min}(x)$). By transitivity, $v \leqslant x$, the desired contradiction. □

Lemma 7. *Given a poset $(L, \leqslant)$ and derivation out-tree T of $(L, \leqslant)$, for any secret allocation function ϕ and every vertex $x \in L$, $\phi(x) \supseteq \phi_{min}(x)$.*

Proof. Clearly $\phi(r) \supseteq \phi_{min}(r)$, by Definition 12. Given $x \neq r$, suppose (in order to obtain a contradiction) that $z \in \phi_{min}(x)$ and $z \notin \phi(x)$. Then, by definition of ϕ_{min}, $x \geqslant z$, and there exists $y \in L$ such that $yz \in A_T$, $x \geqslant z$ and $x \not\geqslant y$. By definition of ϕ, there exists an element t such that $t \rightsquigarrow_T z$ and $t \not\rightsquigarrow_T y$, since $y \not\leqslant x$. Then $t = z$, a contradiction since $z \notin \phi(x)$. Then it must be that $\phi_{min}(x) \subseteq \phi(x)$.

Now, since ϕ is secret allocation function, there exists $t \in \phi(x)$ such that $t \rightsquigarrow_T z$. Since $t \neq z$, it must be that $t \rightsquigarrow_T y$ (since T is a tree and $yz \in A_T$). Therefore, $y \leqslant t$ and $t \leqslant x$, since ϕ is an intermediate secret allocation function. By transitivity, $x \geqslant y$, but this is a contradiction, since $x \leqslant y$. Hence $t = z$. □

Thus, for a given derivation out-tree T, ϕ_{min} is an intermediate secret allocation function that minimises, for each $x \in L$, the number of intermediate secrets required by a user assigned to security label x. Hence, for a given derivation out-tree $T = (L, A_T)$, it is reasonable to assume that we will always use the intermediate secret allocation function ϕ_{min}. Accordingly, we will now write ϕ in preference to ϕ_{min}.

Given an information flow policy $P = ((L, \leqslant), U, O, \lambda)$ and a derivation out-tree T of $(L, \leqslant)$, we define:

$$\mathsf{K}(T) = \sum_{x \in L} |\phi(x)|$$
$$\widehat{\mathsf{K}}(T) = \sum_{x \in L} |U(x)| \cdot |\phi(x)|\,,$$

where $U(x)$ is the set of users assigned to security label x in the policy P.

That is $\mathsf{K}(T)$ represents the total number of intermediate secrets required by a tree-based KAS based on the derivation out-tree T and $\widehat{\mathsf{K}}(T)$ represents the total number of intermediate secrets required to be distributed to the entire user population U in a tree-based key assignment scheme based on T. Note also that $|\phi(x)|$ denotes the number of intermediate secrets required by a user assigned to security label x.

Lemma 8. *Let $(L, \leqslant)$ be an information flow policy poset and let $T = (L, A_T)$ be a derivation out-tree. Then, for all $x \in L$, $\phi(x)$ can be computed in time $O(|L|^2)$.*

Proof. By definition, $\phi(x) = \{z \in L : \exists y \in L$ such that $yz \in A, x \geqslant z, x \not\geqslant y\}$, for any x not equal to r in L. Moreover, there is a single arc in A_T of the form yz, for any $z \in L$, since T is a derivation out-tree. Thus, an algorithm to compute ϕ comprises an outer loop which iterates through the elements of L and an inner loop that iterates through the elements of A_T, where each iteration of the inner loop for arc yz tests whether $x \geqslant z$ and $x \not\geqslant y$. We can compute the adjacency matrix of $H^*(L, \leqslant)$ in time $O(|L|^2)$, which we can use to test whether $x \geqslant z$ (and $x \not\geqslant y$) in constant time. Moreover, $|A_T| = |L| - 1$ (since every vertex except the root has in-degree 1). Thus our algorithm runs in time $O(|L|^2)$. □

4.2.2 Generating Secrets and Keys

We now describe how to instantiate a tree-based key assignment scheme for $(L, \leqslant)$, given an information flow policy $((L, \leqslant), U, O, \lambda)$, using a pseudorandom function (PRF). As mentioned in Section 3.6, we believe that a KAS that takes as input the entire information flow policy and not just the policy poset is better able to optimise the characteristics of the associated KAS (e.g. the number of intermediate secrets required by the user population; see Section 1.1 for other important KAS characteristics that may be optimised). The scheme is a natural extension of the one used by Freire *et al.* for total orders [47].[2] Let ρ be a security parameter and $\mathcal{F}\colon \{0,1\}^\rho \times \{0,1\}^* \to \{0,1\}^\rho$ be a PRF (as formally introduced in Definition 3). Note that we define the domain and range of the PRF to be the same.

Setup: The inputs to the algorithm are ρ and the information flow policy $P = ((L, \leqslant), U, O, \lambda)$.

1. If $(L, \leqslant)$ has no unique maximal element, add a maximal element r such that for every $x \in L$, $x \lessdot r$ and $|U(r)| = 0$.
2. Construct a derivation out-tree $T = (L, A_T)$ for $(L, \leqslant)$, with root vertex r.
3. Select secret value s_r uniformly at random from $\{0,1\}^\rho$.
4. Set

$$\kappa_r \stackrel{\text{def}}{=} \mathcal{F}_{s_r}(r)$$

   and, recursively, if y is a child of vertex x (in T), set

$$s_y \stackrel{\text{def}}{=} \mathcal{F}_{s_x}(y)$$
$$\kappa_y \stackrel{\text{def}}{=} \mathcal{F}_{s_y}(y)$$

   Thus, for $xy \in A_T$, s_y is derived from s_x and the label of y, while κ_y is derived from s_y and the label of y.
5. For each $x \in L$, define $\sigma_x = \{(y, s_y) : y \in \phi(x)\}$.
6. Set $Pub \leftarrow T$.

[2] In the special case of a total order, we obtain the scheme of Freire *et al*, modulo some differences in the choice of the second input to the PRF.

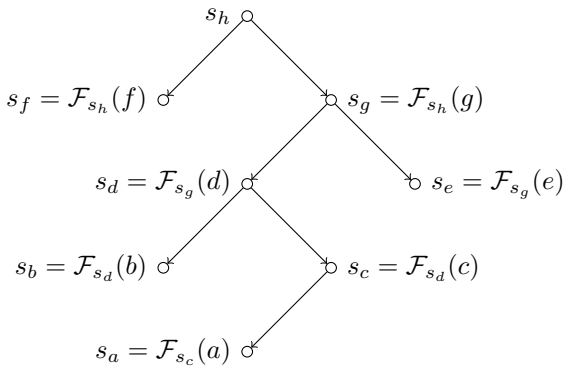


Figure 4.2: The secrets generated for the spanning-out-tree in Figure 4.1c.

7. Return $(\{\sigma_x, \kappa_x\}_{x \in L}, Pub)$.

Derive: Given y, x where $y \leqslant x$, and σ_x where $\sigma_x = \{(l, s_l) : l \in \phi(x)\}$, there (uniquely) exists $(z, s_z) \in \sigma_x$ such that $z \rightsquigarrow_T y$.

If $z = y$, then (since $(z, s_z) \in \sigma_x$), compute $\kappa_z = \mathcal{F}_{s_z}(z)$. If $z \neq y$, then for each intermediate vertex t_i on the path $t_1 \ldots t_m$ between $t_1 = z$ and $t_m = y$ in T, compute $s_{t_i} = \mathcal{F}_{s_{t_{i-1}}}(t_i)$. Finally, compute and return $\kappa_y = \mathcal{F}_{s_y}(y)$.

Our method for generating intermediate secrets is illustrated in Figure 4.2.

4.2.3 Security Analysis

We now prove that our tree-based KAS from Section 4.2.2 is strongly key indistinguishable. Observe that this implies that our scheme is secure in all the models considered in [6, 47]. Because we modify our definition of a KAS to take as input the entire policy $P = ((L, \leqslant), U, O, \lambda)$ instead of just the policy poset $(L, \leqslant)$, we slightly modify the SKI game $\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$ presented in Definition 6 accordingly. Most notably, the Setup algorithm now takes as input P instead of $(L, \leqslant)$ and the adversary is given P instead of $(L, \leqslant)$. We denote this modified experiment by $\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, P, x)$ and denote the advantage of an adversary against this experiment by $\mathrm{Adv}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, P, x)$. For compactness, we will denote the advantage $\mathrm{Adv}^{\mathsf{ind-prf}}_{\mathcal{F},\mathcal{D}_i^b}(1^\rho)$ of a distinguisher $\mathcal{D}_i^b$ in distinguishing a PRF $\mathcal{F}$ from a random function (see Definition 3) as $\mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_i^b}$. More formally, we have the following result.

Theorem 4. *For any information flow policy $P = ((L, \leqslant), U, O, \lambda)$, $x \in L$, and efficient adversary $\mathcal{A}$, there exists a constant $0 \leqslant c \leqslant |L|$ and efficient distinguishers $\mathcal{D}_1^0, \ldots, \mathcal{D}_c^0$,*

$\mathcal{D}_1^1, \ldots, \mathcal{D}_c^1$ *against the underlying PRF such that*

$$\mathrm{Adv}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, P, x) \quad \leqslant \quad \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_1^0} + \cdots + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_c^0} + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_1^1} + \cdots + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_c^1} \ .$$

Proof. The argument proceeds using sequences of $|L| = n$ hybrid games that interpolate between experiments $\mathbf{Exp}^{\mathsf{SKI-0}}_{\mathcal{KAS},\mathcal{A}}$ and $\mathbf{Exp}^{\mathsf{SKI-1}}_{\mathcal{KAS},\mathcal{A}}$. In each hybrid step, if specific conditions are met, we replace one PRF instance by a random function; from the point of view of the adversary, the distance between each two consecutive hybrids is not greater than $\mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_i^b}$, for an appropriate PRF distinguisher $\mathcal{D}_i^b$.

Fix a policy poset $(L, \leqslant)$, a derivation out-tree $T = (L, A_T)$ for $(L, \leqslant)$, a label $x \in L$, and an efficient adversary $\mathcal{A}$. Let $x_n \prec x_{n-1} \prec \cdots \prec x_2 \prec x_1 = r$ be any (reverse) linear extension of L; that is x_n is a minimal element in L and x_1 is the root.[3] For $b \in \{0,1\}$, we set $G_0^b = \mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, P, x)$ and define games $G_1^b, \ldots, G_n^b$ such that, if $x \not\leqslant x_i$ then G_i^b and G_{i-1}^b are identical, and if $x \leqslant x_i$ then the difference between games G_i^b and G_{i-1}^b is precisely that all PRF invocations with key κ_{x_i} are replaced by assignments with values in $\{0,1\}^\rho$ drawn uniformly at random. Let S_i^b denote $\Pr[G_i^b \to 1]$ for all b, i.

Observe that we replace PRF invocations by random assignments for precisely those labels x that do not have a corresponding entry in $Corrupt_x$. Observe also that, as we consider labels $x_i \in L$ in a suitable order, for all switchings from a PRF to a random function we have that the corresponding PRF key κ_{x_i} was replaced with a uniform random value beforehand. That is, if $x \leqslant x_i$, the difference between games G_{i-1}^b and G_i^b is that $\kappa_{x_i} = \mathcal{F}_{s_{x_i}}(x_i)$ in game G_{i-1}^b and κ_{x_i} is a randomly generated string in $\{0,1\}^\rho$ in game G_i^b. We now argue that if an efficient PRF distinguisher $\mathcal{D}_i^b$ exists that can distinguish between these two games, then an adversary, using the distinguisher as a subroutine, can distinguish the two cases (by taking the distinguisher's guess of b as his own). By the security of our PRF, no such distinguisher exists. Thus, in the cases $x \leqslant x_i$ we have

$$|S_i^b - S_{i-1}^b| = |\Pr[G_i^b \to 1] - \Pr[G_{i-1}^b \to 1]| \leqslant \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_i^b}, \tag{4.1}$$

for a specific distinguisher $\mathcal{D}_i^b$. In addition, whenever $x \not\leqslant x_i$ we have $G_i^b = G_{i-1}^b$, and

[3] That is, if $x \leqslant y$ (in L) then $x \preccurlyeq y$ (in the linear extension). Every (finite) partial order has at least one linear extension, which may be computed, in linear time, by representing the partial order as a directed acyclic graph and using a topological sort [27, §22.3].

hence $|S_i^b - S_{i-1}^b| = 0$. Now, by repeated application of the triangle inequality and (4.1), we have

$$\left|S_0^b - S_n^b\right| \leqslant \sum_{i=1}^{n} \left|S_{i-1}^b - S_i^b\right| \leqslant \sum_{i=1}^{c} \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_i^b},$$

where $c = |\{x' \in L : x \leqslant x'\}|$ and distinguishers $\mathcal{D}_i^b$ are constructed as specified. We now consider games G_n^0 and G_n^1. In both cases κ_x is picked uniformly at random. Hence G_n^0 is identical to G_n^1 and $\left|S_n^1 - S_n^0\right| = 0$. Thus, we obtain:

$$\begin{aligned} \mathrm{Adv}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, P, x) = |S_0^1 - S_0^0| &\leqslant |S_0^1 - S_n^1| + |S_n^1 - S_n^0| + |S_n^0 - S_0^0| \\ &\leqslant \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_1^1} + \ldots + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_c^1} + 0 + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_1^0} + \ldots + \mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}_c^0}, \end{aligned}$$

as required. □

4.3 Minimising $\widehat{\mathsf{K}}$ in a Tree-based Key Assignment Scheme

So far, we have shown that it is possible to construct a tree-based KAS for an information flow policy $((L, \leqslant), U, O, \lambda)$ that is SKI-secure. As we observed before, we will usually require our tree-based KAS to have some particular properties, such as minimising the total number of intermediate secrets or ensuring that all derivation paths are no longer than some threshold value. Hence, we require an algorithm to compute a derivation out-tree that satisfies the desired requirements; by Lemma 8, we can then compute the associated intermediate secret allocation function ϕ in polynomial time.

In this section, we consider two questions: how to minimise K, the total number of intermediate secrets allocated to vertices (by the intermediate secret allocation function ϕ); and how to minimise $\widehat{\mathsf{K}}$, the total number of intermediate secrets distributed to users. The second question is interesting because, in practice, we might want to reduce the exposure of intermediate secrets (in transmission to users) by ensuring that very few intermediate secrets are associated with vertices to which many users are assigned. We solve both questions, demonstrating that it is surprisingly efficient to compute: (i) an 'optimal' derivation out-tree (that minimises the total number of intermediate secrets required in the associated KAS); and (ii) the minimal set of intermediate secrets each user secret σ_x should comprise. We then state and prove Theorem 6, the main result of this section.

Recall that $A_{\max}$ is the arc set of the transitive closure $H^\star(L, \leqslant)$ of the Hasse diagram of the policy poset. Our basic approach is to define a weight for each arc in $A_{\max}$ and construct a minimum weight spanning out-tree. Intuitively, for each arc xy in $H^\star(L, \leqslant)$, we define a weight corresponding to the number of users who will require the intermediate secret s_y if xy is chosen to be in the derivation out-tree T and all other arcs to y are omitted (since each vertex in T can have at most one incoming arc). Thus by selecting a spanning out-tree of minimum weight, we minimise the number of intermediate secrets that need to be distributed in the corresponding tree-based key assignment scheme.

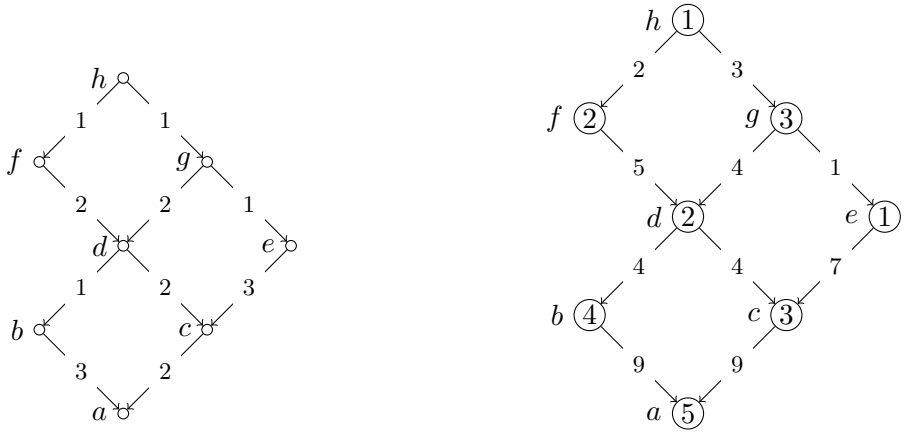
In order to define such weights we will make use of the γ function, defined in Definition 7. Recalling Definition 7, given an information flow policy poset $(L, \leqslant)$, for each arc yz in $H^\star(L, \leqslant)$:

$$\gamma(yz) = \{x \in L : x \geqslant z, x \not\geqslant y\}\,.$$

Then, given an information flow policy $((L, \leqslant), U, O, \lambda)$, where $\lambda : U \cup O \to L$, $U(x) = \{u \in U : \lambda(u) = x\}$, we define the *weight function* $\omega : A_{\max} \to \mathbb{N}$, where, for each arc yz in $H^\star(L, \leqslant) = (L, A_{\max})$,

$$\omega(yz) \stackrel{\text{def}}{=} \sum_{x \in \gamma(yz)} |U(x)|\,.$$

Figure 4.3a shows the values of $|\gamma(xy)|$ for each arc xy in the Hasse Diagram in Figure 3.1. In Figure 4.3b, we label each vertex x with $|U(x)|$, and use these values to compute $\omega(xy)$ for each arc xy.



(a) $|\gamma(xy)|$ for each arc xy.

(b) $\omega(xy)$ for each arc xy.

Figure 4.3: $|\gamma(xy)|$ and $\omega(xy)$ for each arc in Figure 3.1.

Intuitively, $\gamma(yz)$ is the set of vertices that can no longer reach z via a path if the arc yz

is chosen to be in the derivation out-tree and all other arcs to z are removed since every non-root vertex can only have one incoming arc; $\omega(yz)$ is the number of users that must be issued s_z if arc yz is selected to be in the derivation out-tree. We will now establish the relation between the total number of intermediate secrets that have to be issued and the weight function ω, beginning by examining the non-root vertices.

Theorem 5. *Let $T = (L, A_T)$ be any derivation out-tree for $(L, \leqslant)$. Then*

$$\sum_{\substack{x \in L \\ x \neq r}} |U(x)| \cdot |\phi(x)| = \sum_{yz \in A_T} \omega(yz).$$

Proof. By definition, we have, for every $x \neq r$,

$$\begin{aligned} |\phi(x)| &= |\{z \in L : \exists y \in L \text{ such that } yz \in A_T, x \geqslant z, x \not\geqslant y\}| \\ &= |\{yz \in A_T : x \in \gamma(yz)\}|, \end{aligned}$$

and so

$$|U(x)| \cdot |\phi(x)| = |U(x)| \cdot |\{yz \in A_T : x \in \gamma(yz)\}|.$$

Hence

$$\sum_{\substack{x \in L \\ x \neq r}} |U(x)| \cdot |\phi(x)| = \sum_{\substack{x \in L \\ x \neq r}} |U(x)| \cdot |yz \in A_T : x \in \gamma(yz)|,$$

and, since $r \notin \gamma(yz)$ for any $yz \in A_T$, we have:

$$\sum_{\substack{x \in L \\ x \neq r}} |U(x)| \cdot |\phi(x)| = \sum_{yz \in A_T} \sum_{x \in \gamma(yz)} |U(x)| = \sum_{yz \in A_T} \omega(yz).$$

□

Theorem 6. *Given an information flow policy $((L, \leqslant), U, O, \lambda)$, we can compute a derivation out-tree $T = (V, A_T)$ and compute $\phi(x)$ for all $x \in L$ such that $\widehat{\mathsf{K}}(T)$ is minimised in time $O(|A_{\max}| + |L|^2)$.*

Proof. By Theorem 5,

$$\widehat{\mathsf{K}}(T) = |U(r)| + \sum_{yz \in A_T} \omega(yz).$$

An algorithm to compute the weight function ω iterates through the arcs in $A_{\max}$ and, for a given arc yz, iterates through all x in L testing whether $x \geqslant z$ and $x \not\geqslant y$. In other words, we swap the inner and outer loops in the algorithm used in the proof of Lemma 8. Thus, we can compute ω in time $O(|L|^2)$.

Since $|U(r)|$ is fixed, we minimise $\widehat{\mathsf{K}}$ by computing a derivation out-tree that minimises $\sum_{yz \in A_T} \omega(yz)$. By Lemma 5, we can achieve this by selecting, for each non-root vertex $x \in L$, the minimum weight arc to x, where the weights are given by ω. We need only consider each arc (in $A_{\max}$) once, which takes time $O(|A_{\max}|)$. The resulting set of arcs forms a spanning out-tree of minimum weight and the number of additional keys required is $\sum_{yz \in A_T} \omega(yz)$. We can derive the associated intermediate secret allocation function in time $O(|L|^2)$, by Lemma 8; the result follows. □

Corollary 2. *Given an information flow policy $((L, \leqslant), U, O, \lambda)$, we can compute a derivation out-tree T and ϕ such that K is minimised in its associated tree-based KAS in time $O(|A_{\max}| + |L|^2)$.*

Proof. We simply set $|U(x)| = 1$ and apply Theorem 6. □

Corollary 3. *We can find, in time $O(|A_{\max}| + |L|^{3/2} |A_{\max}|^{1/2})$, a minimum weight spanning out-tree that has the minimum number of leaves amongst such trees.*

Proof. Replace $H^*(L, \leqslant)$ by its subgraph $D = (L, E)$ obtained as follows: for each vertex $x \neq r$ delete all arcs to x apart from those of minimum weight (among arcs to x). Observe that D can be constructed in time $O(|A_{\max}|)$. Find an out-tree with minimum number of leaves using the MinLeaf algorithm [60] (see Figure 4.4). It remains to observe that MinLeaf's runtime is $O(|E| + |L|^{3/2} |E|^{1/2})$. □

It is useful to find a minimum weight spanning out-tree with a minimum number of leaves because the number of leaves will impose an upper bound on $|\phi(x)|$. Note, however, that $|\phi(x)|$ may be greater than the width of $(L, \leqslant)$ (and it is not difficult to construct such an example). This is because the set of arcs in the graph that is input to MinLeaf – the

The minleaf algorithm

Input: An acyclic digraph $D = (V, A)$ with vertex set V.

Output: A minimum leaf out-branching T of D if minimum number of leaves in D is greater than 0 and "NO", otherwise.

1. Find a source r in D. If there is another source in D, return "no out-branching". Let $V' = \{v : v \in V\}$.
2. Construct a bipartite graph $B = B(D)$ of D with partite sets V, $V' \setminus r'$ and edge xy' for each arc $xy \in D$.
3. Find a maximum matching M in B.
4. $M^\star := M$. For all $y' \in V'$ not covered by M, set $M^\star := M \cup \{\text{an arbitrary edge incident with } y^\star\}$.
5. $E := \emptyset$. For all $xy' \in M^\star$, set $E := E \cup xy$ (where E is a set of directed edges).
6. Return $T = (V, E)$.

Figure 4.4: MINLEAF algorithm [60].

algorithm used to construct the spanning out-tree – will, in general, be a strict subset of $A_{\max}$ (and thus the graph is less connected than $(L, A_{\max})$). Thus, the size of the maximal independent set in the graph that is input to MinLeaf can exceed the width of the poset (which is the equal to the size of the maximal independent set in $(L, A_{\max})$). We now prove some further properties of γ. This enables us to reduce the running time of our algorithm because we show it is sufficient to consider only arcs in $A_{\min}$ (rather than $A_{\max}$) when constructing the minimum weight spanning out-tree.

Lemma 9. *Let $(L, \leqslant)$ be a partially ordered set. Then for all $x, y, z \in L$ such that $z < y < x$,*

$$\gamma(xy) \cap \gamma(yz) = \emptyset \quad \textit{and} \quad \gamma(xz) \supseteq \gamma(yz) \cup \gamma(xy).$$

Proof. Suppose $t \in \gamma(xy) \cap \gamma(yz)$. Since $t \in \gamma(yz)$, we have $t \geqslant z$ and $t \not\geqslant y$; since $t \in \gamma(xy)$, we have $t \geqslant y$, immediately leading to the desired contradiction.

Now suppose $t \in \gamma(xy)$. Then $t \geqslant y$ and $t \not\geqslant x$. Hence, we have $t > z$, by transitivity; thus $t \in \gamma(xz)$ and $\gamma(xy) \subseteq \gamma(xz)$. Finally, suppose $t \in \gamma(yz)$. Then $t \geqslant z$ and $t \not\geqslant y$. Now $t \not\geqslant x$ (otherwise, we would have $t > y$ by transitivity) and hence $t \in \gamma(xz)$; thus $\gamma(yz) \subseteq \gamma(xz)$. □

Corollary 4. *Let $(L, \leqslant)$ be a partially ordered set with Hasse diagram $H(L, \leqslant)$ and whose transitive closure is $H^\star(L, \leqslant)$. Then, for any path $x_1 x_2 \dots x_p$ in*

$H^\star(L,\leqslant)$, $p>2$, we have:

$$\omega(x_1x_p) \geqslant \sum_{i=1}^{p-1}\omega(x_ix_{i+1}).$$

Proof. Consider the case $p=3$, with path xyz where $x>y>z$. Using Lemma 9 and the fact that $|U(t)|\geqslant 0$ for all t, we have:

$$\begin{aligned}\omega(xz) &= \sum_{t\in\gamma(xz)}|U(t)| \qquad\qquad \text{(by definition)}\\ &\geqslant \sum_{t\in\gamma(xy)}|U(t)| + \sum_{t\in\gamma(yz)}|U(t)|\\ &= \omega(xy)+\omega(yz).\end{aligned}$$

Now suppose the result holds for all $p<N$ and consider a path $x_1\dots x_N$ containing N vertices. Then $x_1x_{N-1}\in A_{\max}$ and, by Lemma 9 and the inductive hypothesis, respectively, we have:

$$\begin{aligned}\omega(x_1x_N) &\geqslant \omega(x_1x_{N-1})+\omega(x_{N-1}x_N)\\ &\geqslant \omega(x_1x_2)+\cdots+\omega(x_{N-2}x_{N-1})+\omega(x_{N-1}x_N)\\ &= \sum_{i=1}^{N-1}\omega(x_ix_{i+1}).\end{aligned}$$

Thus the result holds by induction. □

We now show that a minimum weight derivation out-tree for a given policy poset $(L,\leqslant)$ can always be found from its Hasse diagram (i.e. we do not need to consider its transitive closure). Thus, as we will show in Corollary 6, we can construct a minimum weight derivation out-tree in less time by only considering arcs in the Hasse diagram of $(L,\leqslant)$.

Corollary 5. *Let $(L,\leqslant)$ be a partially ordered set with Hasse diagram $H=(L,A_{\min})$. Then there exists a minimum weight spanning out-tree $T=(L,A_T)$ with $A_T\subseteq A_{\min}$.*

Proof. Let $T'=(L,A'_T)$ be a minimum weight spanning out-tree for $(L,\leqslant)$, and suppose arc xy is in A'_T but not in A_T. Then $x\rightsquigarrow_H y$ and let zy be the last arc in this path. Since $\omega(uv)\geqslant 0$ for each arc uv and by Corollary 4, $\omega(zy)\leqslant\omega(xy)$. Therefore by removing xy from A'_T and adding zy, we have a spanning out-tree with weight at most that of T'. By

replacing every arc in $A'_T \setminus A_{\min}$ in this way, we have a spanning out-tree $T = (L, A_T)$ of weight at most that of T', and therefore of minimum weight. □

Corollary 6. *Given an information flow policy $((L, \leqslant), U, O, \lambda)$, we can compute a derivation out-tree T and ϕ such that $\widehat{\mathsf{K}}$ (K) is minimised in its associated tree-based KAS in time $O(|A_{\min}| + |L|^2)$.*

Proof. By Corollary 5, we may restrict our attention to arcs in the Hasse diagram. Thus we can compute the minimum weight derivation out-tree in time $O(|A_{\min}|)$ and we can compute ϕ in time $O(|L|^2)$. Thus the total running time of the algorithm is $O(|A_{\min}| + |L|^2)$. □

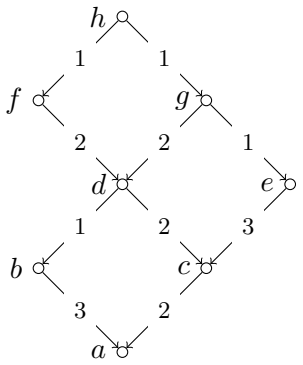
Remark 5. *In practice, we expect that $|U(x)| > 0$ for all $x \in L$, although our proofs do not make this assumption. If we do make this assumption, it is possible to strengthen the statement in Corollary 5 and assert that any minimum weight spanning out-tree* only *contains arcs from the Hasse diagram.*

Figure 4.5 illustrates the construction of the minimum weight spanning out-tree for the poset in Figure 3.1 (assuming there is a single user for each vertex). The weight on arc ec is 3, for example, because $\gamma(ec) = \{c, d, f\}$. (The effect of retaining arc ec would be that s_c would be required for each of c, d and f. Equivalently, $c \in \phi(d)$ and $c \in \phi(f)$ if we were to choose ec to belong to our derivation out-tree.) To construct a minimum weight spanning out-tree, we must select arcs ca and dc (and we select one or other of fd and gd). One possible scheme, when gd is retained rather than fd is illustrated in Figure 5.12c: the scheme requires a total of 11 intermediate secrets, being the sum of the weights on the retained arcs plus an extra one for the root vertex.

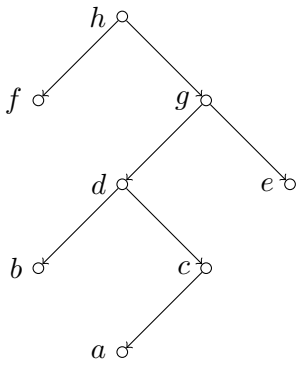
We now provide an example to illustrate our results. Let $[n] = \{1, 2, \ldots, n\}$ and let $[i, j] = \{i, i+1, \ldots, j-1, j\}$ for $i \leq j$. Then define the poset

$$\mathcal{I}(n) = \{[i, j] : 1 \leqslant i \leqslant j \leqslant n\},$$

where $[i, j] \leqslant [i', j']$ if and only if $i' \leqslant i$ and $j' \geqslant j$. The Hasse diagram for $\mathcal{I}(5)$ is illustrated in Figure 4.6a. The poset $\mathcal{I}(n)$ has attracted considerable interest because of its application to 'time-bound' access control (see [8, 31], for example). In particular, the numbers $1, \ldots, n$ represent time points or time intervals, and elements in $\mathcal{I}(n)$ represent contiguous intervals



(a) Weights on arcs.

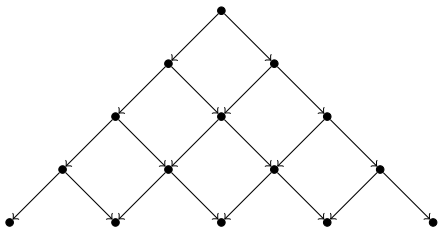


(b) Derivation out-tree.

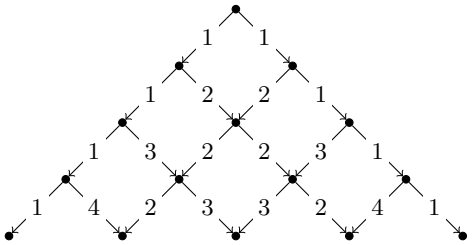
Figure 4.5: Minimum weight chain partition and derivation tree for Figure 3.1.

of time (either consecutive points or a sequence of consecutive intervals). A user u assigned the interval $[i, j]$ is authorised to access any object assigned an interval $[i', j'] \subseteq [i, j]$.

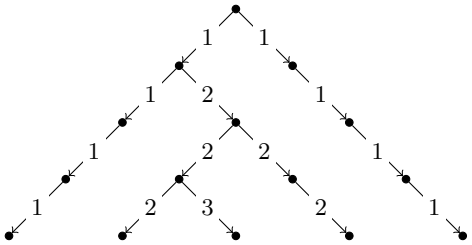
$\omega(yz)$, $y, z \in \mathcal{I}(5)$, $y \succcurlyeq z$, is shown in Figure 4.6b. A tree of minimum weight is shown in Figure 4.6c and the number of intermediate secrets required by each security label is shown in Figure 4.6d. We assume $|U(x)| = 1$ for all security labels $x \in \mathcal{I}(5)$.



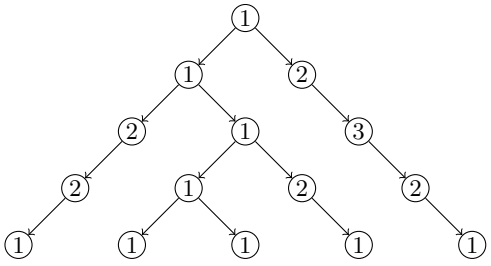
(a) Hasse diagram of a poset.



(b) $\omega(yz)$ for each arc yz.



(c) Tree partition of minimum weight.



(d) $|\sigma_x|$ for each $x \in L$.

Figure 4.6: A minimal tree partition of $(\mathcal{I}(5), \subseteq)$.

4.4 Conclusion

In this chapter, we have introduced a new form of key assignment scheme for the enforcement of information flow policies. Our scheme has the advantage that no public derivation

information is required for the derivation of decryption keys. Nevertheless, our scheme retains the strong security properties that have recently been established for chain-based schemes [47]. From a practical perspective, we provide an efficient algorithm for computing an optimal derivation out-tree, in the sense that it requires the smallest number of intermediate secrets for the entire user population. Furthermore, given an information flow policy $((L, \leqslant), U, O, \lambda)$, we require $\mathcal{O}(|A_{\min}| + |L|^2) \approx \mathcal{O}(L^2)$ time (see Corollary 6) to compute a derivation out-tree of $(L, \leqslant)$ *and* compute σ_l for all $l \in L$, in contrast to $\mathcal{O}(|L|^4\, w)$ time required to compute an optimal chain partition (see Theorem 1). Thus, there are particular practical advantages to using a tree-based approach.

Chapter 5

Binary Tree Key Assignment Scheme

Contents

In this chapter, we consider an alternative method of designing key assignment schemes by mapping read-only information flow policies to a full binary tree structure. 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. We consider how to optimise both the structure and mapping of the policy to this structure in order to reduce the average number of intermediate secrets required by each user. In the worst case, users may require more cryptographic material than in prior schemes; we mitigate this by designing heuristic optimisations of the mapping and show through experimental results that our scheme performs well compared to existing schemes.

This chapter is based on the following published work:

- *J. Alderman, N. Farley and J. Crampton, Tree-based Cryptographic Access Control, ESORICS, 2017.*

5.1 Introduction

Several prior works [3, 6, 8, 33, 34, 36], including Chapters 3 and 4, primarily focus on reducing the amount of key material required by each user and/or distributed to the entire user population. In this chapter, we focus instead on designing a KAS that minimises key derivation time.

Derivation in the KAS by Akl and Taylor [3] was based on expensive computations; derivation in other KASs is often heavily dependent on the graph chosen to represent the policy. For example, several KASs are instantiated on graphs which are subsets of the transitive closure of the poset, often simply the Hasse diagram [33, 34, 36, 83]. Many works [6, 8, 31, 82] reduce derivation costs by adding 'shortcut' arcs to the Hasse diagram of the policy poset but require a substantial amount of additional public (derivation) information e.g. $\mathcal{O}(n^2)$ where n is the number of labels in the policy (approximately one piece per arc in the graphical representation of the policy). Thus the amount of public information used to support key derivation may be large, particularly when labels are defined in terms of subsets of attributes [30, 64].

Recent works [32, 33, 34, 48] (see also Chapters 3 and 4) aim for space-efficient KASs by eliminating public derivation information via partitioning the (transitive closure of the) Hasse diagram into chains or trees. In such schemes keys and intermediate secrets can be derived iteratively using pseudorandom functions. However, as demonstrated in Chapters 3 and 4, users may require additional intermediate secrets and it is not possible to bound derivation costs beyond the trivial $\mathcal{O}(n)$ bound, i.e. the depth of the graphical representation of the poset. For example, the cost to derive any key in a chain-based KAS (Chapter 3) is bounded by the length of the longest chain in the enforced chain partition.

Contributions. In this chapter, we consider mapping the policy poset to a binary tree structure, not a subset of the transitive closure of the poset (as is the case with many prior works [6, 32, 33, 34, 36, 83]). Of course, there may be many such choices of binary

tree, and many ways to map the access policy to the structure. Ideally, one should choose such a structure and mapping 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 primary design criteria to be optimised (e.g. key derivation cost) and choose a structure (e.g. binary tree) that provides these properties;
2. choose a mapping from the policy poset to the structure (in our case, a binary tree) that optimises the performance of the remaining design criteria (since we cannot generally optimise for multiple criteria simultaneously, we choose one to focus on and then, given such choices, we optimise the remaining solution space with respect to the remaining criteria);
3. instantiate a key derivation mechanism over the structure to define the keys and intermediate secrets to be used in the KAS.

Prior KASs were restricted in the choice of enforcement structure (graph representing the policy poset) due to only considering trivial mappings to enforcement structures (i.e. vertices in the enforcement structure corresponded directly to labels in the policy poset). In contrast, we introduce additional flexibility by allowing one to optimise the choice of binary tree 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 (e.g. by considering alternative enforcement structures to binary trees).

In this chapter, we shall design a KAS which eliminates public information *and* in which derivation costs are logarithmically bounded; our KAS therefore bridges the gap between KASs that bound derivation costs [6, 8, 31, 82] and recent schemes, for example those discussed in Section 3 and 4 which eliminate public derivation information but which cannot bound derivation [32, 33, 34, 48]. To achieve this goal, we use a binary tree as our enforcement structure. This choice is simple, enables us to remove public information (since keys can be derived iteratively down paths in the tree, as in Chapter 4), introduces interesting optimisation 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 quickly using lightweight key derivation mechanisms. 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 optimised (through the choice of mapping from policy poset to enforcement structure) is the amount of cryptographic material required by users. As with [33, 34] and as demonstrated in Chapters 3 and 4, removing public derivation information results in users requiring additional intermediate secrets; in our case, the worst-case bound is $\lceil n/2 \rceil$ intermediate secrets, where n is the number of security labels defined in the policy to be enforced. We thus develop heuristic methods for finding a mapping which minimises $\widehat{\mathsf{K}}$ (the amount of cryptographic material required by the user population), and thus minimises the average number of intermediate secrets each users must store. Our experiments also demonstrate that our scheme works well in practice. Indeed, we show that this scheme compares favourably with other KASs that require no public information (e.g. the chain and tree-based KASs described in Chapters 3 and 4 respectively).

In Section 5.2, we introduce our KAS based on a binary tree, and show that it is strongly key indistinguishable in Section 5.3. In Section 5.4, we propose methods to optimise the choices of enforcement structure and mapping of security labels to vertices in the enforcement structure in order to minimise $\widehat{\mathsf{K}}$. In Section 5.5 we discuss some additional interesting policy features enabled by our scheme. In Section 5.6, we provide a theoretical and experimental evaluation of several KASs, including the KASs proposed in this thesis. We conclude the chapter in Section 5.7.

5.2 Our Construction

We begin by motivating our choice of enforcement structure according to the design goals of our example (to minimise the amount of 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.

5.2.1 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 representing the policy poset) has in-degree at most one, i.e. each intermediate secret is derived from *at most one* other intermediate secret [33, 34]. As described in Chapter 4, tree partitions generally require fewer intermediate secrets to be distributed to the user population than chain-based schemes (Chapter 3). For this reason, we will choose a tree structure.

We restrict our focus to *binary* trees, which are simple structures to consider 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 total number of intermediate secrets that must be issued to the user population can be reduced when multiple users are authorised for some set of access rights (security labels) and that these sets correspond to descendants of vertices 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 out-degree of vertices in the tree is low.

The maximum derivation cost for any key is bounded by the length of the maximal path in the enforcement structure. The minimum depth of a binary tree with n leaves is $\lceil \log n \rceil$.[1] Internal vertices with a single child only increase derivation paths[2] and so we restrict our focus to *full* binary trees (where all vertices 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 on the efficiency of the resulting KAS. In this section we assume that the specific tree and mapping are given and we show how to assign and derive intermediate secrets and keys (for an arbitrary policy). We consider methods to optimise these choices to enforce *specific* policies in Section 5.4.

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

[2] It will become clear in later sections that such vertices are unnecessary and only increase the time to derive keys.

5.2.2 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 full binary tree $T_n = (V, A)$, with n leaves and depth $\lceil \log n \rceil$, and a bijective mapping α 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 an intermediate secret to the root node of T_n from which an intermediate secret for each non-root vertex may be derived using standard key derivation methods (i.e. by iteratively applying a PRF). The binary string associated to the vertex dictates how the intermediate secret is derived.
3. For each security label $l \in L$, we define the key κ_l used to protect data objects in the KAS to be the intermediate secret assigned to the leaf labelled $\alpha(l)$. To minimise the material issued to users, we issue intermediate secrets associated to non-leaf vertices of T_n from which intermediate secrets for all descendant vertices can be derived (in particular users can derive all keys for which they are authorised).

Labelling the tree. We label the root vertex of T_n by the empty string ϵ and, for each vertex $x \in V$, label the left and right children of x (if they exist) by $x \parallel 0$ and $x \parallel 1$ respectively. Figure 5.1a gives an example labelling of a tree T_5. We may abuse notation by referring to a vertex of T_n and its associated binary string interchangeably. We denote the set of leaf vertices in T_n by $\overline{V}$. Note that a vertex $x \in V$ is an *ancestor* of a vertex $y \in V$ if and only if the binary string associated to x is a prefix of the string associated to y.

Deriving secrets and keys. We now assign an intermediate secret to each vertex. Let ρ be a security parameter and let $\mathcal{F} : \{0,1\}^\rho \times \{0,1\}^\star \to \{0,1\}^\rho$ be a pseudorandom function (PRF) which takes as input a key κ and a string x and outputs a pseudorandom string of the same length as the key.

The intermediate secret s_ϵ associated to the root vertex $\epsilon \in V$ is chosen uniformly at

random: $s_\epsilon \xleftarrow{\$} \{0,1\}^\rho$. For each non-root vertex $y = x \parallel b$ in V, where $x \in V$ and $b \in \{0,1\}$, we compute the secret $s_y = \mathcal{F}_{s_x}(b)$. If x is a prefix of y, then s_y may be derived from s_x by iteratively applying $\mathcal{F}$ on each remaining bit of y in turn. This is shown in Figure 5.1b and in GetSec in Figure 5.1c. For appropriate choices of $\mathcal{F}$, it is computationally infeasible to compute s_x from s_y.

Assigning keys. Recall that α is a bijective mapping associating each security label $l \in L$ to a unique leaf vertex $\alpha(l)$ 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 data object $o \in O$ is associated with a security label $\lambda(o) \in L$. Hence, $\lambda(o)$ is associated with a leaf vertex $\alpha(\lambda(o)) \in T_n$. We may refer to the intermediate secrets associated to leaf vertices 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 authorised for the set of 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 vertices of T_n to represent multiple elements of $\downarrow\lambda(u)$. If $\alpha(\downarrow\lambda(u))$ contains *all* descendant leaf vertices of a vertex $x \in V$, we may instead issue the single intermediate secret s_x, from which the keys for all descendant leaf vertices can then be efficiently derived. More formally:

Definition 13. *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 a prefix of x in $\lceil X \rceil$;*
2. *for every $y \in \lceil X \rceil$, every $z \in \overline{V}$ that has y as a prefix belongs to X.*

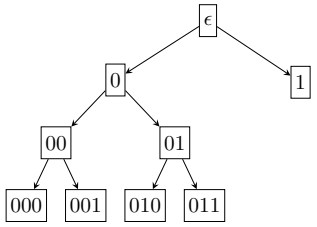
Then, a user issued the user secret $\sigma_{\lambda(u)}$ containing the set of intermediate secrets $\{(x, s_x) : x \in \lceil \alpha(\downarrow\lambda(u)) \rceil\}$ may derive $\kappa_l = s_{\alpha(l)}$, where $l \in L$ if and only if $l \leqslant \lambda(u)$. Condition 1 ensures that a user can derive all keys for which they are authorised (*correctness*), whilst Condition 2 ensures that they cannot derive any other keys (*security*). Since T_n is a full tree (every vertex 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 5.1a and suppose $\alpha(\downarrow l) = \{010, 011, 1\}$ for some label $l \in L$. Then, $\lceil \alpha(\downarrow l) \rceil = \{01, 1\}$,

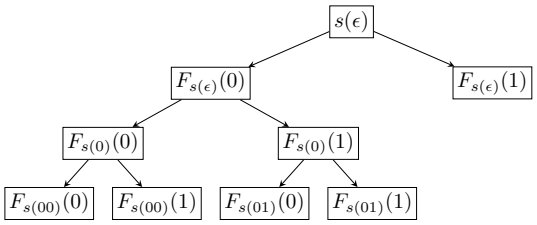
and σ_l contains $s_{01} = \mathcal{F}_{\mathcal{F}_{s_\epsilon}(0)}(1) = \mathcal{F}_{s_0}(1)$ and $s_1 = \mathcal{F}_{s_\epsilon}(1)$.

Let us define the *strict prefix* of bit string $b_0 b_1 \dots b_i$ to be $b_0 b_1 \dots b_{i-1}$. A simple method to compute $\lceil X \rceil$ for $X \subseteq \overline{V}$ is to observe that if two bit strings in X share a strict prefix, both may be replaced by the strict prefix, and the intermediate secrets 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 setup authority need not store the enforcement structure T_n.

5.2.3 Summary



(a) T_5.



(b) Secret generation.

$\mathsf{SetUp}(1^\rho, (L, \leqslant))$

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

$\mathsf{Derive}(-, -, \alpha(y), \sigma_x, -)$

foreach $(l, s_l) \in \sigma_x$:
  if l is a prefix of $\alpha(y)$
    return $\mathsf{GetSec}(\alpha(y), l, s_l)$
return $\perp$

$\mathsf{GetSec}(a, b, s_b)$

if b is not a prefix of a
  return $\perp$
$z \leftarrow b$
for $i = len(b) \dots len(a) - 1$:
  $s_{z \| a_i} = F_{s_z}(a_i)$
  $z \leftarrow z \| a_i$
return s_a

(c) Our KAS construction.

Figure 5.1: Binary tree KAS construction with an example tree T_5 and an illustration of intermediate secret generation. The inputs to the supporting algorithm GetSec in the KAS are two bit strings $a = a_0 \dots a_m, b = b_0 \dots b_n$, where $m, n \in \mathbb{N}$, and an intermediate secret s_b.

Our complete KAS construction is given in Figure 5.1c. Note that the existing definition of the KAS Setup algorithm does not allow one to construct an optimal α mapping that considers other aspects of the policy other than the policy poset itself. In Figure 5.1c, we comply to the current KAS definition, although one can simply tweak the Setup algorithm

to take as input the entire policy instead of just the policy poset. In Section 5.4, we propose a heuristic to find a suitable α mapping that can be constructed using just the policy poset (as well as the security parameter 1^ρ), and also give an alternative heuristic which performs better in experimental results but which requires additional knowledge about the policy (i.e. the set of users U and their security label assignments).

It is easy to establish the following properties of our KAS:

- no user requires more than $\lceil n/2 \rceil$ intermediate secrets;
- no user requires more than $\lceil \log n \rceil$ steps to derive a decryption key; and
- no additional information is required to perform key derivation.

In contrast, for an iterative KAS with public derivation information [36]:

- users require a single intermediate secret;
- derivation may take up to n steps;
- up to $\mathcal{O}(n^2)$ items of public (derivation) 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 intermediate secrets. A more detailed comparison with related work is given in Section 5.6.

Derivation in our construction requires knowledge of a binary label $\alpha(y)$ for $y \in L$; hence one may argue that the α 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 labelled by their security label to identify the objects to be retrieved from the file-system and the decryption keys to use, whilst intermediate secrets must be labelled such that they can be used to derive appropriate decryption keys.[3]

In our scheme, $\sigma_{\lambda(u)}$ contains the appropriate binary labels and we assume that each object $o \in O$ is labelled by $\alpha(\lambda(o))$ instead of $\lambda(o)$. (In fact, $\alpha(\lambda(o))$ is a compact way

[3]It is unfortunate that existing KAS definitions do not permit consideration of such implementation details.

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 α need not be public. Derive requires only the binary string $\alpha(y)$ of the target label y and a suitable user secret σ_x (where $y \leqslant x$ in $(L, \leqslant)$, the policy poset); we omit other unrequired inputs.

To our knowledge, all prior KASs (including those without public derivation information) require that users store (or have public access to) the enforcement structure for use during Derive. In schemes that use public information, this is to identify the information needed to derive the next intermediate secret in the derivation 'path' in the associated enforcement structure. In schemes based on tree or chain partitions [32, 33, 34, 48], the algorithm must know which intermediate 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. It is easy to see that our KAS is *correct* due to Condition 1 of Definition 13 and the iterative nature of the key generation. We may compute s_y from any intermediate secret s_x, where x is a prefix of y, and Condition 1 of Definition 13 ensures that, for all labels $l \in \downarrow\lambda(u)$, there exists a prefix of $\alpha(l)$ in $\lceil \alpha(\downarrow\lambda(u)) \rceil$.

5.3 Strong Key Indistinguishability of our KAS

$\mathbf{Exp}^{\mathsf{SKI-b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$:

$(\{\sigma_l, \kappa_l\}_{l \in L}, Pub) \xleftarrow{\$} \mathsf{SetUp}(1^\rho, (L, \leqslant))$

$\kappa_0^\star \xleftarrow{\$} \mathcal{K}, \kappa_1^\star \leftarrow \kappa_x$

$Corrupt_x \xleftarrow{\$} \{(l, \sigma_l) : l \in L, x \not\leqslant l\}$

$Keys_x \xleftarrow{\$} \{(l, \kappa_l) : l \in L \setminus \{x\}\}$

$b' \xleftarrow{\$} \mathcal{A}(1^\rho, (L, \leqslant), x, \kappa_b^\star, Corrupt_x, Keys_x, Pub)$

return $b' = b$

Figure 5.2: Static strong key indistinguishability of a KAS.

Our scheme (as shown in Figure 5.1c) meets the strongest security property currently defined for KASs:

Theorem 7. *Let $\mathcal{F} : \{0,1\}^\rho \times \{0,1\}^\star \to \{0,1\}^\rho$ be a secure pseudorandom function with security parameter $\rho \in \mathbb{N}$ and let $(L, \leqslant)$ be a poset. Then the KAS $\mathcal{KAS}$ defined in Figure 5.1c is* strongly key indistinguishable.

Proof. The proof is based on that of Theorem 4. Let $n = |L|$ and $m = \lceil \log n \rceil$. For a bit string $B = b_0 b_1 \dots b_t$, define $B^{(i)} = b_0 b_1 \dots b_{i-1}$ to be the prefix of length i. We consider a sequence of m hybrid games that interpolate between $\mathbf{Exp}^{\mathsf{SKI}-0}_{\mathcal{KAS},\mathcal{A}}$ and $\mathbf{Exp}^{\mathsf{SKI}-1}_{\mathcal{KAS},\mathcal{A}}$ (see Figure 5.2). Let $|\alpha(x)|$ be the length of the binary string $\alpha(x)$. For $i = 0, \dots, m$, if $i \leqslant |\alpha(x)|$, then game i replaces one PRF instance in game $i-1$ by a truly random function, otherwise game i is the same as game $i-1$. We show that an adversary cannot distinguish two consecutive games with advantage greater than that of distinguishing the PRF.

Set $G^b_0 = \mathbf{Exp}^{\mathsf{SKI}-\mathsf{b}}_{\mathcal{KAS},\mathcal{A}}(1^\rho, (L, \leqslant), x)$ for $b \in \{0,1\}$, as in Figure 5.2. For $i = 1$ to $|\alpha(x)|$, define game G^b_i to be the same as game G^b_{i-1} with the exception that the PRF $\mathcal{F}_{s_{\alpha(x)^{(i-1)}}}(\cdot)$ is replaced by a function r chosen uniformly at random from $\langle \{0,1\}^\star \to \{0,1\}^\rho \rangle$, the set of all functions with with domain $\{0,1\}^\star$ and range $\{0,1\}^\rho$. If $|\alpha(x)| < m$, define $G^b_j = G^b_{|\alpha(x)|}$ for $j \in \{|\alpha(x)|+1, \dots, m\}$.

Note that, by definition, $Corrupt_x$ does not contain any σ_v such that $v \geqslant x$. Therefore, for all bit strings $B \in \bigcup_{\sigma_v \in Corrupt_x} \lceil \downarrow \alpha(v) \rceil$, B is *not* a prefix of $\alpha(x)$. In particular, $Corrupt_x$ will not include the PRF key s_ϵ.

Since PRFs are swapped to random functions iteratively we may observe that, in Game G^b_i, the value $s_{\alpha(x)^{(i-1)}}$ used as the PRF key to compute $s_{\alpha(x)^{(i)}}$ has already been replaced by the output of a truly random function.

Let S^b_i denote $\Pr[G^b_i \to 1]$ for all b, i. For compactness, we will denote the advantage $\mathrm{Adv}^{\mathsf{ind}-\mathsf{prf}}_{\mathcal{F},\mathcal{D}^b_i}(1^\rho)$ of a distinguisher $\mathcal{D}^b_i$ in distinguishing a PRF $\mathcal{F}$ from a random function (see Definition 3) as $\mathrm{Adv}^{\mathcal{F}}_{\mathcal{D}^b_i}$. We now argue that if an efficient PRF distinguisher $\mathcal{D}^b_i$ exists that can distinguish between the two games G^b_i and G^b_{i-1} then an adversary, using the distinguisher as a subroutine, can also distinguish the two cases (by taking the distinguisher's guess of b as his own). By the security of our PRF, no such distinguisher exists. Thus, we

have that, whenever $i \leqslant |\alpha(x)|$:

$$|S_i^b - S_{i-1}^b| \leqslant Adv_{D_i^b}^{\mathcal{F}}.$$

In addition, whenever $i > |\alpha(x)|$, we have $G_i^b = G_{i-1}^b$, and hence $|S_i^b - S_{i-1}^b| = 0$. By the triangle inequality,

$$|S_0^b - S_m^b| \leqslant \sum_{i=1}^{m} |S_{i-1}^b - S_i^b| \leqslant \sum_{i=1}^{m} Adv_{D_i^b}^{\mathcal{F}}.$$

Note that in G_m^b, $\mathcal{F}_{s_{\alpha(x)(m-1)}}(\cdot)$ is replaced by a random function, hence for both choices of b (even in the 'real' case), κ_x is a truly random value; thus, $|S_m^1 - S_m^0| = 0$. Again by the triangle inequality,

$$\begin{aligned}\mathrm{Adv}_{\mathcal{KAS},\mathcal{A}}^{\mathsf{SKI}-b}(1^\rho, (L, \leqslant), x) = |S_0^1 - S_0^0| &\leqslant |S_0^1 - S_m^1| + |S_m^1 - S_m^0| + |S_m^0 - S_0^0| \\ &\leqslant \sum_{i=1}^{\lceil \log n \rceil} Adv_{D_i^1}^{\mathcal{F}} + 0 + \sum_{i=1}^{\lceil \log n \rceil} Adv_{D_i^0}^{\mathcal{F}}.\end{aligned}$$

Hence the advantage of $\mathcal{A}$ against our KAS is bounded by a negligible probability. □

As mentioned previously, one may want to to modify our KAS construction defined in Figure 5.1c such that the Setup algorithm takes as input the entire policy $P = ((L, \leqslant), U, O, \lambda)$ instead of just $(L, \leqslant)$. By making this alteration, one could argue that the α mapping in Setup can be further optimised for the given policy.

Then, to prove that the modified KAS is SKI secure, we may play the SKI game in Figure 5.2 as before, with the exception that the Setup algorithm takes as input the entire policy $P = ((L, \leqslant), U, O, \lambda)$ instead of just $(L, \leqslant)$. Then, by using this modified experiment in the proof of Theorem 7 instead of the original experiment (shown in Figure 5.2), the result holds.

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 security label to be associated with an intermediate secret *and* a key. In our case, intermediate secrets are associated with interior vertices of the tree (which are not associated to a security label), while keys are just intermediate secrets associated with leaf vertices; the

values issued to users (i.e. user secrets $\sigma_{\lambda(u)}$) may, and do, contain keys themselves.

Related Work. Our construction is similar to the puncturable PRF construction proposed in [23], which makes use of the Goldreich, Goldwasser and Micali (GGM) construction of a PRF from pseudorandom number generators [56]. Hohenberger *et al.* [63] note that one could alternatively describe a puncturable PRF based on the GGM construction as a prefix-fixing *constrained* PRF. (Constrained PRFs are also referred to as *functional* [24] or *delegatable* PRFs [67] in the literature.) In Section 5.5, we take advantage of the inherent puncturing mechanism of our scheme 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.* [11], 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.* [21] also considered methods to derive keys using tree structures in the context of access control matrices, showed that finding optimal trees to minimise the size of 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 5.4.

5.4 Optimising 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 (i.e. it requires no public information and the number of key derivations is logarithmically bounded), but that users may require many intermediate secrets in the worst-case. We therefore aim to design methods that, given a policy poset, mitigate this concern and optimise the performance of the resulting KAS by trying to reduce the average number of intermediate secrets any user requires. (Prior schemes are limited in this regard as they only consider enforcement structures based directly on the poset, e.g. Hasse diagrams.)

Recall that each user $u \in U$ is issued a set of intermediate secrets $\sigma_{\lambda(u)}$ associated to the minimal cover $\lceil \alpha(\downarrow\lambda(u)) \rceil$ of their authorised set. Thus, whenever $\alpha(\downarrow\lambda(u))$ contains *both*

children of a vertex in T_n, the size of $\sigma_{\lambda(u)}$ is reduced by one. To minimise the average size of $\sigma_{\lambda(u)}$ over all users $u \in U$, we therefore aim to define α such that the authorised sets $\alpha(\downarrow\lambda(u))$ contain as many such pairs of child vertices 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 5.3 illustrates the effect of choosing two different α mappings when $n = 5$.

We conjecture that finding a mapping that minimises the number of intermediate secrets required by each user 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 intermediate secrets.

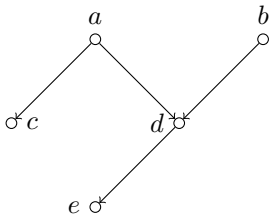
Our goal in this section, therefore, is to introduce heuristics to find 'good' α mappings that minimise the average number of intermediate secrets required by each user. We first describe our best performing heuristic, based on finding maximal matchings between sets of labels with respect to suitable weightings.

We then discuss a considerably cheaper heuristic which, in our experiments, provides reasonable performance.

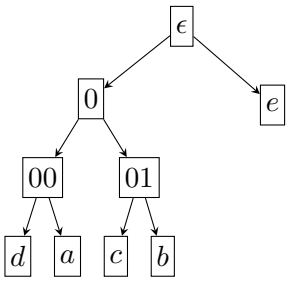
5.4.1 The FindTree Heuristic

Recall that the size of a binary label represents the depth of the associated vertex in T_n; thus we may fully describe the structure of T_n and the assignment of labels to leaves via an α 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 D. For each $l \in L$, define $P = [l]$ to be a partition (of depth $\mathsf{D}(P) = 0$). For two partitions P and Q, define $[P, Q]$ to 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 5.3c is represented by:

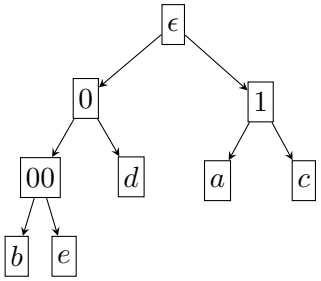
$$[[[[b], [e]], [d]], [[a], [c]]].$$



(a) Poset.



(b) T_5 generated by α_1.



(c) T_5 generated by α_2.

l	$\downarrow l$	$\alpha_1(l)$	$\alpha_2(l)$	$\lceil\alpha_1(\downarrow l)\rceil$	$\lceil\alpha_2(\downarrow l)\rceil$
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	$\{1\}$	$\{001\}$

Figure 5.3: An example showing the effects of two different choices of α mappings. Observe that the average size of $\lceil\alpha_2(\downarrow l)\rceil$ is smaller than that of $\lceil\alpha_1(\downarrow l)\rceil$.

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

Recall that a *matching* of an undirected graph $G = (V, E)$ is a set $M \subseteq E$ of pairwise non-adjacent undirected edges, i.e. no two edges in M share a common vertex. If G had 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. 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 authorised for *all* labels in the connected partitions. We find a *maximum weight matching* on this graph which selects edges to maximise the associated weights; matched vertices represent partitions that should be grouped as a sub-tree in T_n.

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

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 authorised 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 $[V[0], V[1]]$.

Figure 5.4: FindTree heuristic.

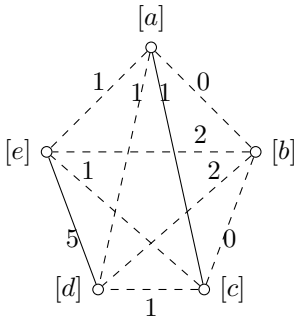
We iterate this process to form larger groups, beginning with pairs, since smaller sets of labels are most likely to occur in multiple authorisation sets and hence benefit the most users. Ultimately we create a sequence of nested partitions (of differing sizes) describing which labels should be grouped, 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 5.4. Figure 5.5 illustrates the heuristic on the poset in Figure 5.3a; the selected maximum weight matchings are illustrated by solid edges. The average number of intermediate secrets required (when $|U(l)| = 1$ for all $l \in L$) is $\frac{6}{5}$ using the mapping found via FindTree, compared to $\frac{8}{5}$ when using the α_2 mapping from Figure 5.3c. (Note that we compute the average by first summing the number of intermediates secrets required by each user, and then dividing this total by the number of users.)

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

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

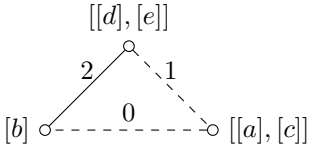
(a) Initial vertices and user assignments.



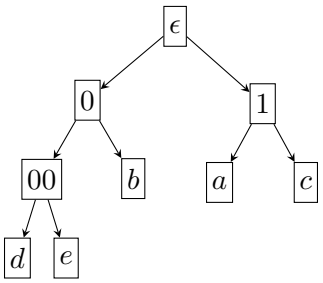
(b) First matching.

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 α and minimal covers.

Figure 5.5: Example application of the FindTree heuristic on the poset in Figure 5.3a with user assignments shown in Table 5.5a.

number of users authorised for all labels in the connected partitions. Recall the *order filter* $\uparrow l = \{x \in L : x \geqslant l\}$ describes the labels authorised 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. 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 authorised 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). Note that edges must be non-adjacent and so each choice restricts the choices for other vertices; the matching may not include the edge with the greatest weight if a combination of smaller weights is greater overall.

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 [50]. Since we iterate $\mathcal{O}(n)$ times, our heuristic requires $\mathcal{O}(n^4)$ time.

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 lowest, and the leaves are as far left as possible.)

We now 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 $\mathcal{O}(n^3 \log n)$. As mentioned, we find that the tree structure chosen by FindTree on randomly generated posets of varying sizes is left-balanced (or a tree structure in which left and right branches of vertices can be swapped to form a left-balanced tree) approximately half the time. Therefore, our first step in reducing the computational cost is to fix the tree structure and merely find the best mapping of labels to the leaves of a left-balanced tree. To do so, the FindTree algorithm is amended such that unmatched vertices at each iteration are removed from subsequent iterations; each removed vertex

represents a sub-tree that is added to the right of the tree. The final partitions are sorted into decreasing order of size (according to the number of labels in each partition), and leaves of the tree are labelled from left to right. By removing unmatched vertices rather than performing additional iterations until all such vertices are (eventually) matched, the run-time is reduced to $\mathcal{O}(n^3 \log n)$. Figure 5.6 shows that fixing the tree structure does not particularly degrade the performance of the heuristic in our experiments

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.

5.4.2 The Order Filter Sort Heuristic

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 vertices from left to right. Intuitively one hopes that by pairing labels with large order filters, the order filters are likely to intersect. Users authorised for a label within the intersection require at least one fewer intermediate secret. This heuristic requires $\mathcal{O}(n \log n)$ time, and we shall see in Section 5.6 that it performs remarkably well in practice. Unlike FindTree, this heuristic does not consider the number of users assigned to labels; in practice, however, we may expect more users to be assigned 'low' labels (with large order filters), e.g. there are likely to be more employees than managers. Thus, whilst FindTree may be better in general, many realistic scenarios may favour this second

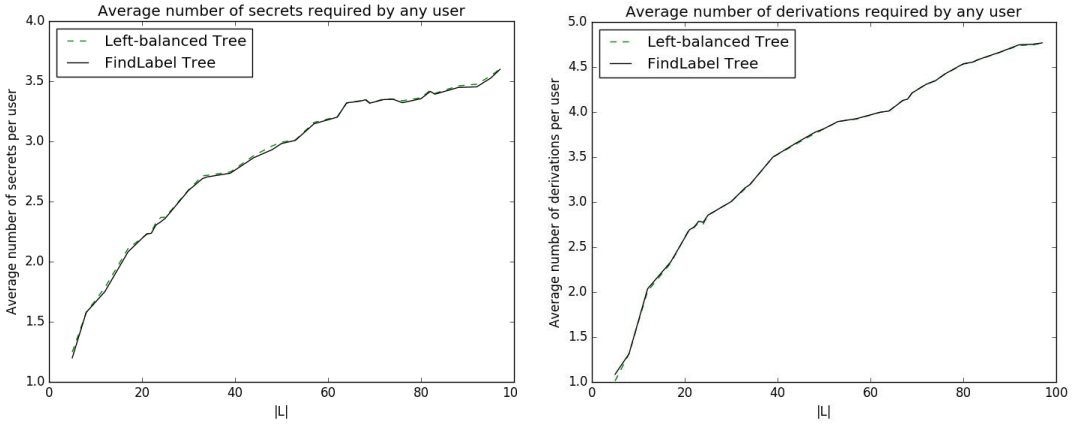


Figure 5.6: Performance of FindTree with a fixed left-balanced tree.

heuristic.

5.5 Flexible Access Management

In this section, we summarise some additional features enabled by our KAS.

Prior KASs [6, 36, 48, 83] require *all* keys, secrets and derivation information to be defined and assigned during Setup 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 [30]). In contrast, using our KAS, one can define T_n for n primitive labels and define a *single* intermediate secret (for the root node of T_n) during Setup. Instead of defining additional labels for each potential combination of access rights, informally, one could define a new algorithm Assign that could dynamically issue intermediate secrets corresponding to the minimal cover of a required set of primitives as required — one can use such a mechanism to dynamically form new 'labels' that cover the required access rights as users join the system. This mechanism is similar to the GGM puncturable PRF [56] and thus can be viewed as utilising the puncturing mechanism to define access rights. A puncturable PRF issues keys restricting the pseudorandom outputs that may be computed. This is precisely the goal of a KI-secure KAS, albeit with additional contextual information to relate keys to a policy. This puncturing mechanism can enable useful features such as:

Limited Depth Inheritance is an important component of some hierarchical access policies to prevent senior users aggregating excessive access rights [6, 28, 78, 80]. For example, one may want to restrict certain users from inheriting all of the access rights associated to their label. Considering Figure 5.3a for example, one may want to allow a user assigned security label b or d to inherit the access rights of security label e, but prevent users assigned security label a from inheriting the access rights for security label e. Encoding such restrictions directly into the policy 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 increase the number of intermediate secrets users must

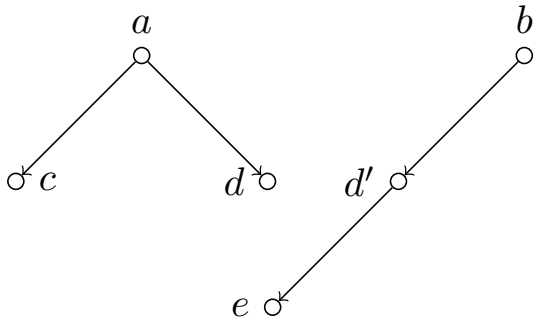


Figure 5.7: Modified poset with limited depth inheritance.

hold [33, 34]). For example, one may construct a new poset, such as the one shown in Figure 5.7, to satisfy this constraint. In this example, we create another copy of security label d, d' in order to satisfy the limited depth constraint.

To our knowledge, the only KAS that directly allows limited depth inheritance [6] requires public derivation 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 authorised 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 intermediate secrets of labels $l' > l$. In our KAS, on the other hand, intermediate secrets correspond to interior vertices 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 intermediate secrets. Then, considering again Figure 5.3a, we may give a user assigned security label a who is not authorised for security label e the intermediate secrets corresponding to labels in the minimal cover $\lceil \alpha(\downarrow a \setminus e) \rceil = \lceil \alpha(a, c, d) \rceil$. Using the mapping α_1 shown in Figure 5.3b, this would correspond to the set of intermediate secrets $\{00, 010\}$.

Separation of Duty policies [80] compartmentalise objects and users to avoid conflicts of interests. For example, it is particularly useful in the context of role-based access control, and enables one to enforce rules such as 'users authorised for role r_1 are not authorised for role r_2' and so on. 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 the set of intermediate

secrets corresponding to the set of labels $\lceil \alpha(\downarrow u \setminus X) \rceil = \lceil \{\alpha(l) : l \in \downarrow u \setminus X\} \rceil$.

Interval-based Policies such as temporal or geo-spatial policies [7, 31] can be handled in the same way. Consider a temporal policy where L is a set of time periods $[0, n]$ and users are authorised for time intervals $[a, b]$ for $0 \leqslant a, b \leqslant n$. 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.

5.6 Scheme Comparison

We now compare the binary tree scheme to prior KASs with respect to the following parameters: $\mathsf{k}_{\max}$ is an upper bound on the number of keys/intermediate secrets any user must be issued, $\mathbf{p}$ is the amount of public derivation information, and $\mathbf{d}$ is an upper bound on the number of key derivation steps required by any user. The discussion is summarised in Table 5.1. Note that this discussion extends the scheme comparison made in Section 2.5.4.

As mentioned previously, many schemes issue users a single key ($\mathsf{k}_{\max} = 1$) and enable iterative derivation along paths in the enforcement structure using public information. In several schemes [36, 83], the enforcement structure is simply the Hasse diagram of the information flow policy poset $(L, \leqslant)$, 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 arc 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 $\mathsf{k}_{\max} = \mathcal{O}(n)$, $\mathbf{p} = 0$ and $\mathbf{d} = 0$. Recent schemes remove public information by forming a sub-graph of the (transitive closure of the) Hasse diagram which is either a tree [34] or a chain partition [33, 32, 48]. 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 intermediate secrets/keys: for the chain-based KAS described in Chapter 3, $\mathsf{k}_{\max} = w$ where w is the width of $(L, \leqslant)$; for the tree-based KAS described in Chapter 4, $\mathsf{k}_{\max} = \ell$ keys, where $\ell \geqslant w$ is the number of leaves in the derivation out-tree; for the binary tree scheme: $\mathsf{k}_{\max} = \lceil n/2 \rceil$; $\mathbf{p} = 0$; and $\mathbf{d} = \mathcal{O}(\lceil \log n \rceil)$.

Scheme	$\mathsf{k_{max}}$	p	d
Trivial [36]	n	0	0
Iterative [6, 36]	1	$\lvert A_{\min}\rvert$	n
Direct [6, 36]	1	$\lvert A_{\max}\rvert$	1
Tree [34] (Chapter 4)	ℓ	0	n
Chain [33] (Chapter 3)	w	0	n
Binary Tree [5] (Chapter 5)	$\lceil \frac{n}{2} \rceil$	0	$\lceil \log(n) \rceil$

Table 5.1: Comparison of different KASs. $|A_{\min}|$ and $|A_{\max}|$ represent the number of arcs in the Hasse diagram $H(L, \leqslant)$ and its transitive closure, respectively.

Whilst the comparison appears promising, we remain concerned by the worst-case upper bound of $\lceil \frac{n}{2} \rceil$ intermediate secrets for users in the binary tree scheme. However, we now present an experimental evaluation showing how these KASs perform *in practice* in the worst- and average-cases as $|L|$ increases. For each value of $|L|$, we average the results on 30 posets generated as follows: for each vertex x we randomly generate a 'connection probability' p uniformly at random; for each other vertex 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 our comparison, we aim to evaluate the KASs on a variety of posets and have not aimed towards any particular policy. Future work should 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, thus it is interesting to see how various KASs compare in practice.

We experimented with generating random posets in different ways: for example, we considered generating a probability p for a graph, such that each possible arc in the graph was added with probability p. However, we found that this often resulted in posets which were not (well) connected. We expect that in practice, information flow policy posets will be well connected, and thus such posets did not seem to best represent what we considered to be real-life policies. Again, future work should consider how to best generate random posets for such an experimental comparison.

We compare an *iterative* scheme that uses public derivation information (the extended scheme by Atallah *et al.* [6] instantiated on the Hasse diagram of the poset), the *chain*-[33] and *tree*-based schemes [34] (which do not require public derivation information) described in Chapters 3 and 4 respectively, and the binary tree KAS (this Chapter) using both the FindTree (see Figure 5.5b) and the order filter sort heuristics. Figures 5.8 and

5.10 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 [6] also requires a number of decryptions which are not counted). Figures 5.9 and 5.11 show the average and maximum number of intermediate secrets (or keys) required by any user in each scheme. The iterative scheme is omitted for clarity, since each user requires one intermediate secret.

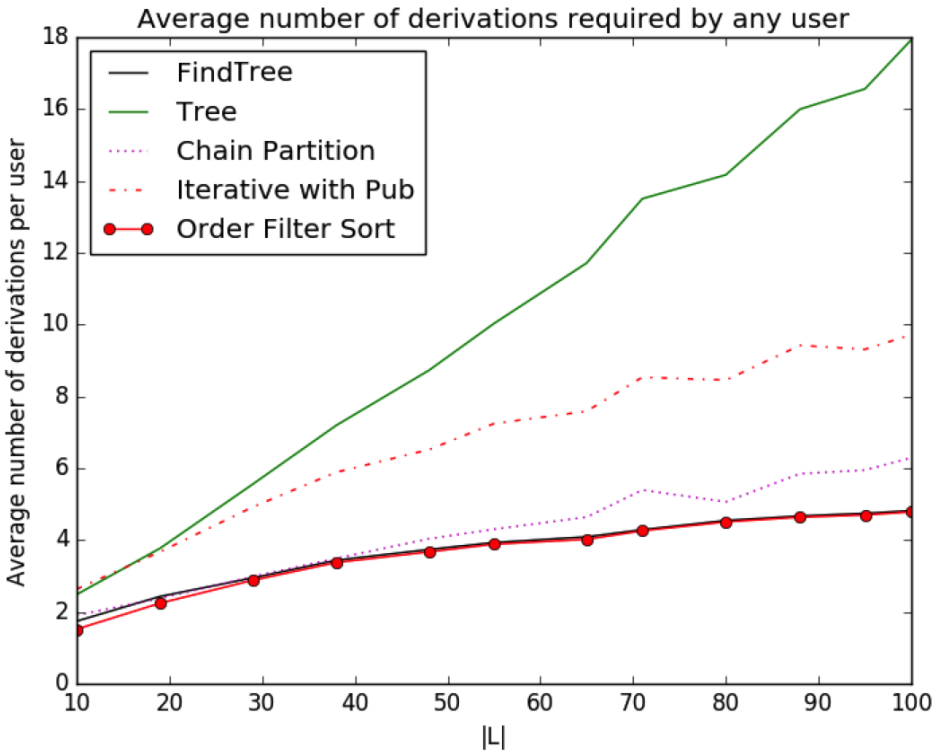


Figure 5.8: Average number of key derivations per user.

Derivation Costs. Table 5.1 and Figures 5.8 and 5.10 show that the binary tree scheme out-performs chain-based, tree-based and iterative [6, 36] schemes in terms of derivation costs. Indeed, the primary design goal of the binary tree scheme was to bound derivation costs whilst eliminating public information, and it can be seen that this is achieved. In particular, its logarithmic growth contrasts with the linear cost of tree-based schemes which, particularly in the worst-case, can become rather high. Furthermore, recall that the storage costs are further reduced in the binary tree scheme compared to other KASs since users need not store the enforcement structure. Of course, Table 5.1 shows that the trivial [36] and direct [6, 36] schemes require the smallest amount of key derivation, however, the trade-off is that the trivial scheme requires users to store all their necessary keys, whilst the direct scheme may require a significant amount of public information.

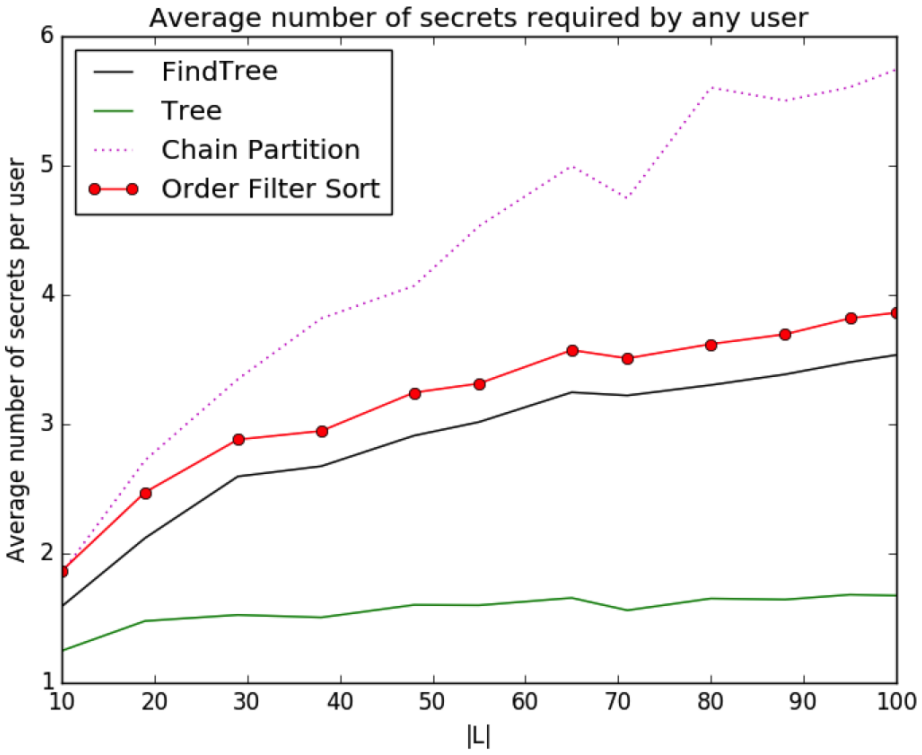


Figure 5.9: Average number of intermediate secrets per user.

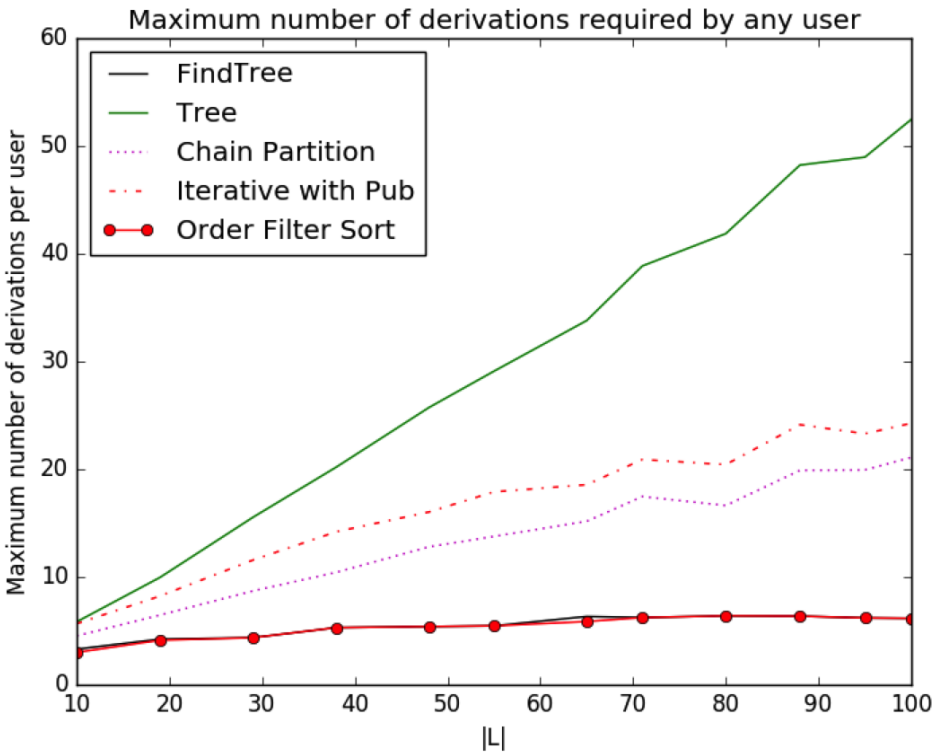


Figure 5.10: Maximum number of key derivations required by any user.

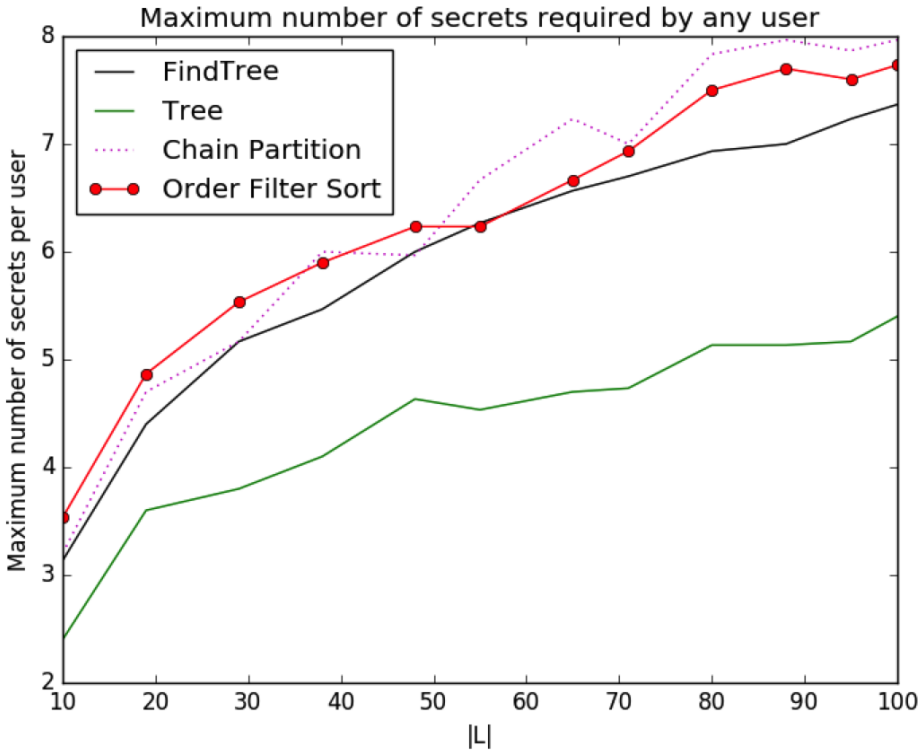


Figure 5.11: Maximum number of intermediate secrets required by any user.

Depending on the requirements of the scheme, if derivation and public information are important parameters to minimise, then one may consider using the binary tree KAS described in this chapter over the tree-based KAS described in Chapter 4.

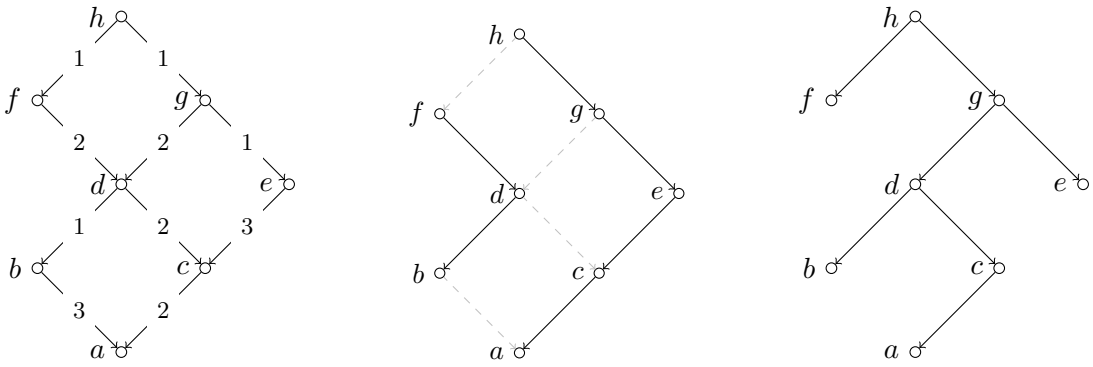
Secret information distributed to user population. Figures 5.9 and 5.11 suggest that the tree-based KAS performs better than the alternative schemes considered (excluding the iterative scheme) in terms of minimising the average and maximum number of intermediate secrets required by each user. Furthermore, it is worth noting that the tree-based construction proposed in Chapter 4 will almost always require fewer intermediate secrets than a scheme based on chain partitions. This follows by noting that any vertex x, such that $x > y$, $x > z$ and $\{y, z\}$ is an antichain, necessarily requires (at least) two intermediate secrets in a chain partition scheme, but this is not necessarily true of the tree-based scheme (since the derivation tree may include many antichains). For example, consider the chain partition in Figure 5.12b and the derivation out-tree in Figure 5.12c. The former would require 13 intermediate secrets, while the latter only 11. Thus, if minimising the amount of secret material distributed to the user population is a priority, then the tree-based KAS described in Chapter 4 should be considered.

Figure 5.9 also suggests that the binary tree scheme performed better than the chain-based scheme in minimising the total number of intermediate secrets required on average by each user. Importantly, in these experiments, the 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 the heuristics proposed 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 $\mathcal{O}(n^4)$). As mentioned previously, we expect that this is because labels with large order filters are likely to have many labels in their intersection, i.e. given two labels x and y with large order filters, we expect that $|{\uparrow}x \cap {\uparrow}y|$ is large.

However, if client storage is limited, one may still prefer the chain-based scheme described in Chapter 3 since it ensures that no user requires more than w intermediate secrets, where w is the width of the policy poset. In the tree-based scheme, however, users may be required to store at most l intermediate secrets, where l is the number of leaves in the derivation out-tree and $l \geqslant w$. Although we can minimise the number of leaves in a minimum weight spanning out-tree (see Corollary 3), it is not always possible to construct a tree which both minimises the total number of distributed intermediate secrets and has precisely $l = w$ leaves. Thus, if it important to ensure that no user has to store more than w intermediate secrets, one may prefer to use chain-based schemes instead of tree-based schemes.

Of course, the direct and iterative schemes perform the best, in regards to minimising secret material required by users, since each user only requires one piece of secret information. However, these schemes may require a significant amount of public information.

Public information. The trivial scheme [36] does not require any public information, since every user is given all their necessary keys. Similarly, the binary tree KAS eliminates public information through the use of binary labels for keys and data objects. The chain and tree-based KASs require only the respective tree and chain-based partitions to be public. Such partitions can be represented as an adjacency matrix of size $\mathcal{O}(|L|^2)$. The iterative and direct schemes [6, 36] require both the graph representing the policy and a piece of public information per arc in the graph representation of the poset used to be



(a) Weights on arcs. (b) A chain partition. (c) Derivation out-tree.

Figure 5.12: Minimum weight chain partition and derivation tree for Figure 3.1.

public. Thus, the iterative and direct schemes require approximately $\mathcal{O}(|L|^2)$ pieces of public information, where $|L|$ is the number of labels in the information flow policy.

Thus, if minimising public information is a priority, then the trivial and binary tree KASs may be preferred. Nevertheless, the chain and tree-based KASs significantly reduce public information compared to the iterative and direct schemes, since they do not require public derivation information. Furthermore, if the chain/tree partition representation is small, this could be distributed to each user, therefore removing the need to publish such information.

Summary. Ultimately, the best choice of KAS will always depend on the requirements of the specific application setting and on the policy being enforced. The binary tree KAS we have proposed in this chapter appears to be a good well-rounded candidate and may be the best choice if derivation costs or storage requirements are a concern.

5.7 Conclusion

We have given a very simple KAS based on a binary tree and introduced heuristics to find 'good' mappings from the policy to a binary tree. We have proposed a novel technique of mapping the policy poset to a structure (i.e a binary tree) which is not some subset of the transitive closure of the Hasse diagram, as to optimise characteristics of the KAS (in our case, minimise key derivation). Our experimental evaluation also shows that the binary tree KAS performs well in practice in reducing the average number of intermediate secrets

required by users and logarithmically bounds derivation costs.

It is also important to consider how keys and intermediate secrets can be updated. KASs with public information [6, 36] may amend a portion of that information to define new intermediate secrets using the same derivation mechanism, but prior work [33, 34] has not considered how to perform updates without public information. A natural solution is to include counters in the PRF inputs when deriving intermediate secrets 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 should be a priority for future work.

Chapter 6

Cryptographic Enforcement Schemes for Information Flow Policies

Contents

In this chapter, we provide a rigorous definitional framework for a cryptographic enforcement scheme that enforces read-only information flow policies (which encompass many practical forms of access control, including role-based policies). This framework (i) provides a tool by which instantiations of CESs can be proven correct and secure, (ii) is independent of any particular cryptographic primitives used to instantiate a CES, and (iii) helps to identify the limitations of current primitives (such as key assignment schemes) as components of a CES.

This chapter is based on the following published work:

- *J. Alderman, J. Crampton and N. Farley, A Framework for the Cryptographic Enforcement of Information Flow Policies, SACMAT, 2017.*

6.1 Introduction

Many multi-user systems require some form of access control which requires specifying and enforcing a policy that defines the actions each user is authorised to perform. Traditionally, enforcement has required trusted on-line monitors to evaluate access requests. However, this approach is not necessarily appropriate for systems where the policy enforcement mechanism is not controlled by a trusted party (e.g. the policy author), or if the mechanism is not always available. An alternative is to use cryptographic techniques.

A cryptographic enforcement scheme (CES) to control read access to data objects must, at its most basic, provide a method to protect (encrypt) data and issue users the necessary cryptographic materials (keys) to access (decrypt) data that they are authorised to read. Furthermore, some CESs can support making changes to the policy, such as extending or retracting the access rights of a user, or changing the security level of a data object; such policy changes can have an effect on both the required cryptographic material, and on the security and correctness of the policy enforcement itself. Furthermore, as cryptographic material is vulnerable to compromise or leakage through exposure, a CES should provide a mechanism to refresh cryptographic material.

Whilst enforcement by a trusted monitor should be secure by design (i.e. permit only authorised requests) efficient cryptographic primitives are usually *computationally* secure (due to their probabilistic nature) [45]. Further, there may be real-world concerns to be addressed by an implementation that are not required in idealised, theoretical models. Thus, as observed by Ferrara *et al.* [45], there may exist a gap between the theoretical specification of an access control policy and a cryptographic implementation of an enforcement mechanism. Hence, one must carefully consider whether cryptographic primitives can achieve the correctness and security requirements to properly enforce an access control policy and, if multiple primitives are required, whether they can be safely combined. A vital part of such consideration is the establishment of rigorous definitions and security models for the required functionality of a CES.

To emphasise the gap between policy specification and cryptographic enforcement mechanisms, let us consider KASs [6] used to enforce an information flow policy (similar arguments can be made for other primitives such as functional encryption schemes). In general, KASs define how key material is generated, and derived, for a given access structure but do not define algorithms for encrypting objects, updating key material, or for carrying out changes to the policy. In fact, this additional functionality can have a significant effect on the cryptographic material supplied by the KAS — for example, assigning a user additional access rights may require extra keys to be securely distributed to the user, whilst the removal of a user typically requires that all of their keys (at least) be updated, under the assumption that removed users may locally store their keys and could continue to decrypt objects for which they are no longer authorised. If such changes are not implemented carefully, the security and correctness of the KAS itself could be compromised, as well as that of the CES as a whole.

6.1.1 Related Work

Many cryptographic enforcement mechanisms have been proposed, primarily to enforce *read* access to data objects via an encryption mechanism. Two particularly notable proposals are key assignment schemes (KASs) [3, 36], three of which we have considered in this thesis (in Chapters 3-5), and functional encryption schemes, especially *attribute-based encryption* (ABE) [18, 58]. Throughout this chapter, we shall periodically refer to both KASs and ABE as example cryptographic mechanisms that may be used within the context of a CES.

In general, *write* access can be more difficult to cryptographically enforce than *read* access and typically requires additional assumptions on the trustworthiness and capabilities of the storage provider, or additional trusted entities [37]. A common trust assumption is that the storage provider will correctly store data but is not trusted to enforce an access policy on the data and may try and learn the contents of the data. Such an assumption seems reasonable - one would expect a third party storage provider to store data correctly (in order to protect its reputation and business model) but may be given incentives to learn about the data (either to sell the data to another third party or to use for its own gains, e.g. targeted advertising to its users).

Whilst one can often use cryptographic primitives that provide data origin authentication to *detect* data originating from an unauthorised writer [54, 73], it can be difficult to *prevent* unauthorised writes to the (externally controlled) file system in the first place. Furthermore, to ensure correctness of the system following an unauthorised write, one must ensure the storage provider maintains the ability to 'roll-back' data objects or to otherwise ensure that legitimate writes are maintained. Ferrara *et al.* proposed a cryptographic enforcement scheme for role-based access control (RBAC) policies [44] which allows users to perform 'append only' writes to the file system. However, in order to ensure correctness after an unauthorised write to a file, such a scheme requires: (i) some form of version control on the file system; and (ii) entities requesting read access to verify versions of the file until they find a valid version. In recent work, Damgård *et al.* introduced the notion of *access control encryption* [37] which aims to restrict write access within an encryption scheme. Whilst this work certainly appears to be in a promising direction, it requires an additional trusted entity known as a *sanitizer* to process all data sent over public channels, and to perform sanitisation checks on data before it is written to the file system.

In this chapter, like most related work, we focus our attention on read-only policies, with the observation that mechanisms to support write access and the *detection* of unauthorised writes to the system should be a simple future extension to this work if required. Read-only policies are particularly useful for content distribution networks [49] and subscription services (e.g. Netflix, NowTV) which typically require a single writer and many readers.

Many works in the literature assume that the policy to be enforced is either static [18, 30, 77], or changes infrequently over time. However, as Garrison III *et al.* [54] suggest, such policies are "not representative of real-world systems" in which the policy and/or data changes frequently over time. In these situations, the system may need to support changes to the policy, such as the addition and deletion of users, the addition and removal of access rights, as well as changes to data. Whilst some of the literature describes how updates and changes to the policy should be cryptographically handled and/or what information needs updating as a result of making changes to the policy, such schemes are often not proven to be secure [6, 54, 84].

Ferrara *et al.* provide a formal security framework for (read-only) cryptographic RBAC policies [45] which support dynamic changes to the policy, and provide a secure instantiation based on predicate encryption. Garrison III *et al.* [54] attempt to "understand the

practical costs of leveraging public-key cryptographic primitives to implement outsourced dynamic access controls in the cloud". They produce constructions for enforcing a dynamic RBAC policy using identity-based and traditional public-key cryptography, and provide experimental results to show that the cost of enforcing such a policy using public key cryptography is computationally expensive. For example, for two of the datasets they considered, carrying out changes to the policy, such as removal of a user from a role could potentially require thousands of (identity-based) encryptions and over one hundred file re-encryptions [54]. Symmetric primitives may be preferable to public key primitives due to their smaller key sizes and cheaper cryptographic operations (as mentioned previously, public key encryption/decryption typically requires exponentiations and/or bilinear pairings [18, 22, 52, 58, 66, 76]).

Recall that key assignment schemes [3, 36] are symmetric cryptographic primitives that can be used to enforce read-only information flow policies. The security notions for KASs [6] capture the requirements that no (collusion of) users may compute a key for which they are unauthorised (*key recovery*), and the stronger notion that no information is leaked about keys for which users are unauthorised (*key indistinguishability (KI)*).

Key indistinguishability of a KAS states that a user who is not authorised to hold a *key* cannot learn anything about the key even having learned the keys (and intermediate secrets) of other unauthorised users. We argue that a secure CES requires that an unauthorised user attempting to access a particular *data object* cannot learn anything about the *data* written to that object[1] even if it can learn the keys of other unauthorised users, see the entire file system, know the data written to other objects, and force certain policy updates. In other words, security for KASs is defined in terms of decryption keys, whilst we consider the more relevant property of access to data objects which, as we will see, is not the same as prior security notions.

Clearly, without defining the required protection properties for objects, which keys are to be used, and how keys should be handled, it is not necessarily true that a lack of knowledge about a single key implies that nothing is learned about an object in a CES. Indeed, the logical combination of a KI-secure KAS and an IND-CPA secure encryption scheme [16] can be trivially *insecure* if, for example, the file system publishes secret keys

[1] In the context of a CES where data objects are stored on an externally controlled file system, we cannot prevent physical access to a data object but instead must protect the *data* written to an object from being learned by unauthorised entities.

defined by the KAS when writing data objects. Whilst this simple example is very easy to avoid, other scenarios may be more subtle, especially when using multiple, complex cryptographic primitives with intricate security properties in a system, such as a CES, comprising many components, entities and feasible execution paths. Thus we believe that the requirements of a CES system *as a whole* must be considered rather than just a single component. At the very least, it must be clear what the security and correctness objectives of the system are in order to select suitable cryptographic components.

To this end, Ferrara *et al.* [45] emphasise the importance of providing a formal model for secure cryptographic role-based access control (CRBAC). They describe how cryptographic access control schemes often only informally analyse the gap between policy specification and a proposed implementation. To illustrate this point, they describe how cryptographic guarantees are *probabilistic* whilst policies are *deterministic* (some party does/does not have access to some object). Gifford [55] previously presented a framework for cryptographic access control (including information flow), but could not, at the time, consider modern cryptographic security notions for computationally secure primitives, and presented separate models for symmetric and asymmetric primitives. In contrast, our framework provides formal cryptographic games to model correctness and security and is defined independently of particular cryptographic primitives. Further discussion of the relation between this work and the CRBAC framework proposed by Ferrara *et al.* [45] is found in Section 6.5.

6.1.2 Motivation

In order to ensure that a cryptographic mechanism adequately enforces an information flow policy, it is vital to have a rigorous and concrete framework to specify the functional, correctness and security requirements of a CES. The aim of this chapter is to introduce such a framework, which is intended to be useful to designers and implementers of CESs, both to guarantee the adequacy of existing proposals and to identify areas that need further research.

In this chapter, we consider CESs for read-only information flow policies. Crampton [30] showed that many access control policies of practical interest, such as attribute- and role-based policies, can be represented as information flow policies; therefore, our framework

is widely applicable and can be viewed as a continuation of the work of Ferrara *et al.* [45] (for RBAC policies) to bridge the gap between the specification of access control models and the capabilities of cryptographic primitives. In addition, it is interesting to consider how cryptographically enforcing a hierarchical access policy, such as an information flow policy, compares to enforcing a non-hierarchical policy such as Core RBAC, as considered by Ferrara *et al.* [45]. For example, one could imagine that if secrets or keys for labels are somehow related (e.g. derivable from one another), then additional care must be taken when one such key becomes compromised, since it may lead to other keys also becoming compromised. Indeed, as future work, Ferrara *et al.* [45] suggested modeling general access control frameworks; one can view our framework as a step towards this goal.

Whilst there is a wealth of work considering cryptographic access control requirements [1, 3, 36, 41, 58, 59, 61, 62, 54, 72], such works often focus on using particular cryptographic primitives or are tailored to a specific application. In contrast, we start from the specification of a general access control policy (information flow policies), from which we identify the requirements of a CES. We do not target any particular primitives and, instead, aim to provide a framework that can be instantiated by a range of cryptographic primitives, both symmetric and public key. We define several classifications of CESs based on their desired, generic, functionality. As a result, we hope to provide a framework within which one can analyse specific CES instantiations to ensure correctness and security.

We begin Section 6.2 by introducing some notation and recalling basic concepts related to information flow policies. We then introduce our model of CESs and classify the required functionality, before defining correctness and security in Section 6.3. In Section 6.4, we discuss some example schemes, constructed using cryptographic primitives often considered for cryptographically enforcing access control, and highlight their shortcomings in the context of our model. We conclude the chapter with a summary of our contributions and some ideas for future work.

6.2 Cryptographic Enforcement of Information Flow Policies

Recall that a *read-only information flow policy* is a tuple $P = ((L, \leqslant), U, O, \lambda)$, where:

- $(L, \leqslant)$ is a partially ordered set of *security labels*;
- U is the set of users;
- O is the set of data objects; and
- $\lambda : U \cup O \to L$ is a function mapping users and objects to security labels in L.

We say $u \in U$ is *authorised* to read an object $o \in O$ if $\lambda(o) \leqslant \lambda(u)$.

For simplicity, and without loss of generality, we may choose U and O to be arbitrarily large and fixed, and assume that L has a top element $\sqcap$ and a bottom element $\sqcup$; that is, for all $x \in L$, $x \leqslant \sqcap$ and $\sqcup \leqslant x$. For any object o that is inactive, we set $\lambda(o)$ equal to $\sqcap$; and for any user u that is inactive, we set $\lambda(u)$ to be $\sqcup$. No user is assigned to $\sqcap$ and no object is assigned to $\sqcup$. In other words, inactive objects cannot be read by any user, and inactive users cannot read any object. Then, to model the addition of a user or object, we can instead activate an inactive user or object by changing the security label from $\sqcup$ or $\sqcap$, respectively; users and objects can similarly be removed by setting the security label to $\sqcup$ or $\sqcap$.

Recently, we have seen considerable interest in outsourcing the storage of data. In this case, the storage provider, not the data owner, controls access to the data. Moreover, the storage provider may have incentives to inspect the data it stores on behalf of its clients (e.g. to sell data on to other third parties). Conversely, the data owner may not wish the storage provider to have read access to the data. Informally, the data owner may wish to encrypt data before giving it to the storage provider, thus preventing the storage provider (and any entity to which the storage provider releases the data) from reading the data. In addition, the data owner will distribute appropriate cryptographic material, such as decryption keys, to authorised users.

As mentioned, we focus on *read* access in this chapter. We assume that the data owner

(or a *manager* entity) is responsible for the protection of all objects and supplying the encrypted objects to the storage provider via an authenticated channel. (In practice, the manager could represent a set of authorised writers if required.) The storage provider simply stores all encrypted objects it is given and releases them on request to users. In other words, the storage system is essentially public and *all* users have access to *all* encrypted objects (but not all users have access to all decryption keys). We model the storage provider as an honest-but-curious adversary — it will store objects correctly and release them on request, but may try to learn information about the stored contents. Such a model seems to be the most realistic — one would expect that a storage provider, who wants to maintain a good reputation and good business, will correctly store data objects. However, the storage provider may want to learn the contents of the data a user stores on its servers for its own benefit, e.g. in order to better select advertisements to send to its users.

As mentioned in the introduction, it is important, especially when considering complex cryptographic primitives, to have a rigorous framework for the requirements of a CES, both to aid the design of CESs and to identify areas for future work. In this section, we formulate the requirements of a read-only CES, building from the access control requirements of the policy with no particular instantiation or cryptographic primitives in mind. Indeed, our definitions of the algorithms that a CES must implement are intentionally general, in order to cater for different possible instantiations. In particular, our definitions may be instantiated using symmetric or asymmetric cryptographic primitives. Where appropriate, we shall, however, refer to example instantiations to illustrate certain concepts.

6.2.1 State Requirements

In a CES, data objects are encrypted using some kind of cryptographic primitive and read access to an object is effected by decrypting. Thus, any CES needs to maintain a certain amount of cryptographic material, some of which will be public and some secret, held by different entities. We begin our development of a framework by considering the information, or state, that each entity within a CES must maintain, distinguishing between user, object and system states. We distinguish between an object (as created by the data owner) and its state in the system (in a protected format with any necessary metadata). We will then, in Section 6.2.2, consider the algorithmic requirements to use, maintain and

Notation	Meaning	Part of
$\mathrm{st}_{\mathcal{M}}$	State of the manager/system	-
$\tau(l)$	Secret material associated to label l	$\mathrm{st}_{\mathcal{M}}$
ϕ	Private additional information held by the manager	$\mathrm{st}_{\mathcal{M}}$
Pub	Public information including the file system FS	-
FS	Public file system	Pub
$\pi(l)$	Public material associated to label l	Pub
ψ	Additional public information	Pub
o	An object identifier	O
$d(o)$	Data written to o	-
$\overline{d(o)}$	Protected form of o	FS
u	A user identifier	U
st_u	State of user u	-

Table 6.1: Notation used for modelling states of entities.

update these states, which will lead us to consider a classification of CESs according to their functional requirements. Table 6.1 summarises the notation we shall introduce in the next section to describe states in a CES.

System. Clearly, within a CES, some cryptographic material must be generated. This is performed by the trusted system manager (or data owner), $\mathcal{M}$. The manager will also need to use some of the generated material to *protect* objects as they are written (recall that the manager performs all write operations in a read-only CES), to *refresh* existing material throughout the lifetime of the system, and to *grant access* to users (by distributing appropriate material). Therefore, the manager must store some or all of the material it generates for later use. We denote the *state*, containing all information currently held by the manager, by $\mathrm{st}_{\mathcal{M}}$.

In information flow policies, access is determined in terms of security labels. Hence, a CES for such policies may require, for each label $l \in L$:

- some secret information, denoted $\tau(l)$ (e.g. cryptographic material for performing encryption and decryption of objects that have security label l); and
- some public information, denoted $\pi(l)$ (e.g. public information to aid the derivation of $\tau(l)$ in a KAS).

Each user u must be provided with a means to learn some or all of $\tau(l)$ for all $l \leqslant \lambda(u)$.

Similarly, each object $o \in O$ must be protected using some or all of $\tau(\lambda(o))$.

The manager must store (or be able to efficiently regenerate) $\tau(l)$ for each label such that it may be issued to users when relevant. $\mathcal{M}$ may also require additional material to perform his duties (beyond that associated purely to labels), e.g. additional system parameters. We denote such material, which is known only to $\mathcal{M}$, by ϕ. The *private state* of $\mathcal{M}$ is therefore:

$$\mathrm{st}_{\mathcal{M}} = (\phi, \{\tau(l)\}_{l \in L}).$$

The manager must also make certain information publicly available to users and the storage provider. We have already seen that some public information, $\pi(l)$, related to security labels, may be required. In addition, the file system, FS, containing all protected objects (i.e. the information that is outsourced to the storage provider) is assumed to be publicly available (as any entity can request any outsourced data directly from the storage provider) and therefore forms part of the public state of the system. Finally, we may define ψ to be any additional public information required by a particular instantiation. The *public state* of the system is therefore:

$$Pub = (\psi, \{\pi(l)\}_{l \in L}, FS).$$

We refer to the state of the system as a whole as $\mathrm{st}_{\mathcal{M}}$ and Pub and note that, together, they model all information held in the system (we shall shortly introduce user states which will identify which components of the system state is held by which entities).

Data Objects. Each data object within a CES must be protected according to its security label. The protected object is written to a file system maintained by an untrusted storage provider.

In non-cryptographic settings for information flow policies, objects can be abstractly modelled *entirely* by an identifier and their security label — a reference monitor is guaranteed to permit or deny access to objects based only on consideration of security labels. This is *not* the case in a CES: the enforcement mechanism (encryption) operates not only on the label but also on the *content* of an object o (the data) and the cryptographic material ($\tau(\lambda(o))$ and $\pi(\lambda(o))$) associated to the label.

With these considerations in mind, we introduce the following notation to fully describe an object in O:

- o is a unique *identifier* which allows us to refer simply to an object and to apply the labelling function λ;
- $d(o)$ is the data written to the object o, to which we wish to control access; and
- $\overline{d(o)}$ denotes the protected form of o that is outsourced and to which all entities have access; we may assume that $\overline{d(o)}$ includes the label $\lambda(o)$.

Hence, we assume that the set of objects O is a set of object identifiers. We define $D(O) = \{(o, d(o)) : o \in O\}$ to be a set of tuples of the form $(o, d(o))$, where $d(o)$ is the plaintext data that should be written to object o on the file system during system setup (note that $d(o)$ may be equal to the empty string ϵ for some objects $o \in O$). Then the public data includes the file system FS which contains a set of pairs of the form $(o, \overline{d(o)})$.[2] It may be helpful to think of o as a filename, $d(o)$ as the contents of a file and $\overline{d(o)}$ as the encrypted file content. Clearly, one can refer to the entire object simply by referring to the filename, and writing to the file may change the content $d(o)$ without changing the filename.

Users. A user u is authorised to read an object o if $\lambda(u) \geqslant \lambda(o)$. Hence, u must be given information (derived from material contained in $st_{\mathcal{M}}$) that enables u to decrypt objects. This information may simply be the decryption keys associated with labels $l \leqslant \lambda(u)$, or data that enables the derivation of those keys. For example, in many KASs [6], a user $u \in U$ is given a user secret comprising a single intermediate secret $s_{\lambda(u)}$ enabling the derivation of decryption keys associated to any $y \leqslant \lambda(u)$. We may assume that st_u contains the label $\lambda(u)$.

[2] Note that we aim to protect only $d(o)$, and not any further meta-data of objects. In particular, the identifiers and security labels of objects are assumed to be public such that users can efficiently decide which objects to retrieve from the file system and how to decrypt them.

6.2.2 Functional Requirements

Having determined the minimal information that each entity must hold within a CES, we now look at the required algorithms. We shall see that one can model many different forms of CES depending on the required functionality, and this shall lead us to produce a classification of CESs.

To achieve a general definition satisfiable by *any* suitable cryptographic primitives, we have strived to define general, abstract input and output parameters for each algorithm that act as general 'containers', into which one can place the required cryptographic components of the particular primitives in use. Whilst our definitions may appear complicated, due to their generality, we believe that they give the simplest possible definition of a CES, since they show the required information flow between algorithms without relating parameters with their supposed format within a particular instantiation (e.g. we do not specify that an input is a cryptographic key, but a more general parameter that may or may not contain one or more keys when instantiated by a particular construction). For example, looking at the Setup algorithm, we will see that to initialise the system one must specify the policy to be enforced and the level of security required, and the algorithm simply generates some private information (state) for each entity (manager and users) and some public information accessible to all. We shall see concrete examples of how such a CES can be instantiated in Section 6.4.

A CES must support, at least, the following algorithms:

$$(st_{\mathcal{M}}, \{msg_u\}_{u \in U}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, P, D(O));$$
$$(m \ \text{ or } \ \perp) \leftarrow \mathsf{Read}(o, \mathrm{st}_u, Pub).$$

Setup is probabilistic and takes the policy $P = ((L, \leqslant), U, O, \lambda)$, set of data $D(O)$ (including object identifiers) to be written to the file system and a security parameter 1^ρ as input. (Informally, ρ determines the strength of cryptographic keys, and thus affects parameters such as key length.) It generates an initial system state ($\mathrm{st}_{\mathcal{M}}$ and Pub) enabling the remaining algorithms of the CES to run, and a set of messages $\{msg_u\}_{u \in U}$ that will be sent to users so that users can initialise their respective user states, st_u. The initial data $d(o)$ for all objects $o \in O$ (contained in $D(O)$) is protected and written to the file system (within Pub).

We assume that msg_u is sent over a secure channel to the user $u \in U$. In effect, we assume that any messages sent by the manager to users are received as intended and without leaking any information to an adversary. (However, as we discuss in Section 6.3.2, we will allow an adversary to corrupt users, thereby allowing the adversary to learn user state.)

Read, run by a user u, is a deterministic algorithm which takes as input the identifier of an object to which access is being requested, the state of the user requesting access, and the public information for the CES, which includes the file system and, in particular, $\overline{d(o)}$. The algorithm uses the cryptographic material contained within st_u (and perhaps Pub) to attempt to remove the protection mechanism applied to the data $d(o)$. It should output $m = d(o)$ (the data last written to o) if $\lambda(u) \geqslant \lambda(o)$, and an error symbol $\perp$ otherwise.

The Setup and Read algorithms alone are sufficient to provide the basic functionality required to enforce an information flow policy cryptographically — that is, Setup generates cryptographic material and protects objects, whilst Read removes the protection if the user is authorised. However, we note that it may be necessary, more efficient or otherwise convenient to extend the number of algorithms used. We now discuss some of these alternatives.

6.2.2.1 Writeable

Although Setup writes the initial data $d(o)$ contained in $D(O)$ for each object in O, in many systems one may wish to update the data associated with a given object and stored on the file system over the course of the system lifetime. A *writeable* CES allows the manager to update the contents of data objects and supports the following algorithm:

$$Pub \xleftarrow{\$} \mathsf{Write}(o, d(o)', \mathrm{st}_{\mathcal{M}}, Pub).$$

This algorithm takes as input the object identifier o of the object to be written to, the data $d(o)'$ to be written to object o, the state of the manager, and public information. It outputs updated public information, which informally, should include $\overline{d(o)'}$ in FS.

In a *writeable* CES, it may be that objects written to the filesystem during the initial call to Setup are empty, (i.e. the input to Setup is $D(O) = \{(o, \epsilon) : o \in O\}$) and that each object $o \in O$ stored on the file system is written to via a call to the Write algorithm with

inputs $o, d(o)$.

6.2.2.2 Refreshability

Over time, cryptographic material may need to be *refreshed* if, for example, material is compromised or lost, or following the removal of an authorised user. Computing advances, prolonged key exposure or the threat of a long-running attack may also necessitate periodic key refreshing. Thus, many CESs should include a mechanism by which cryptographic material can be updated.

Whilst a trivial solution would be to update cryptographic material simply by re-running the Setup algorithm, this will update *all* keys within the system simultaneously. It is likely to be more efficient to provide a targeted Refresh algorithm (to be run by the manager):

$$(st_{\mathcal{M}}, \{msg_u\}_{u \in U}, Pub) \xleftarrow{\$} \mathsf{Refresh}(l, st_{\mathcal{M}}, Pub).$$

Refresh takes a label $l \in L$, the state $\mathrm{st}_{\mathcal{M}}$ of the manager and Pub as input (which, together, contain the material $\tau(l)$ and $\pi(l)$ associated to the target label), and outputs updated values of $\mathrm{st}_{\mathcal{M}}$ and Pub, along with a set of messages $\{msg_u\}_{u \in U}$, which may contain updated cryptographic material such that users can continue to decrypt and read data for which they are authorised.

We say that a CES is *refreshable* if it uses an efficient Refresh algorithm, rather than Setup, to update cryptographic material on a per-label basis. Refreshes may also result in changes to the cryptographic material associated with other security labels; we denote this set of labels by L'. In a CES instantiated using an iterative KAS [6] (see Section 2.5.3) for example, $L' = \{l' \in L : l' \leqslant l\}$. Following a refresh, therefore, we may need to update Pub, st_u for some users (typically those where $\lambda(u) \in L'$) and $\overline{d(o)}$ for objects o where $\lambda(o) \in L'$.

6.2.2.3 Dynamic Policy

In some settings, it may be that the security labels assigned to each object and user never change (the policy is static). The Setup algorithm may assign the appropriate labels and cryptographic materials for all users and objects, and write all objects to the file system. In some systems, however, a user or object's label may be changed to/from any label in L during the lifetime of the system (e.g. in the event that a user's role changes, a user gets promoted or an object becomes declassified). A basic (but perhaps inefficient) solution to fulfilling this requirement is to re-run the Setup algorithm with a modified labelling function λ.

A potentially more efficient approach is to introduce randomised algorithms ChUsL and ChObL, for changing a user and object's label respectively:

$$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg}_{\mathrm{u}}\}_{u \in U}, Pub) \xleftarrow{\$} \mathsf{ChUsL}(u, l', \mathrm{st}_{\mathcal{M}}, Pub);$$
$$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg}_{\mathrm{u}}\}_{u \in U}, Pub) \xleftarrow{\$} \mathsf{ChObL}(o, l', \mathrm{st}_{\mathcal{M}}, Pub).$$

Both algorithms take the identifier of the user or object and the new label $l' \in L$ to be assigned, along with the manager state and public information, and result in updated manager states and public information along with update messages for each user that may update the user state st_u.

Note that ChUsL may affect the states of other users (or the secret information $\tau(y)$ associated to labels $y \neq l'$). For example, if the access rights of u are *decreased* then the cryptographic material for all labels that u is no longer authorised for may need to be changed; subsequently, objects protected using keys that have been updated may require re-protecting. Typically, ChObL could be implemented by decrypting $\overline{d(o)}$, calling Refresh on $\lambda(o)$ [3] and re-encrypting $d(o)$ using $\tau(l')$.

In contrast to the framework proposed by Ferrara *et al.* [45] which includes AddUser and DeleteUser algorithms, we capture the functionality of such algorithms within ChUsL. Recall that we assume a large population of users, many of which may be assigned to the

[3] If the key associated to $l = \lambda(o)$ is not updated, then an attacker u may store $\overline{d(o)}$ prior to o's label being changed. Then if u becomes authorised for l and the key has not been updated, then u may decrypt $\overline{d(o)}$ despite not being authorised for o.

security label $\sqcup$. The 'creation' of a user may be modelled as the activation of a user that has been assigned to $\sqcup$ (i.e. the change of label assignment of that user from $\sqcup$ to some other security label), whilst user deletion can be modelled as the assignment of an existing user to $\sqcup$. We can create and delete objects in a similar fashion by assigning from and to the label $\sqcap$. We say a CES is *dynamic* if it supports ChUsL and ChObL.

6.2.2.4 Decentralised Updates

Note that several algorithms (Setup, ChUsL and Refresh) are run by the manager and require resulting updates to a user's local state st_u. Certainly, since user states are subsets of the manager state, the manager could compute the updated user state st_u for all u that are affected, and distribute msg_u containing st_u. We call this a *centralised update* as it is performed entirely by the manager. However, this may place an unnecessarily onerous burden on the manager. In some instantiations, a more efficient solution (in terms of manager workload and bandwidth costs) may be to provide each user u with (a smaller amount of) data that enables u to derive st_u themselves. For example, each user u could use some key derivation function to update their own user state using a counter value or nonce broadcast by the manager (as part of msg_u). Hence, we introduce a final algorithm UserUpdate, run by the user:

$$st_u \leftarrow \mathsf{UserUpdate}(st_u, msg_u, Pub).$$

6.2.2.5 Classes of CES

We have seen that CESs in different settings may require different functionality. In Table 6.2, therefore, we classify CESs according to their required properties. We do not claim this classification to be exhaustive but believe that it captures many of the generic requirements of CESs. Each class of CES also includes the algorithms of those in the Basic class, and classes may be combined. Each algorithm may return $\perp$ to denote failure if, for example, the inputs are invalid.

CES Class	Algorithms	Run by
Basic	Setup	Manager
	Read	User
Writeable	Write	Manager
Refreshable	Refresh	Manager
Dynamic	ChUsL	Manager
	ChObL	Manager
Decentralised	UserUpdate	User

Table 6.2: Algorithms required in different classes of CES.

6.3 Correctness and Security

We now formalise our correctness and security notions for CESs for read-only information flow policies.

For the purposes of this framework, we make the assumption that all updates following a state transition occur immediately, so that we can model a system being secure and correct at any time. In practice, one may need to lock files whilst updates are performed [54].

6.3.1 Correctness

Informally, an information flow policy is correctly enforced if all authorised requests are permitted — that is, if a user u can read any object o where $\lambda(o) \leqslant \lambda(u)$. When considering a cryptographic enforcement mechanism, we would like to consider a stronger notion of correctness in which a user can not only read any object o where $\lambda(o) \leqslant \lambda(u)$, but the data obtained by the user was also the last data written to that object. Thus we want to ensure that it is not possible for the system to enter a state in which an authorised user performing a Read operation does not receive the correct data (the last data that should have been written to the object). To do so, we model the system as a game, given in Figure 6.1, played between a *scheduler* $\mathcal{A}$, which can observe and control the execution of the system, and a *challenger*; by considering all such schedulers, we consider all possible valid sequences of algorithms.

The aim of the experiment (from the scheduler's perspective) is to force the system into a state in which the output of reading an object $o^\star$ does not equal the data that should have

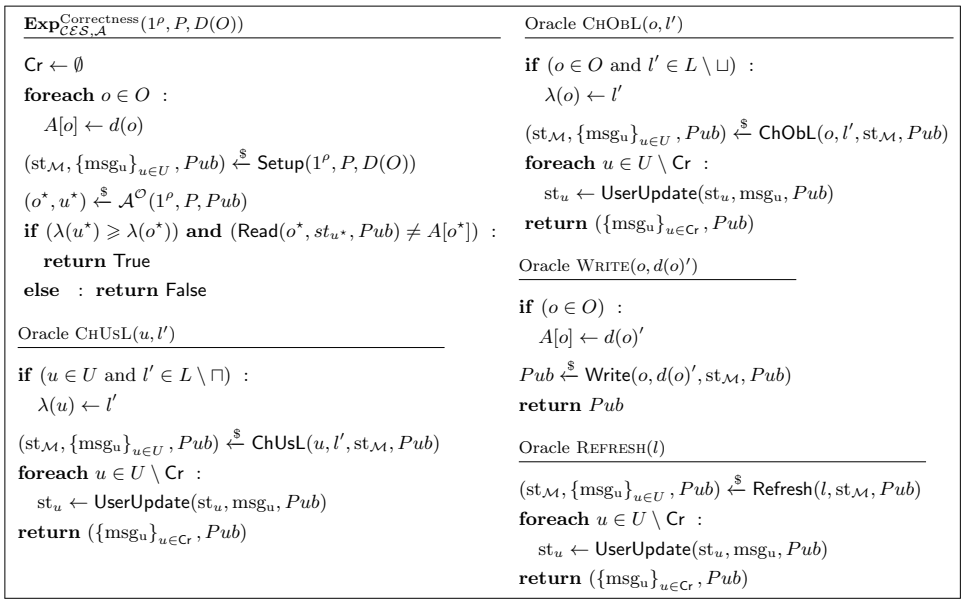


Figure 6.1: Correctness of a CES.

been last written to this object. We must ensure that the protection mechanism can be applied to, and removed from, data correctly by authorised users, and that the algorithms specified in the CES do not interfere with this operation. Recall that the storage provider is modelled as an honest-but-curious adversary; we therefore need not consider integrity properties since the provider is trusted to accept data only from the manager and to store it (unmodified) in the file system. In effect, we must ensure our specified algorithms conform to our expectation of a correct execution; we do not consider malicious storage providers that deviate from these algorithms in this work.

The experiment, given as $\mathbf{Exp}^{\text{Correctness}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$ in Figure 6.1, begins with the challenger setting up the system and initialising an array A, where $A[o]$ contains the data $d(o)$ for each object $o \in O$ defined in the policy; this array is used to store the data that (according to the policy and any subsequent write requests) should currently be stored by the storage provider. The challenger then gives $\mathcal{A}$ the public information and access to a set of oracles (also shown in Figure 6.1), which enables $\mathcal{A}$ to run CES algorithms on inputs of its choice. Most oracles simply check that the inputs are valid, update the policy or the array A as required, and then call the relevant CES algorithm. The CorruptU oracle allows the scheduler to learn the user state for a queried user (i.e. everything that the user knows) which models compromised or colluding users. The challenger maintains a list Cr of users that have been corrupted.

Recall that some algorithms output a set of update messages for some users. Messages

for users that the scheduler has corrupted are given to $\mathcal{A}$ (in this way, $\mathcal{A}$ learns any additional, leaked information from the update messages and can choose to update the corrupted user state itself in a *decentralised* CES). The challenger runs the $\mathsf{UserUpdate}$ algorithm to update the state of all *non-corrupted* users so that they remain synchronised with the remainder of the system, and so any future corruptions will reveal a correctly updated user state.

After polynomially many queries to the oracles, the scheduler selects a challenge object identifier $o^\star \in O$ and a user $u^\star \in U$. The challenger then runs Read for $o^\star$ using the state of the user $u^\star$. If $u^\star$ is authorised for $o^\star$, and Read does not output $A[o^\star]$ (the data that *should* have been most recently written to $o^\star$), the scheduler wins — it has found a sequence of state transitions that results in an authorised user not gaining the correct data. Otherwise, if the adversary does not have access to o, or $\mathsf{Read}(o) = A[o]$, then the adversary loses.

Definition 14. *Let $P = (\mathcal{P}, \mathcal{U}, O, \lambda)$ be an information flow policy. A cryptographic enforcement scheme $\mathcal{CES}$ for P is* correct *if, for all probabilistic polynomial-time schedulers $\mathcal{A}$, all valid policies P, all data arrays $D(O)$ and all security parameters $\rho \in \mathbb{N}$,*

$$\Pr\left[\mathsf{True} \leftarrow \mathbf{Exp}^{\text{Correctness}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))\right] = 0.$$

6.3.2 Security

Informally, a CES for a read-only information flow policy is *secure* if it denies all unauthorised read requests, i.e. a user u cannot learn $d(o)$ if $\lambda(u) \not\geqslant \lambda(o)$. A stronger cryptographic notion of security may require that unauthorised users can learn *nothing* about the contents of objects for which they are unauthorised.[4] Unlike an enforcement mechanism based on a reference monitor, there are often no absolute guarantees of security in a CES because cryptographic primitives are typically only computationally secure. Thus, security is defined in terms of the probability of an adversary learning something about an object that they are not authorised to read.

This ideal notion that unauthorised users learning nothing about an object for which they

[4] Whilst a user u who was authorised for an object o may have learned the contents of $d(o)$ prior to the object's label being changed such that u is no longer authorised for o, the user should not be able to read any further writes to o.

are not authorised can be viewed as a form of *semantic security* [57]. Unfortunately, it can be difficult to model exactly what is meant by an adversary learning 'no information' in arbitrary settings as one must account for any prior information the adversary may have about data in the file system (e.g. the language). Instead, it is common to consider an indistinguishability game [16] in which the adversary can choose data to be written to the filesystem (a chosen plaintext attack).

In our indistinguishability game for a CES, the adversary $\mathcal{A}$ chooses a challenge object (for which it is unauthorised) and two data values. The challenger chooses one of the data values at random and writes it to the chosen challenge object. To win, the adversary, having observed the file system, must state which data value was written. The adversary can expect to win 50% of the time by guessing; thus we model the adversary's advantage in this game as the difference between the probability of identifying the encrypted data correctly and $\frac{1}{2}$. For a secure CES, we require this advantage to be close to 0, i.e. the adversary cannot do (much) better than to guess.

This notion of indistinguishability implies the notion that a user is not able to learn $d(o)$ if $\lambda(u) \not\geqslant \lambda(o)$. Whilst the weaker notion requires only that the *entirety* of $d(o)$ is not revealed, our notion requires that *no* information about $d(o)$ may be leaked from an outsourced $\overline{d(o)}$ (even when the adversary may choose the data options to maximise its ability to distinguish the resulting protected data items). This ensures that the file system reveals nothing about written data (except perhaps metadata such as file-size); if any additional information were to leak, an adversary could win this game by choosing two messages that can be distinguished by the leaked information.

Our notion of *security* of a CES for an information flow policy $P = (\mathcal{P}, \mathcal{U}, O, \lambda)$ is captured in $\mathbf{Exp}^{\mathrm{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$ in Figure 6.2. The challenger $\mathcal{C}$ initialises an empty list Cr of corrupted users. $\mathcal{C}$ then initialises the system via Setup and then provides the adversary $\mathcal{A}$ with the public information and oracle access.

After polynomially many oracle queries, $\mathcal{A}$ chooses an object identifier $o^\star$ and two data items d_0 and d_1 (of equal length). $\mathcal{C}$ checks that no corrupted user in Cr is authorised for $o^\star$ (to prevent a trivial win for the adversary) and writes d_b to $o^\star$. The resulting public parameters, and oracle access, are given to the adversary who must correctly identify the data item written to the object.

$\mathbf{Exp}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$

$o^\star \leftarrow \perp$; $\mathsf{Cr} \leftarrow \emptyset$

$(\mathrm{st}_\mathcal{M}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, P, D(O))$

$(o^\star, d_0, d_1) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\rho, P, Pub)$

if $|d_0| \neq |d_1|$: **return** False

foreach $u \in \mathsf{Cr}$:

  if $\lambda(o^\star) \leqslant \lambda(u)$:

    return False

$Pub \xleftarrow{\$} \mathsf{Write}(o^\star, d_b, \mathrm{st}_\mathcal{M}, Pub)$

$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\rho, P, Pub)$

if $b = b'$: **return** True

else : **return** False

Oracle $\textsc{Write}(o, d(o)')$

$Pub \xleftarrow{\$} \mathsf{Write}(o, d(o)', \mathrm{st}_\mathcal{M}, Pub)$

return Pub

Oracle $\textsc{CorruptU}(u)$

if $u \notin U$: **return** $\perp$

if $\lambda(u) \geqslant \lambda(o^\star)$:

  return $\perp$

$\mathsf{Cr} \leftarrow \mathsf{Cr} \cup \{u\}$

return st_u

Oracle $\textsc{ChObL}(o, l')$

if $o = o^\star$:

  foreach $u \in \mathsf{Cr}$:

    if $l' \leqslant \lambda(u)$: **return** $\perp$

if $(o \in O$ **and** $l' \in L \setminus \sqcup)$:

  $\lambda(o) \leftarrow l'$

$(\mathrm{st}_\mathcal{M}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{ChObL}(o, l', \mathrm{st}_\mathcal{M}, Pub)$

foreach $u \in U \setminus \mathsf{Cr}$:

  $\mathrm{st}_u \leftarrow \mathsf{UserUpdate}(msg_u, \mathrm{st}_u)$

return $(Pub, \{msg_u\}_{u\in\mathsf{Cr}})$

Oracle $\textsc{ChUsL}(u, l')$

if $(u \in Cr$ **and** $\lambda(o^\star) \leqslant l')$: **return** $\perp$

if $(u \in U$ **and** $l' \in L \setminus \sqcap)$:

  $\lambda(u) \leftarrow l'$

$(\mathrm{st}_\mathcal{M}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{ChUsL}(u, l', \mathrm{st}_\mathcal{M}, Pub)$

foreach $u \in U \setminus \mathsf{Cr}$:

  $\mathrm{st}_u \leftarrow \mathsf{UserUpdate}(msg_u, \mathrm{st}_u)$

return $(Pub, \{msg_u\}_{u\in\mathsf{Cr}})$

Oracle $\textsc{Refresh}(l)$

$(\mathrm{st}_\mathcal{M}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{Refresh}(l, \mathrm{st}_\mathcal{M}, Pub)$

foreach $u \in U \setminus \mathsf{Cr}$:

  $\mathrm{st}_u \leftarrow \mathsf{UserUpdate}(msg_u, \mathrm{st}_u)$

return $(Pub, \{msg_u\}_{u\in\mathsf{Cr}})$

Figure 6.2: Security of a CES.

Oracles may perform 'housekeeping' to ensure that inputs are valid and do not permit a trivial win by allowing $\mathcal{A}$ to:

1. corrupt a user who is authorised for $o^\star$;
2. change the challenge object's label such that a corrupted user is now authorised for $o^\star$;
3. change a corrupted user's label such that the user is now authorised for $o^\star$.

Note that the set of oracles the adversary has access to depends on the class of CES. Recall that a non-refreshable CES may be (inefficiently) refreshed by recalling Setup with new policy inputs; we therefore provide a Refresh oracle so that the adversary can influence the manager to call Setup. A non-refreshable CES will replace the call to Refresh within the Refresh oracle with a call to Setup with the current policy as input. In our model,

we do not permit the poset to change over time, and hence the only input to the REFRESH oracle is the label to be refreshed; the adversary may not specify a new policy as this may include an alternative poset (permitted policy changes can be effected through other oracles). In a non-writeable CES, the call by the challenger in $\mathbf{Exp}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$ to Write to the challenge object (line 8) is replaced by a call to $\mathsf{Setup}(1^\rho, P, D(O))$ where $(o, d(o))$ in $D(O)$ is replaced by (o, d_b).

Whenever the policy is to be updated, the challenger updates the policy correctly and calls the relevant algorithm. Thus, the challenger's view of the policy is always correct, enabling the checks for trivial wins to be performed correctly.

We define the advantage of adversary $\mathcal{A}$ in the $\mathbf{Exp}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$ for a given CES $\mathcal{CES}$ to be:

$$\text{Adv}^{\text{Ind}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O)) = \left| \Pr\left[\mathsf{True} \leftarrow \mathbf{Exp}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))\right] - \frac{1}{2} \right|.$$

Definition 15. *A CES $\mathcal{CES}$ for an information flow policy is* secure *if, for all probabilistic polynomial-time adversaries $\mathcal{A}$, all valid policies P and data arrays $D(O)$, and all security parameters $\rho \in \mathbb{N}$,*

$$\text{Adv}^{\text{Ind}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O)) \leqslant \mathsf{negl}(\rho),$$

where negl *is a negligible function.*

One may observe that a *secure* CES, in accordance with Definition 15, must employ some form of *forward-security* (e.g. one should not be able to learn old versions of label keys). This prevents users locally storing ciphertexts for objects that *used* to be assigned to a security label l, obtaining authorisation for l, and being able to derive the old decryption key for l to enable successful decryption of such ciphertexts.

6.4 Example Instantiations

In this section, we look at some example instantiations of a CES.

$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, P)$

Parse P as $((L, \leqslant), U, O, \lambda)$

$(\{\sigma_x, \kappa_x\}_{x\in L}, Pub_{\mathcal{KAS}}) \xleftarrow{\$} \mathcal{KAS}.\mathsf{Setup}(1^\rho, (L, \leqslant))$

foreach $x \in L$:

  $\tau(x) \leftarrow \{\sigma_x, \kappa_x\}$

$\phi \leftarrow P$

$\mathrm{st}_{\mathcal{M}} \leftarrow (\phi, \{\tau(x) : x \in L\})$

foreach $u \in U$:

  $\mathrm{st}_u \leftarrow (\sigma_{\lambda(u)}, \lambda(u))$

foreach $o \in O$:

  $\overline{d(o)} \xleftarrow{\$} (\mathcal{SE}.\mathsf{Encrypt}_{\kappa_{\lambda(o)}}(d(o)), o, \lambda(o))$

$FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$

$\psi \leftarrow (Pub_{\mathcal{KAS}}, (L, \leqslant))$

$Pub \leftarrow (\psi, FS)$

return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, Pub)$

$d(o) \leftarrow \mathsf{Read}(o, \mathrm{st}_u, Pub)$

if $o \notin O$: **return** $\perp$

$\kappa_{\lambda(o)} \leftarrow \mathcal{KAS}.\mathsf{Derive}(\lambda(o), \lambda(u), \sigma_{\lambda(u)}, Pub_{\mathcal{KAS}})$

if $\kappa_{\lambda(o)} \neq \perp$:

  Parse $\overline{d(o)}$ as $(c_o, o, \lambda(o))$

  return $\mathcal{SE}.\mathsf{Decrypt}_{\kappa_{\lambda(o)}}(c_0)$

return $\perp$

$Pub \xleftarrow{\$} \mathsf{Write}(o, d(o)', \mathrm{st}_{\mathcal{M}}, Pub)$

if $o \notin O$: **return** $\perp$

$\overline{d(o)} \xleftarrow{\$} (\mathcal{SE}.\mathsf{Encrypt}_{\kappa_{\lambda(o)}}(d(o)'), o, \lambda(o))$

$FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$

$Pub \leftarrow (\psi, FS)$

return Pub

Figure 6.3: A Writeable, Centralised CES using a KAS.

6.4.1 KAS instantiation

Figure 6.3 gives an example CES instantiation $\mathcal{CES}$ using a KAS $\mathcal{KAS}$ and a symmetric encryption scheme $\mathcal{SE}$ where the key space for $\mathcal{KAS}$ and $\mathcal{SE}$ is the same. We use Pub to denote the public information output by $\mathcal{CES}$ and $Pub_{\mathcal{KAS}}$ to denote the public information output by $\mathcal{KAS}$. The manager state includes all generated keys and secrets; each user state includes the user secret assigned to the user's security label, and Pub includes the public information output by the KAS.

Theorem 8. *Let $\mathcal{KAS}$ be secure in the sense of strong key indistinguishability and let $\mathcal{SE}$ be IND-CPA secure. Then the instantiation in Figure 6.3 is a secure static, writeable, centralised, non-refreshable CES.*

Proof. We first define a modified game, **Game 1**, which is the same as that shown in Figure 6.2 (which we call **Game 0**) except that the key used to encrypt the challenge object $o^\star$ is chosen randomly rather than derived within the KAS. We show that an adversary cannot distinguish **Game 1** from **Game 0** with non-negligible advantage. Therefore, we may run the adversary against **Game 1**, and, with all but negligible probability, the adversary will run correctly.

Having transitioned to **Game 1**, we will be in a position where the challenge encryption is

generated using a random key; therefore we can reduce security to IND-CPA of the symmetric encryption scheme. We show that if an adversary $\mathcal{A}_{CES}$ can break the security of our CES, then we can construct an adversary $\mathcal{A}_{IND}$ that, using $\mathcal{A}_{CES}$ as a subroutine, can break the IND-CPA security of the symmetric encryption scheme. Since the encryption scheme is assumed to be secure, such an adversary should not exist; therefore a successful adversary against the CES cannot exist.

We first show that **Game 1** is indistinguishable from **Game 0**. Suppose, for contradiction, that $\mathcal{A}_{CES}$ is an adversary that can distinguish these games, using at most $q = poly(1^\rho)$ queries to the $\mathsf{Refresh}$ oracle. Let $\mathcal{C}_{KI}$ be a challenger for the SKI game. We construct an adversary $\mathcal{A}_{KI}$ which uses $\mathcal{A}_{CES}$ to break the SKI security of the KAS.

$\mathcal{A}_{KI}$ must simulate either **Game 0** or **Game 1** for $\mathcal{A}_{CES}$. It forms a policy P, using $(L, \leqslant)$ from its game with $\mathcal{C}_{KI}$, and its choice of U, O and λ. Note that $\mathcal{A}_{KI}$ is given a single challenge key for a single security label and that, in this static CES, all keys are replaced whenever $\mathsf{Refresh}$ is called. Thus, to correctly embed the SKI challenge into **Game 0** or **Game 1** before $\mathcal{A}_{CES}$ decides its challenge parameters, $\mathcal{A}_{KI}$ must guess the challenge label that $\mathcal{A}_{CES}$ will choose *and* which version of that key will be challenged (i.e. how many times $\mathsf{Refresh}$ will be called before the challenge). Let r be a counter, initially 0, denoting the number of calls $\mathcal{A}_{CES}$ makes to $\mathsf{Refresh}$. Thus, $\mathcal{A}_{KI}$ must guess $c \xleftarrow{\$} L$ for the challenge label *and* guess $i \xleftarrow{\$} \{0, 1, \ldots, q\}$ for the value of r when the challenge parameters are chosen.

The proof is as follows:

1. $\mathcal{A}_{KI}$ sends its SKI challenge label $c \in L$ to $\mathcal{C}_{KI}$.
2. $\mathcal{C}_{KI}$ runs $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub_{\mathcal{KAS}}) \xleftarrow{\$} \mathcal{KAS}.\mathsf{Setup}(1^\rho, (L, \leqslant))$. If $b = 0$, then the challenge key $\kappa_b^\star = \kappa_c$, else $\kappa_b^\star$ is chosen randomly from the key space.
3. $\mathcal{C}_{KI}$ sends $Pub_{\mathcal{KAS}}$, $Corrupt_c$ and $Keys_c$ (see Figure 2.6) to $\mathcal{A}_{KI}$.
4. $\mathcal{A}_{KI}$ initialises $\mathsf{Cr} = \emptyset$ and $o^\star = \perp$.
5. Now, if $i \neq 0$, then $\mathcal{A}_{KI}$ does not embed $\mathcal{C}_{KI}$'s outputs in the initial CES setup. Instead, it runs Setup as in Figure 6.3, running $\mathcal{KAS}.\mathsf{Setup}$ itself. Else (when $i = 0$), $\mathcal{A}_{KI}$ sets $\mathrm{st}_{\mathcal{M}}$ to include $Corrupt_c$ and $Keys_c$ and Pub to include $Pub_{\mathcal{KAS}}$.

6. In the CES game, for each user $u \in U$, if $\lambda(u) \not\geqslant c$, $\mathcal{A}_{KI}$ defines $\mathrm{st}_u = \{\sigma_{\lambda(u)}, \lambda(u)\}$, and $\mathrm{st}_u = \{\cdot, \lambda(u)\}$ otherwise, where $\sigma_{\lambda(u)} \in Corrupt_c$.

7. $\mathcal{A}_{CES}$ is given Pub and a set of oracles $\mathcal{O}$ as in Figure 6.2.

8. If $\mathcal{A}_{CES}$ calls $\mathsf{CorruptU}$ on a user $u \in U$ where $\lambda(u) \geqslant c$, then $\mathcal{A}_{KI}$ loses the game ($\mathcal{A}_{CES}$ would now not be allowed to choose c as his challenge and so $\mathcal{A}_{KI}$'s initial guess of c was wrong, so he would not be able to win). Similarly, $\mathcal{A}_{KI}$ loses if $\mathcal{A}_{CES}$ chooses a challenge object $o^\star$ such that $\lambda(o^\star) \neq c$.

9. Whenever the $\mathsf{Refresh}$ oracle is called, r is increased by 1. When $r = i$ (and $i \neq 0$), $\mathcal{A}_{KI}$ runs $\mathsf{Refresh}$ but instead of running $\mathcal{KAS}.\mathsf{Setup}$, it uses the key material received from $\mathcal{C}_{KI}$, and re-initialises the state of the manager, users, and objects as described above in Step 5 whens $i = 0$. $\mathcal{A}_{KI}$ loses the game if r exceeds i and $\mathcal{A}_{CES}$ has not yet chosen a challenge object.

10. Eventually, $\mathcal{A}_{CES}$ guesses that it was playing **Game** b'. If $r \neq i$, $\mathcal{A}_{KI}$ loses the game.

11. $\mathcal{A}_{KI}$ forwards b' to $\mathcal{C}_{KI}$ as its guess of whether the key for the challenge label was real ($b = 0$) or random ($b = 1$).

$\mathcal{A}_{KI}$ wins with non-negligible probability $\frac{\mathsf{Adv}(\mathcal{A}_{CES})}{(q+1)|L|}$. Since the KAS is assumed SKI-secure, such a distinguisher $\mathcal{A}_{CES}$ with non-negligible advantage cannot exist. We can therefore hop from **Game 0** to **Game 1**.

We now show that if an adversary $\mathcal{A}_{CES}$ playing **Game 1** can identify the message written to a challenge object with non-negligible probability, then an adversary $\mathcal{A}_{IND}$ playing the LOR IND-CPA security game for $\mathcal{SE}$ (see Figure 2.2) can use $\mathcal{A}_{CES}$ to win the LOR IND-CPA game against a challenger C_{IND}.

1. C_{IND} randomly selects a key k from the key space and gives $\mathcal{A}_{IND}$ 1^ρ and access to the LOR oracle (as shown in Figure 2.2) which takes two messages m_0, m_1 of the same length and always outputs the encryption of m_b under key k. (We use the Left-or-Right version of the IND-CPA experiment, as shown in Figure 2.2 instead of the Find-then-Guess version [16] as it allows multiple challenges; thus we need only guess the challenge label and not the object itself.) For this game, $\mathcal{A}_{IND}$ guesses the value of b; let us denote his guess by $b^\star$.

2. $\mathcal{A}_{IND}$ runs line 1 of the CES experiment (Figure 6.2) and guesses the security label c of the challenge object $o^\star$ that $\mathcal{A}_{CES}$ will choose. Let r be a counter initialised to 0, denoting the number of calls $\mathcal{A}_{CES}$ makes to Refresh. $\mathcal{A}_{IND}$ guesses $i \xleftarrow{\$} \{0, 1, \ldots, q\}$ for the value of r when the challenge parameters are chosen. Whenever the Refresh oracle is called, r is increased by 1.

3. $\mathcal{A}_{IND}$ runs line 2 of the CES experiment. If $i = 0$, then all encryptions using the key κ_c will be replaced by encryptions under k. That is, when an object o with label c is to be written, the adversary $\mathcal{A}_{IND}$ calls the LOR oracle on inputs $(d(o), d(o))$ to obtain an encryption under k.

4. $\mathcal{A}_{IND}$ gives oracle access to $\mathcal{A}_{CES}$.

5. When Refresh is called such that i becomes equal to r, all encryptions using the key κ_c will be replaced by encryptions under k (as described in Step 3).

6. When r = i, if $\mathcal{A}_{CES}$ corrupts a user $u \in U$ such that $\lambda(u) \geqslant c$ before the challenge object has been chosen, the experiment fails (as the CES is not dynamic, $\mathcal{A}_{CES}$ can no longer choose a challenge object $o^\star$ such that $\lambda(o^\star) = c$ and thus $\mathcal{A}_{IND}$'s guess of c was wrong).

7. Eventually, $\mathcal{A}_{CES}$ chooses a challenge object $o^\star$ and two messages m_0, m_1. If $\lambda(o^\star) \neq c$, the experiment fails. Similarly, the experiment fails if $r \neq i$.

8. If $\mathcal{A}_{CES}$ calls Refresh after the challenge object has been chosen, then $\mathcal{A}_{IND}$ encrypts $m_{b^\star}$ under κ_c and writes the result to $\overline{d(o^\star)}$.

9. After $\mathcal{A}_{CES}$ has finished querying his oracles, $\mathcal{A}_{CES}$ sends b' to $\mathcal{A}_{IND}$ as its guess of b.

10. $\mathcal{A}_{IND}$ forwards b' to C_{IND} as its guess. If $\mathcal{A}_{CES}$ calls Refresh after choosing his challenge object, then the game only succeeds if $b^\star = b$, else the game fails.

If $\mathcal{A}_{CES}$ can correctly guess which data was written with non-negligible advantage $\mathrm{Adv}_{\mathcal{A}_{CES}}$, then $\mathcal{A}_{IND}$ wins the IND-CPA game at least with non-negligible advantage

$$\frac{1}{2(q+1)\,|L|} \cdot \mathrm{Adv}_{\mathcal{A}_{CES}}.$$

This is a contradiction, since the encryption scheme is assumed IND-CPA secure. □

It is interesting to note that, although KASs are often proposed as symmetric cryptographic enforcement mechanisms for information flow policies, the natural pairing of a KI-secure KAS and an IND-CPA secure encryption scheme yields a rather basic CES according to our classifications. Indeed, it appears that constructing a richer class of CES using current KASs as a black box (i.e. using the defined algorithms without using the particular details of a specific instantiation) would be challenging. Current KASs specify only two algorithms and the Setup algorithm generates and outputs *all* public and secret information for the entire system; there is no alternative method by which to generate subsets of this information. Thus allowing for dynamic or refreshable CESs will be problematic — there is no mechanism by which a single key can be generated or replaced for example. Whilst some KAS constructions do allow for some aspects to be altered [7], this mechanism is scheme dependent and does not form part of the definition or, crucially, the security model. Future work on KASs should aim to meet the requirements of our proposed framework if they are to ensure utility as a component of a CES; in particular, a KAS used to instantiate a more complex CES will require algorithms to update and refresh components, and the KI security notion will need to be adapted to accommodate changes to cryptographic material over time.

6.4.2 KP-ABE instantiation

Our second example uses a large-universe key-policy attribute-based encryption (KP-ABE) [58] scheme, as described in Section 2.3.3. Figure 6.4 gives an instantiation of a dynamic, centralised, refreshable, writeable CES.

Each security label is associated with an attribute. Objects are encrypted using the singleton attribute set $\{\lambda(o)\}$ and user decryption keys are generated using the disjunctive policy $\bigvee_{l \leqslant \lambda(u)} l$; hence users can decrypt any object where $\lambda(o) \leqslant \lambda(u)$ as required. Whilst more efficient instantiations are likely possible (e.g. using revocable KP-ABE [10]), we have aimed here to use a simple, standard KP-ABE scheme. We use a large-universe construction (where any string can be an attribute) to enable 'versions' of attributes to disable out-of-date keys (a counter is appended to each attribute and is updated whenever a user loses access to an object assigned that attribute). That is, we define a counter $A[l]$ for each label $l \in L$ (where $A[l]$ is initialised to 0), and associate each label $l \in L$ with the attribute (set) $l||A[l]$. Then, whenever cryptographic material associated to l needs

$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, P, D(O))$

Parse $P = ((L, \leqslant), U, O, \lambda)$
$(MK, PP) \xleftarrow{\$} \mathcal{ABE}.\mathsf{Setup}(1^\rho)$
for $l \in L$:
  $A[l] \leftarrow 0$
$\kappa_M \xleftarrow{\$} \mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{l\in L} l||A[l]), MK, PP)$
$\phi \leftarrow (A, \kappa_M, P, MK)$
$\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
foreach $u \in U$:
  $k_u \xleftarrow{\$} \mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{l\leqslant\lambda(u)} l||A[l]), MK, PP)$
  $\mathrm{st}_u \leftarrow (\kappa_u, \lambda(u))$
foreach $o \in O$:
  $c_o \xleftarrow{\$} \mathcal{ABE}.\mathsf{Encrypt}(d(o), \{\lambda(o)||A[\lambda(o)]\}, PP)$
  $\overline{d(o)} \leftarrow (c_o, o, \lambda(o))$
$FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
$\psi \leftarrow (PP, (L, \leqslant))$
$Pub \leftarrow (\psi, FS)$
return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, Pub)$

$d(o) \leftarrow \mathsf{Read}(o, \mathrm{st}_u, Pub)$

if $o \in O$:
  Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
  Parse $\mathrm{st}_u = (\kappa_u, \lambda(u))$
  return $\mathcal{ABE}.\mathsf{Decrypt}(c_o, \kappa_u, PP)$
return $\perp$

$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg_u}\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, Pub)$

if $l \in L$:
  $A[l] = A[l] + 1$
  $\kappa'_M \xleftarrow{\$} \mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{l\in L} l||A[l]), MK, PP)$
  foreach $u \in \{u' \in U : l \leqslant \lambda(u')\}$:
    $\mathrm{st}_u \xleftarrow{\$} (\mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{l'\leqslant\lambda(u)} l'||A[l']), MK, PP), \lambda(u))$
  foreach $o \in \{o' \in O : \lambda(o') = l\}$:
    Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
    $d \leftarrow \mathcal{ABE}.\mathsf{Decrypt}(c_o, \kappa_M, PP)$
    $\overline{d(o)} \xleftarrow{\$} (\mathcal{ABE}.\mathsf{Encrypt}(d, \{\lambda(o)||A[\lambda(o)]\}, PP), o, \lambda(o))$
  $\phi \leftarrow (A, \kappa'_M, P, MK)$
  $\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $Pub \leftarrow (\psi, FS)$
  return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, Pub)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, Pub)$

$(\mathrm{st}_{\mathcal{M}}, \{msg_u\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{ChObL}(o, l', \mathrm{st}_{\mathcal{M}}, Pub)$

if $o \in O$ and $l' \in L \setminus \sqcup$:
  $l \leftarrow \lambda(o)$
  Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
  $d \leftarrow \mathcal{ABE}.\mathsf{Decrypt}(c_o, \kappa_M, PP)$
  $\overline{d(o)} \xleftarrow{\$} \mathcal{ABE}.\mathsf{Encrypt}(d, \{l'||A[l']\}, PP), o, l')$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $\lambda(o) \leftarrow l'$
  $Pub \leftarrow (\psi, FS)$
  return $\mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, Pub)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, Pub)$

$(\mathrm{st}_{\mathcal{M}}, \{msg_u\}_{u\in U}, Pub) \xleftarrow{\$} \mathsf{ChUsL}(u, l', \mathrm{st}_{\mathcal{M}}, Pub)$

if $u \in U$ and $l' \in L \setminus \sqcap$:
  $X \leftarrow \{l \in L : l \leqslant \lambda(u), l \not\leqslant l'\}$
  foreach $x \in X$:
    $A[x] = A[x] + 1$
    foreach $o \in \{o \in O : \lambda(o) = x\}$:
      Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
      $d \leftarrow \mathcal{ABE}.\mathsf{Decrypt}(c_o, \kappa_M, PP)$
      $\overline{d(o)} \xleftarrow{\$} (\mathcal{ABE}.\mathsf{Encrypt}(d, \{\lambda(o)||A[\lambda(o)]\}, PP), o, \lambda(o))$
  if $X \neq \emptyset$:
    $\kappa_M \xleftarrow{\$} \mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{l\in L} l||A[l]), MK, PP)$
    $\phi \leftarrow (A, \kappa_M, P, MK)$
    $\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
    $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
    $Pub \leftarrow (\psi, FS)$
    foreach $u' \in \{u' \in U \setminus u : \exists x \in X, x \leqslant \lambda(u')\}$:
      $st_{u'} \xleftarrow{\$} (\mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{x\leqslant\lambda(u')} x||A[x]), MK, PP), \lambda(u'))$
  $\lambda(u) \leftarrow l'$
  $\mathrm{st}_u \xleftarrow{\$} (\mathcal{ABE}.\mathsf{KeyGen}((\bigvee_{x\leqslant l'} x||A[x]), MK, PP), l')$
  return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, Pub)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, Pub)$

$Pub \xleftarrow{\$} \mathsf{Write}(o, d(o)', \mathrm{st}_{\mathcal{M}}, Pub)$

if $o \in O$:
  $\overline{d(o)} \xleftarrow{\$} (\mathcal{ABE}.\mathsf{Encrypt}(d(o)', \{\lambda(o)||A[\lambda(o)]\}, PP), o, \lambda(o))$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $Pub \leftarrow (\psi, FS)$
return Pub

Figure 6.4: Construction of a Dynamic, Centralised, Refreshable, Writeable CES using attribute-based encryption.

updating, we increment $A[l]$ by 1 and define $l||A[l]$ to be the new attribute associated to label l.

Theorem 9. *Let $\mathcal{ABE}$ be a fully secure IND-CPA large-universe KP-ABE scheme. Then the instantiation $\mathcal{CES}$ in Figure 6.4 is a secure dynamic, centralised, refreshable, writeable*

CES.

In order to prove the above theorem, we first introduce a left-or-right (LOR) version of the FTG IND-CPA security experiment (see Figure 2.4), which is shown in Figure 6.5. In this experiment, the adversary has access to another oracle, LOR. Unlike the FTG experiment shown in Figure 2.4, in which the adversary can only ask for a single ciphertext challenge for a given set of attributes A and set of messages $\{m_0, m_1\}$, an adversary in the LOR experiment in Figure 6.5 can ask for polynomially many challenge ciphertexts via calls to the LOR oracle. Similarly to the FTG experiment, a challenger $\mathcal{C}$ in the LOR experiment records the access structures the adversary has a decryption key for in the list X_{lor}. The challenger also records the attribute sets that the adversary has called a LOR query on (and thus requested a challenge ciphertext for) in the Ch_{lor} list.

In this experiment, a query to the KeyGen oracle with input $\mathbb{A}$ is only valid if there does not exist an attribute set $A' \in Ch_{lor}$ which satisfies $\mathbb{A}$. If this check fails, a distinguished failure symbol $\perp$ is returned. A query to the LOR oracle with inputs (m_0, m_1, A) is only valid if $|m_0| = |m_1|$ and the attribute set A does not satisfy any access structure $\mathbb{A}$ associated with a key which the adversary possesses. Thus, informally, an adversary in the LOR experiment can request more than one ciphertext challenge, and therefore may have more information to assist him in winning the LOR experiment than an adversary in the FTG experiment.

The advantage $\mathrm{Adv}^{\mathrm{lor-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho)$ of an adversary $\mathcal{A}$ in the Left-or-Right IND-CPA experiment shown in Figure 6.5 for a large attribute universe KP-ABE scheme $\mathcal{ABE}$ is defined as [16]:

$$\mathrm{Adv}^{\mathrm{lor-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) = \left| \Pr\left[\mathbf{Exp}^{\mathrm{lor-cpa-1}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) = 1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{lor-cpa-0}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) = 1\right] \right|.$$

Definition 16. *A large attribute-universe key-policy attribute-based encryption scheme $\mathcal{ABE}$ is (fully) secure (in the LOR IND-CPA case) if, for all PPT adversaries $\mathcal{A}$, for all $\rho \in \mathbb{N}$,*

$$\mathrm{Adv}^{\mathrm{lor-cpa}}_{\mathcal{ABE},\mathcal{A}}(1^\rho) \leqslant \mathsf{negl}(\rho),$$

where negl *is a negligible function.*

The advantage of an adversary in the IND-CPA experiment for large attribute universe KP-ABE schemes can be quantified in terms of their advantage in the FTG experiment.

$\mathbf{Exp}^{\text{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}}(1^\rho)$

$(MK, PP) \xleftarrow{\$} \mathcal{ABE}.\mathsf{Setup}(1^\rho)$
$X_{lor}, Ch_{lor} \leftarrow \emptyset$
$\mathbb{A}^\star \leftarrow \emptyset$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\rho, PP)$
return $b' = b$

Oracle KeyGen($\mathbb{A}$)

for $c \in Ch_{lor}$
  if $c \in \mathbb{A}$
    return $\perp$
$X_{lor} \leftarrow X_{lor} \cup \{\mathbb{A}\}$
return $\mathcal{ABE}.\mathsf{KeyGen}(\mathbb{A}, MK, PP)$

Oracle LOR(m_0, m_1, A)

if $|m_0| \neq |m_1|$:
  return $\perp$
for $\mathbb{A} \in X_{lor}$:
  if $A \in \mathbb{A}$:
    return $\perp$
$Ch_{lor} \leftarrow Ch_{lor} \cup \{A\}$
return $\mathcal{ABE}.\mathsf{Encrypt}(m_b, A, PP)$

Figure 6.5: Left-or-Right (LOR) IND-CPA security experiment (Figure 2.4) for large attribute-universe KP-ABE Schemes.

Lemma 10. *Let $\mathcal{ABE}$ be a large attribute universe KP-ABE scheme. Let* $\mathrm{Adv}^{\text{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$ *be the advantage of an adversary $\mathcal{A}_{ftg}$ in the experiment* $\mathbf{Exp}^{\text{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$ *shown in Figure 2.4. Let* $\mathrm{Adv}^{\text{lor-cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ *be the advantage of an adversary $\mathcal{A}_{lor}$ in the experiment* $\mathbf{Exp}^{\text{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ *shown in Figure 6.5 who makes $q = poly(1^\rho)$ queries to their* LOR *oracle. Then,*

$$\mathrm{Adv}^{\text{lor-cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) \leqslant q \cdot \mathrm{Adv}^{\text{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho).$$

Proof. The proof is similar to those provided by Bellare *et al.* [16, 17] for showing the relation between the Find-then-Guess and Left-or-Right experiments for IND-CPA security of both asymmetric and symmetric encryption schemes.

We prove Lemma 10 by constructing an adversary $\mathcal{A}_{ftg}$ that uses an adversary $\mathcal{A}_{lor}$, playing Experiment $\mathbf{Exp}^{\text{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$, as a subroutine in order to try to win $\mathbf{Exp}^{\text{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$. Informally, $\mathcal{A}_{ftg}$ plays the Experiment $\mathbf{Exp}^{\text{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$ with a challenger $\mathcal{C}$. During this game, $\mathcal{A}_{ftg}$ acts as the challenger in experiment $\mathbf{Exp}^{\text{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ for $\mathcal{A}_{lor}$, and uses $\mathcal{A}_{lor}$'s guess b' to help him win $\mathbf{Exp}^{\text{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$.

Let $\mathcal{C}$ setup $\mathbf{Exp}^{\text{ftg-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$, initialise an empty array X_{ftg} and attribute set $A^\star$, and give oracle access, PP and 1^ρ to $\mathcal{A}_{ftg}$. $\mathcal{A}_{ftg}$ randomly chooses $i \xleftarrow{\$} \{1, \ldots, q\}$, and initialises two empty arrays, X_{lor} and Ch_{lor}. $\mathcal{A}_{ftg}$ also maintains a counter j initialised to 0. $\mathcal{A}_{ftg}$ sends PP and 1^ρ to $\mathcal{A}_{lor}$, and gives $\mathcal{A}_{lor}$ oracle access (to KeyGen and LOR). Every time $\mathcal{A}_{lor}$ makes a valid query to his LOR oracle, j is increased by 1.

$\mathcal{A}_{ftg}$'s strategy. Informally, the strategy of $\mathcal{A}_{ftg}$ is to return $\mathcal{ABE}.\mathsf{Encrypt}(m_0, A, PP)$ for the first $i - 1$ valid queries $\mathcal{A}_{lor}$ makes to his LOR oracle with inputs (m_0, m_1, A).

When $\mathcal{A}_{lor}$ makes his ith valid query to his LOR oracle, with inputs (m_0, m_1, A), $\mathcal{A}_{ftg}$ will forward this to $\mathcal{C}$ as their challenge in the FTG game, and will return their challenge response to $\mathcal{A}_{lor}$. For any further valid queries $\mathcal{A}_{lor}$ makes to the LOR oracle with inputs (m_0, m_1, A), $\mathcal{A}_{ftg}$ will return $\mathcal{ABE}.\mathsf{Encrypt}(m_1, A, PP)$. Then, $\mathcal{A}_{ftg}$ returns the output of $\mathcal{A}_{lor}$. $\mathcal{A}_{ftg}$ simulates the oracles for $\mathcal{A}_{lor}$ as follows:

KeyGen Oracle. Each time $\mathcal{A}_{lor}$ queries their KeyGen oracle with input $\mathbb{A}$, $\mathcal{A}_{ftg}$ will check that no attribute set $A \in Ch_{lor}$ satisfies $\mathbb{A}$ as specified. If this check fails, $\mathcal{A}_{ftg}$ returns $\perp$ to $\mathcal{A}_{lor}$. Else, $\mathcal{A}_{ftg}$ queries his challenger $\mathcal{C}$ for the key, returns his response to $\mathcal{A}_{lor}$ and adds $\mathbb{A}$ to X_{lor}. Note that the challenger $\mathcal{C}$ always returns a key, and not $\perp$ since:

- If $j < i$, then $A^\star$ has not been set and the challenger returns the key for $\mathbb{A}$.
- If $j \geqslant i$, then since $A^\star \in Ch_{lor}$, it must be that $A^\star \notin \mathbb{A}$ (else $\mathcal{A}_{ftg}$ would have returned $\perp$ to $\mathcal{A}_{lor}$). Thus $\mathcal{C}$ will return the key for $A^\star$.

Note that $\mathbb{A}$ is added to X_{ftg} by $\mathcal{C}$.

LOR Oracle. When $\mathcal{A}_{lor}$ makes his j^{th} valid query, for $0 < j \leqslant q$, to the LOR oracle with inputs $\{m_0, m_1, A\}$, A is added to Ch_{lor} and:

- if $j < i$, $\mathcal{A}_{ftg}$ returns $\mathcal{ABE}.\mathsf{Enc}(m_1, A, PP)$.
- if $j > i$, $\mathcal{A}_{ftg}$ returns $\mathcal{ABE}.\mathsf{Enc}(m_0, A, PP)$.
- if $j = i$, $\mathcal{A}_{ftg}$ forwards $\{m_0, m_1, A\}$ as his own challenge parameters to $\mathcal{C}$, and returns the response, y, to $\mathcal{A}_{lor}$.

We will define the two messages sent to $\mathcal{C}$ to be m_0^i, m_1^i. Since $\mathcal{A}_{ftg}$ only queries $\mathcal{C}$ for keys for access structures in X_{lor} (i.e. $X_{ftg} = X_{lor}$), $\{m_0, m_1, A\}$ are also valid challenge parameters in the FTG game (i.e. $A \notin \mathbb{A}$ for any $\mathbb{A} \in X_{ftg}$), and thus $\mathcal{C}$ will always return a valid ciphertext. $\mathcal{C}$ sets $A^\star \leftarrow A$. j is incremented after each valid query.

We calculate $\mathcal{A}_{ftg}$'s advantage using a standard hybrid argument. We define a sequence of $q + 1$ experiments: for $r = 0, \ldots, q$ define $\mathbf{Exp}^{\mathrm{hyb-cpa-r}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ to be an experiment in

which one answers the first r valid LOR oracle queries of $\mathcal{A}_{lor}$ (each with inputs of the form (m_0, m_1, A)) with $\mathcal{ABE}.\mathsf{Enc}(m_1, A, PP)$ and the rest with $\mathcal{ABE}.\mathsf{Enc}(m_0, A, PP)$. The output of the experiment is defined to be the output of $\mathcal{A}_{lor}$.

Note that:

$$\Pr\left[\mathbf{Exp}^{\text{hyb}-\text{cpa}-0}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right] = \Pr\left[\mathbf{Exp}^{\text{lor}-\text{cpa}-0}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right],$$

and

$$\Pr\left[\mathbf{Exp}^{\text{hyb}-\text{cpa}-\text{q}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right] = \Pr\left[\mathbf{Exp}^{\text{lor}-\text{cpa}-1}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right].$$

Now let us consider the experiment $\mathbf{Exp}^{\text{ftg}-\text{cpa}-\text{b}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$ where $\mathcal{A}_{ftg}$ is the adversary described above, using $\mathcal{A}_{lor}$ as a subroutine. Observe that when $b = 0$, the challenge ciphertext for $\mathcal{A}_{ftg}$ is $y = \mathcal{ABE}.\mathsf{Enc}(m_0^i, A, PP)$. Thus, the inputs to $\mathcal{A}_{lor}$ are identically distributed to those in $\mathbf{Exp}^{\text{hyb}-\text{cpa}-(\text{i}-1)}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$. Conversely, when $b = 1$, the challenge ciphertext is $y = \mathcal{ABE}.\mathsf{Enc}(m_1^i, A, PP)$ and the inputs to $\mathcal{A}_{lor}$ are identically distributed to those of $\mathbf{Exp}^{\text{hyb}-\text{cpa}-\text{i}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$. Thus we have that, for all $i \in \{1, \ldots, q\}$:

$$\Pr\left[\mathbf{Exp}^{\text{ftg}-\text{cpa}-1}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho) = 1 | r = i\right] = \Pr\left[\mathbf{Exp}^{\text{hyb}-\text{cpa}-\text{i}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right]$$

and

$$\Pr\left[\mathbf{Exp}^{\text{ftg}-\text{cpa}-0}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho) = 1 | r = i\right] = \Pr\left[\mathbf{Exp}^{\text{hyb}-\text{cpa}-(\text{i}-1)}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = 1\right].$$

Then

$$
\begin{aligned}
\mathrm{Adv}^{\mathrm{ftg-cpa}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho) &= \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-1}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-0}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1\right] \\
&= \Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-1}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1 | r=i\right] \cdot \Pr\left[r=i\right]\Big) - \\
&\quad \Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-0}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1 | r=i\right] \cdot \Pr\left[r=i\right]\Big) \\
&= \frac{1}{q}\Big(\Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-1}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1 | r=i\right]\Big) - \\
&\quad \Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{ftg-cpa-0}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)=1 | r=i\right]\Big)\Big) \\
&= \frac{1}{q}\Big(\Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{hyb-cpa-i}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right]\Big) - \\
&\quad \Big(\sum_{r=1}^{q} \Pr\left[\mathbf{Exp}^{\mathrm{hyb-cpa-(i-1)}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right]\Big)\Big) \\
&= \frac{1}{q}\left(\Pr\left[\mathbf{Exp}^{\mathrm{hyb-cpa-q}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{hyb-cpa-0}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right]\right) \\
&= \frac{1}{q}\left(\Pr\left[\mathbf{Exp}^{\mathrm{lor-cpa-1}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{lor-cpa-0}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)=1\right]\right) \\
&= \frac{1}{q} \cdot \mathrm{Adv}^{\mathrm{lor-cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho).
\end{aligned}
$$

□

Proof of Theorem 9. We show that if an adversary $\mathcal{A}_{CES}$ can break the security of our CES, then we can construct an adversary $\mathcal{A}_{lor}$ that, using $\mathcal{A}_{CES}$ as a subroutine, can break the (LOR) IND-CPA security of the large attribute universe KP-ABE scheme $\mathcal{ABE}$. Since the encryption scheme is assumed to be secure, such an adversary should not exist; therefore a successful adversary against the CES cannot exist.

Let $\mathcal{C}$ be the challenger for $\mathcal{A}_{lor}$ in experiment $\mathbf{Exp}^{\mathrm{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$. $\mathcal{A}_{lor}$ will act as the challenger for $\mathcal{A}_{CES}$ in the experiment $\mathbf{Exp}^{\mathrm{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P, D(O))$.

- $\mathcal{C}$ initialises $\mathbf{Exp}^{\mathrm{lor-cpa-b}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ for $\mathcal{A}_{lor}$ and gives $\mathcal{A}_{lor}$ 1^ρ, PP and KeyGen and LOR oracle access. As part of this, $\mathcal{C}$ initialises two empty lists: Ch_{lor} and X_{lor}.
- $\mathcal{A}_{lor}$ sets up $\mathbf{Exp}^{\mathrm{Ind-b}}_{\mathcal{CES},\mathcal{A}}(1^\rho, P = ((L, \leqslant), U, O, \lambda), D(O))$ for $\mathcal{A}_{CES}$ where P is arbitrarily chosen by $\mathcal{A}_{lor}$ such that O and L are non-empty.

- $\mathcal{A}_{lor}$ runs the experiment as described in Figure 6.2, but instead of running $\mathcal{ABE}.\mathsf{Setup}$ (as described in Figure 6.4) to generate MK and PP, he will use those generated by $\mathcal{C}$: $\mathcal{A}_{lor}$ is given PP as input and will use oracle access to $\mathcal{C}$ to simulate the use of MK when necessary. Furthermore, $\mathcal{A}_{lor}$ does not generate, nor request, the κ_M (requesting κ_M from $\mathcal{C}$ could prevent subsequent oracle calls since they could lead to trivial wins; instead, use of κ_M will again be simulated via oracle access as and when required). $\mathcal{A}_{lor}$ sets $\phi = (A, P)$.

  $\mathcal{A}_{lor}$ does not generate any keys in this game; instead, he will query $\mathcal{C}$ for keys required for users in Cr. Since Cr is empty at setup, $\mathcal{A}_{lor}$ need not define κ_u or st_u for any $u \in U$ here.

  $\mathcal{A}_{lor}$ encrypts $d(o)$ for each object $o \in O$ as stated (this is possible since encryption requires only public information) and outputs ψ and FS as Pub.

- $\mathcal{A}_{lor}$ gives $1^\rho, P, Pub$ and oracle access to $\mathcal{A}_{CES}$. We will describe how $\mathcal{A}_{lor}$ responds to each oracle query below. $\mathcal{A}_{lor}$ will store the (polynomially-sized) set of plaintext objects $D(O)$, such that he can use these as required without requesting the relevant decryption keys from $\mathcal{C}$ in order to 'decrypt' the ciphertexts stored in FS. Each time $\mathcal{A}_{lor}$ is required to decrypt and then encrypt an object (under a new key), he will simply find the relevant plaintext object and perform the relevant encryption. Of course, each time $\mathcal{A}_{CES}$ calls his WRITE oracle with valid inputs $(o, d(o)')$, $\mathcal{A}_{lor}$ updates his record of $d(o)$ and replaces it with $d(o)'$. $\mathcal{A}_{lor}$ will also store the last key sent to each user so that it can be re-sent in the event that the adversary calls CORRUPTU on a previously corrupted user.

- When $\mathcal{A}_{CES}$ chooses his challenge $(o^\star, d_0, d_1)$, $\mathcal{A}_{lor}$ proceeds as stated in the experiment but instead of performing Write, he will instead call his LOR oracle with inputs $(d_0, d_1, \lambda(o^\star)||A[\lambda(o^\star)])$ where $A[\lambda(o^\star)]$ is the current counter for $\lambda(o^\star)$. Since $\mathcal{A}_{lor}$ only makes KEYGEN oracle queries for keys for users in Cr, the access structures stored by $\mathcal{C}$ in X_{lor} are precisely those for which corrupted users hold corresponding decryption keys. Thus, checking whether $\lambda(o^\star) \leqslant \lambda(u)$ for any $u \in \mathsf{Cr}$ is equivalent to checking that there does not exist an $\mathbb{A} \in X_{lor}$ which $\lambda(o^\star)||A[\lambda(o^\star)]$ satisfies. Thus $\mathcal{C}$ will always return an encryption of d_b under $\lambda(o^\star)||A[\lambda(o^\star)]$ in response to this challenge query. $\mathcal{A}_{lor}$ will then set $\overline{d(o^\star)}$ to be his response from $\mathcal{C}$, and output an updated FS. $\mathcal{A}_{lor}$ also stores the challenge messages d_0, d_1 in case the challenge message for $o^\star$ needs re-encrypting (e.g. as a result of $\mathcal{A}_{CES}$ calling REFRESH on

$\lambda(o^\star)$). $\mathcal{A}_{lor}$ then proceeds as stated in the experiment. He forwards $\mathcal{A}_{CES}$'s eventual guess of b as his own to $\mathcal{C}$.

We will now show that $\mathcal{A}_{lor}$ can successfully respond to all valid oracle queries made by $\mathcal{A}_{CES}$.

ChObL Oracle. Whenever $\mathcal{A}_{CES}$ queries the ChObL oracle with inputs (o, l'), $\mathcal{A}_{lor}$ proceeds as stated in Figure 6.2. Note that if $\mathcal{A}_{CES}$ has already called WRITE on $o^\star$ then the challenge data d_b has been overwritten with some data $d(o)$. We have the following two cases to consider:

- **Case 1:** If $o \neq o^\star$ or if $\mathcal{A}_{CES}$ has already called WRITE on $o^\star$ then, instead of decrypting the ciphertext of o, $\mathcal{A}_{lor}$ simply finds the plaintext object $d(o)$ (which he has stored), encrypts it under attribute $l'||A[l']$ using the public paremeters and runs $\mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, Pub)$ as simulated by the REFRESH oracle, but without the oracle checks.

- **Case 2:** If $o = o^\star$ and $\mathcal{A}_{CES}$ has *not* previously called WRITE on $o^\star$, then $\mathcal{A}_{lor}$ submits a query to his LOR oracle with inputs $(d_0, d_1, l'||A[l'])$ (where d_0, d_1 were the challenge plaintexts chosen by $\mathcal{A}_{CES}$ and stored by $\mathcal{A}_{lor}$) and updates $\overline{d(o^\star)}$ to be the ciphertext returned by $\mathcal{C}$.

  We now show that $\mathcal{C}$ will return a valid ciphertext i.e. that $l'||A[l']$ does not satisfy some access structure $\mathbb{A} \in X_{lor}$. Since $\mathcal{A}_{lor}$ only makes queries to his KEYGEN oracle whenever he is prompted to generate new keys for users in Cr, it suffices to show that no user $u \in Cr$ is authorised for $l'||A[l']$.

  By the checks performed by $\mathcal{A}_{lor}$ during the first steps of the CHOBL oracle, there exists no $u \in Cr$ such that $l' \leqslant \lambda(u)$ (else $\mathcal{A}_{lor}$ would have returned $\perp$).

  Now, if a user was *previously* authorised for a label $l^\star \geqslant l'$ (but is no longer) then, at some point, the CHUSL oracle would have been called on inputs $(u, l^\star)$ (recall that all users are initially assigned to $\perp$ and so CHUSL oracle is required to assign a user to any other label). During that call to CHUSL, $\mathcal{A}_{lor}$ would have queried its KEYGEN oracle for the key associated to $\bigvee_{l \leqslant l^\star} l||A[l]$ (noting that l' is one such label since $l' \leqslant l^\star$) and this access structure would have been added to X_{lor} by $\mathcal{C}$.

Let $i = A[l']$ be the counter for l' when this key was queried. Now, since the user is no longer authorised for l', ChUsL must have been subsequently called to assign u some other label $l \not\geq l'$. During this function call, the counter associated to l' would have been incremented by one and thus the current counter $A[l']$ must be some $j > i$. Thus, the current attribute $l'||A[l']$ queried to the LOR oracle does *not* satisfy any access structure $\mathbb{A} \in X_{lor}$ and $\mathcal{C}$ will return a valid ciphertext.

$\mathcal{A}_{lor}$ completes by running $\mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, Pub)$ described as below for the Refresh oracle, but without the oracle checks.

ChUsL Oracle. $\mathcal{A}_{lor}$ proceeds as stated in the ChUsL oracle in Figure 6.2. When it comes to decrypting and re-encrypting objects o such that $\lambda(o) \in X$ (where X is as defined in our construction in Figure 6.4), we have the following two cases:

- **Case 1:** If $o \neq o^\star$, or if $o = o^\star$ and $\mathcal{A}_{CES}$ has already called Write on $o^\star$ (i.e. the challenge data d_b has been overwritten) then, instead of decrypting and re-encrypting $d(o)$, $\mathcal{A}_{lor}$ simply retrieves the plaintext object $d(o)$ (which he has stored) and encrypts this using PP and its (new) attribute $\lambda(o)||A[\lambda(o)]$.
- **Case 2:** If $o = o^\star$ and $\mathcal{A}_{CES}$ has *not* called Write on $o^\star$, then $\mathcal{A}_{lor}$ queries their LOR oracle with inputs $(d_0, d_1, \lambda(o^\star)||A[\lambda(o^\star)])$ (where $A[\lambda(o^\star)]$ has been incremented).

  Note that all prior calls to KeyGen for a key related to label $\lambda(o^\star)$ will be associated to an access structure containing an attribute $\lambda(o^\star)||x$ where $x < A[\lambda(o^\star)]$ (where $A[\lambda(o^\star)]$ is the current counter for $\lambda(o^\star)$). Since $\lambda(o^\star)||A[\lambda(o^\star)]$ is a 'new' attribute (in particular no KeyGen query has been made which involves this attribute), $\mathcal{A}_{lor}$ has not yet made a query to their KeyGen oracle for a key whose access structure is satisfied by this attribute, and hence $\mathcal{C}$ will respond with a valid encryption of d_b.

$\mathcal{A}_{lor}$ must also generate new keys to reflect the updated attribute counters. Note that keys are only distributed to corrupted users, and so new user states are only generated for users $u' \in \mathsf{Cr}$ who are authorised for a label which belongs to X. In particular, $\mathcal{A}_{lor}$ need not generate a new key κ_M.

For each such user u', $\mathcal{A}_{lor}$ queries his KeyGen oracle for the key associated to the access

structure:

$$\bigvee_{x \leqslant \lambda(u')} x||A[x] = \bigvee_{y \leqslant \lambda(u'), y \notin X} y||A[y] \vee \bigvee_{z \leqslant \lambda(u'), z \in X} z||A[z] \tag{6.1}$$

Where $A[z]$ is the new counter for $z \in X$. Note that each user $u^\star \in Cr$ has previously been issued the key for an access structure

$$\bigvee_{y \leqslant \lambda(u^\star), y \notin X} y||A[y] \vee \bigvee_{t \leqslant \lambda(u^\star), t \in X} t||(A[t]-1). \tag{6.2}$$

Since $\mathcal{A}_{lor}$ must have successfully queried $\mathcal{C}$ for each such key, it must be that (by the check performed by the KeyGen oracle) no $A \in Ch_{lor}$ satisfied the access structure in 6.2. Thus, it remains to show that, for each $A \in Ch_{lor}$ and $u^\star \in Cr$, $A \notin \bigvee_{z \leqslant \lambda(u^\star), z \in X} z||A[z]$. Since all prior calls to the LOR oracle for an access structure associated to a label $z \in X$ contain some attribute $z||i$ where $i < A[z]$, no attribute set $A \in Ch_{lor}$ is equal to $z||A[z]$ and thus the KeyGen query is valid and hence $\mathcal{C}$ returns the required keys.

Refresh Oracle. When $\mathcal{A}_{CES}$ queries the Refresh oracle, $\mathcal{A}_{lor}$ proceeds as described but does not generate a new manager key κ'_M, and only generates new states for corrupted users. For each user $u \in Cr$ where $l \leqslant \lambda(u)$, the access structure associated to their decryption key gets updated from

$$l||A[l] \vee \bigvee_{l^\star \leqslant \lambda(u), l' \neq l} l'||A[l']$$

to

$$l||(A[l]+1) \vee \bigvee_{l' \leqslant \lambda(u), l' \neq l} l'||A[l']$$

where $A[l]+1$ is the new counter for l. To generate such a key, $\mathcal{A}_{lor}$ makes a KeyGen query to $\mathcal{C}$ for the updated access structure.

Since $l||(A[l]+1)$ is a new attribute, all prior LOR queries associated to label l are for some attribute $l||i$ where $i < A[l]+1$ and thus $l||(A[l]+1) \notin Ch_{lor}$. Furthermore, there exists no attribute $A \in Ch_{lor}$ that satisfies $l||A[l] \vee \bigvee_{l' \leqslant \lambda(u), l' \neq l} l'||A[l']$ for any $u \in Cr$, since that would mean a previous oracle call led to a trivial win. Then, no attribute $A \in Ch_{lor}$ satisfies $l||(A[l]+1) \vee \bigvee_{l' \leqslant \lambda(u), l' \neq l} l'||A[l']$ for all $u \in Cr$ and thus it follows that

KeyGen returns valid decryption keys for all such queries.

For every $o \in O$ where $\lambda(o) = l$:

- if $o \neq o^\star$ or if $\mathcal{A}_{CES}$ has already called Write on $o^\star$, $\mathcal{A}_{lor}$ retrieves the plaintext object $d(o)$ that it has stored, encrypts it using PP under its new attribute $l||A[l]$ and updates FS.
- If $o = o^\star$ and $\mathcal{A}_{CES}$ has *not* called Write on $o^\star$, then $\mathcal{A}_{lor}$ queries his LOR oracle with inputs $(d_0, d_1, l||A[l])$ (where $A[l]$ has been incremented) and updates $\overline{d(o)}$ to be the returned output. Since $l||A[l]$ is a new attribute, all prior calls to the KeyGen oracle for a key associated to label l contain some attribute $l||i$ where $i < A[l]$. Thus it follows that $l||A[l]$ does not satisfy any access structure $\mathbb{A} \in X_{lor}$ and thus LOR returns a valid ciphertext.

$\mathcal{A}_{lor}$ then proceeds as stated.

Write Oracle. If $\mathcal{A}_{CES}$ calls the Write oracle with inputs $(o, d(o)')$, $\mathcal{A}_{lor}$ replaces the plaintext data he stores for object o with $d(o)'$, encrypts it under PP and the attribute $\lambda(o)||A[\lambda(o)]$, and updates the file system and Pub as described.

CorruptU Oracle. Whenever $\mathcal{A}_{CES}$ calls the CorruptU oracle with input u where $\lambda(u) \not\geqslant \lambda(o^\star)$, $\mathcal{A}_{lor}$ adds u to Cr. If u was already in Cr, $\mathcal{A}_{lor}$ returns $st_u = k_u$, the key last sent to u (either from the CorruptU oracle or another oracle).

If u was not previously in Cr, $\mathcal{A}_{lor}$ must recreate a valid state for u (since states for non-corrupted users have not been generated during the prior oracle calls). $\mathcal{A}_{lor}$ calls his KeyGen oracle with input $\bigvee_{l \leqslant \lambda(u)} l||A[l]$ and returns the output, along with $\lambda(u)$, as the output of CorruptU oracle. Then, since $\lambda(o^\star) \not\leqslant \lambda(u)$, we know that $\lambda(o^\star)||A[\lambda(o^\star)]$ does not satisfy u's access structure. If $o^\star$ was previously associated to some label $x \leqslant \lambda(u)$, then when ChObL was last called on $o^\star$ to change its label to some label $l^\star \not\leqslant \lambda(u)$, the counter/attribute associated to x would have been incremented by one by ChObL. Thus no attribute set $A \in Ch_{lor}$ satisfies $\bigvee_{l \leqslant \lambda(u)} l||A[l]$ and thus $\mathcal{C}$ returns a valid decryption key, which $\mathcal{A}_{lor}$ forwards to $\mathcal{A}_{CES}$.

Once $\mathcal{A}_{CES}$ has finished making oracle queries and submits his guess b' of b, $\mathcal{A}_{lor}$ forwards b' to $\mathcal{C}$ and thus uses $\mathcal{A}_{CES}$'s guess of b as his own. Then, it follows that whenever $\mathcal{A}_{CES}$ wins, $\mathcal{A}_{lor}$ wins.

We define the advantage $\text{Adv}^{\text{lor}-\text{cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho)$ of $\mathcal{A}_{lor}$ in the above experiment to be:

$$\text{Adv}^{\text{lor}-\text{cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) = \text{Adv}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}_{CES}}(1^\rho, P, D(O))$$

where $\text{Adv}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}_{CES}}(1^\rho, P, D(O))$ is the non-negligible advantage of $\mathcal{A}_{CES}$.

Since the advantage of an adversary in an IND-CPA experiment for a KP-ABE scheme is typically defined in terms of the advantage of an adversary in the FTG version of the experiment, by Lemma 10, it follows that:

$$\begin{aligned}\text{Adv}^{\text{ftg}-\text{cpa}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho) &= \frac{1}{q}\text{Adv}^{\text{lor}-\text{cpa}}_{\mathcal{ABE},\mathcal{A}_{lor}}(1^\rho) \\ &= \frac{1}{q}\text{Adv}^{\text{Ind-b}}_{\mathcal{CES},\mathcal{A}_{CES}}(1^\rho, P, D(O))\end{aligned}$$

where $q = poly(1^\rho)$.Thus, it follows that $\text{Adv}^{\text{ftg}-\text{cpa}}_{\mathcal{ABE},\mathcal{A}_{ftg}}(1^\rho)$ is negligible.

Since we made the assumption that $\mathcal{ABE}$ is IND-CPA secure, no such adversary $\mathcal{A}_{CES}$ can exist and thus the instantiation in Figure 6.4 is a secure dynamic, centralised, refreshable, writeable CES. □

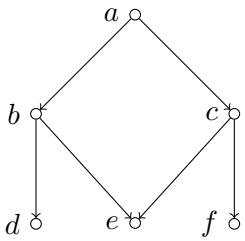
Again, by considering cryptographic primitives within our framework, it becomes apparent that some existing proposals for enforcement mechanisms for access control are not entirely sufficient. For example, whilst there are many works considering revocation within ABE [10, 75], it seems more difficult to reduce access rights rather than remove the user completely without assigning an entirely new user identifier.

6.5 Comparison To Prior Frameworks

The cryptographic framework proposed by Ferrara *et al.* [45] focuses on the enforcement of non-hierarchical RBAC policies, whereas the framework we have proposed in this chapter considers the enforcement of information flow policies which are hierarchical in nature.

Although Ferrara *et al.* state that their framework can be used to construct a CES to enforce hierarchical RBAC policies, the hierarchical RBAC policy must first be translated into a core RBAC (non-hierarchical) policy, which may lead to users being assigned to many roles (i.e. each role to which they would have previously inherited the access rights of), or could lead to several copies of each object being stored on the file system (e.g. for every role r that inherits read access to the object in the hierarchical policy, an encryption of the object under the encryption key for r may need to be available). Similarly, if one represents an information flow policy as a core RBAC policy then additional user and/or object assignments would be required, potentially leading to users requiring additional cryptographic material and/or additional public information (either in terms of encryptions of objects under multiple keys or additional information required to support key derivation). Figure 6.6a shows an example information flow policy, whose user-object label assignment in Figure 6.6b. Figure 6.7a shows a core RBAC representation of the information flow policy policy, where each arc represents a user/object label assignment. Note that in the core RBAC policy, each user and object is assigned to each security label for which they are authorised (i.e. each security label whose access rights they would have inherited in the information flow policy shown in Figure 6.6b). Thus, whilst user u_1 is only assigned to label b in the information flow polcy, u_1 is now assigned to three labels: b, d and e in the associated core RBAC representation of the policy.

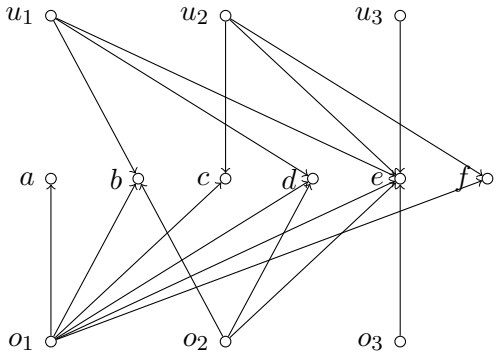
Thus a more efficient CES may be gained by considering a framework for the cryptographic enforcement of information flow policies *directly*, rather than trying to first translate these policies into core RBAC policies. Additionally, it may be difficult to maintain some of the original hierarchical structure when trying to make changes to the policy. For example, consider the policy in Figure 6.6a and suppose we want that user u_2 should be moved from security label c to security label f. Now, in the information flow policy, this would mean that u_2 should not be authorised for object o_3 assigned to label e, since f does not inherit the access rights of security label e. To make the same change in the associated core RBAC policy, one would have to manually identify all such roles or labels to which a user should no longer be authorised in order to make the resulting change. In this example, u_2 would thus need to be deassigned from role e as well as role c. Thus, it may be considered more difficult to enforce a hierarchical policy represented as a core RBAC policy, particularly if one wants to maintain some sort of hierarchical access structure when making policy changes.



(a) Example information flow policy where λ is shown in Figure 6.6b.

user/object x	$\lambda(x)$
u_1	b
u_2	c
u_3	e
o_1	a
o_2	b
o_3	e

(b) User-object label assignment for Figure 6.6a.

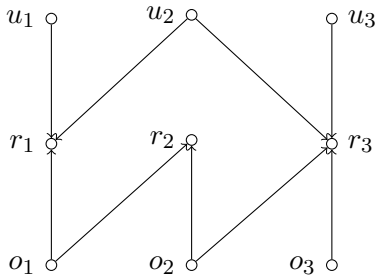


(c) Example Core RBAC policy representation of information flow policy in Figure 6.6a.

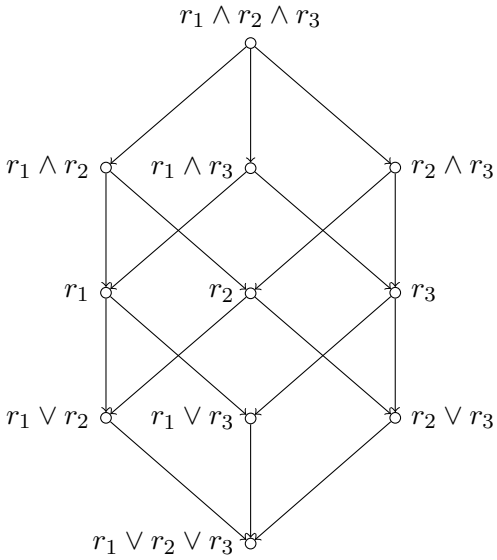
Figure 6.6: Example Information flow policy and Core RBAC representation.

Alternatively, one could consider representing a core or hierarchical RBAC policy as an information flow policy [30]. Figure 6.7 shows an example core RBAC policy with three roles $\{r_1, r_2, r_3\}$ and the information flow policy representation of that policy [30]. Then, one could use the framework proposed in this chapter to construct a CES to enforce (core or hierarchical) RBAC policies. The disadvantage in converting an RBAC policy into an information flow policy is that the resulting information flow policy may consist of a large number of security labels ($\mathcal{O}(2^n)$, where n is the number of roles in the original core RBAC policy), each of which may require its own cryptographic material (both public and secret). For example, considering Figure 6.7, the core RBAC policy contains 3 roles, and the information flow policy poset contains $2^3 = 8$ security labels[5]. Thus, one could argue that converting an information flow policy into an RBAC policy, or vice versa, may reduce the efficiency of the resulting CES used to enforce it. Thus, both the framework

[5]Although one could argue that if the policy is static, then only the labels to which a user/object is assigned need be defined. However, for dynamic policies, it may be necessary to define all such labels.



(a) Example Core RBAC policy.



(b) Information flow policy representation [30] of the Core RBAC policy shown in Figure 6.7a where λ is shown in Figure 6.7c.

user/object x	$\lambda(x)$
u_1	r_1
u_2	$r_1 \wedge r_3$
u_3	r_3
o_1	$r_1 \vee r_2$
o_2	$r_2 \vee r_3$
o_3	r_3

(c) User-object label assignment for Figure 6.7b.

Figure 6.7: Example Core RBAC policy and an information flow policy representation of the RBAC policy.

defined in this chapter *and* that defined by Ferrara *et al.* for RBAC policies have their own advantages for constructing CESs for their associated policies.

Additionally, whilst the correctness and security notions for such frameworks are very similar, the internal checks that occur within the experiments are influenced by the type of policy for which they are designed. For example, to corrupt a user in the security experiment for the RBAC framework, one has to check that the user u is not authorised for a role to which the challenge object $o^\star$ is assigned, whereas in the security experiment

for our framework one has to check that the user is not authorised for any security label that inherits access permissions to the challenge object $o^\star$, i.e. $\lambda(u) \not\geqslant \lambda(o^\star)$.

One could also argue that in comparison to the framework proposed by Ferrara *et al.*, our framework for information flow policies has additional efficiency benefits. For example, within our framework, one could move a user u from a security label l to *any* other security label l' by a *single* call to ChUsL with inputs u and l'. To perform a similar action (move a user u assigned to a set of roles R to a set of roles R') in the framework proposed by Ferrara *et al.* [45] one would have to call DeassignUser to deassign u from each role in $R \setminus R'$ (one algorithm call per role in this set), and then call AssignUser to assign u to each role in the set $R' \setminus R$ (again, one algorithm call per role in this set). For example, if a user u_1 was to be assigned roles r_1, r_2, r_3, then in the predicate encryption instantiation that Ferrara *et al.* provide, we would have to call AssignUser for each role in the set $\{r_1, r_2, r_3\}$, with each such call producing an updated decryption key for user u_1, which may be expensive to generate. Instead, it may have been more efficient to produce a single decryption key for u_1 that authorises u to access objects assigned to roles r_1, r_2 and r_3. Similarly, our framework enables an initial policy state to be specified and enacted during setup whereas the cryptographic RBAC framework by Ferrara *et al.* initialises a trivial state and iteratively creates the initial policy.

By grouping together multiple small changes within a single algorithm call, one may avoid repeated work that may occur as a result of performing such changes one after the other, potentially resulting in a more efficient implementation. In addition, such an algorithm could be considered easier for the manager to run. If the manager wants to move a user u from one set of roles to another using the CES framework proposed by Ferrara *et al.* [45], then the manager would have to think more carefully as to which roles he should assign/deassign the user from, and the order in which he should do so, in order to result in the same policy change (and to also perhaps minimise overhead).

The disadvantage of enabling the manager to dynamically change the security label assigned to an object or user using just one algorithm call (instead of incrementally revoking and/or permitting access) is that one must be more careful when designing the framework to ensure that such changes maintain security and correctness. For example, in the policy shown in Figure 6.6, to change a user's security label from $b \wedge c$ to a via ChUsL, we have to identify *all* security labels for which the user is no longer authorised for (in this case,

labels b, d, f) — it is not just a case of simply updating label f. In addition, depending on the cryptographic primitives used (e.g. consider an iterative KAS described in Section 2.5.3) where keys for labels are derived from the keys/secrets for other labels), it may not be sufficient to 'just' refresh the cryptographic material associated to the compromised labels. This is because it may no longer be possible for users, authorised for some label l, whose cryptographic material has been refreshed, to derive the current secrets and keys associated to labels $l' \leqslant l$ which have not been updated. Thus, to maintain correctness, cryptographic material for non-corrupted labels (i.e. labels whose keys should be derivable from the updated cryptographic material) may also need updating. In some sense, therefore, our framework shifts the burden of making such decisions and ensuring that correctness and security hold at all times from the manager (during the execution of the system in a live deployment) to the design of the framework (against which a specific implementation should be tested prior to deployment). Hence, a framework for information flow policies directly (as opposed to converting the policy to a core RBAC policy first) may improve the efficiency of implementations and may improve their safety by reducing the burden of the managing entity.

6.6 Conclusion

We have developed a rigorous definitional framework for the cryptographic enforcement of information flow policies. Our framework has been developed 'bottom up' from the requirements of the access control policy, rather than targeting a particular cryptographic primitive or application scenario. We have provided several example classes of CES and discussed the algorithmic requirements of each, and provided a formal notion of correctness and security. Finally we have provided two instantiations, based on very different primitives (e.g one is based on symmetric cryptography, and the other, on public key cryptography), to exemplify the utility of our framework. By providing an instantiation based on a key assignment scheme, we were able to highlight the current limitations of a KAS, and how it restricts what classes of CES one may construct under its current definition.

Chapter 7

Conclusion

In this chapter, we summarise the contribution of this thesis and discuss ideas for future work.

In this thesis, we have studied the cryptographic enforcement of read-only information flow policies. We have identified that, when one is trying to reduce the size of system parameters, such as ciphertext sizes, key sizes and computational costs, the use of symmetric primitives may be preferred over public key primitives. We thus identified key assignment schemes as useful mechanisms for reducing the amount of key material each user requires. Prior key assignment schemes typically required additional public information in order to support the derivation of keys by users, which may be large and expensive to maintain. Hence this motivated us to consider key assignment schemes which do not require public derivation information. In Chapters 3-5, we thus proposed three different key assignment schemes which do not require public derivation information, each with varying characteristics. A nice feature of all our proposed KASs is that they meet the strongest security notion for KASs, and can all be constructed in polynomial time.

In Chapter 3, we described a key assignment scheme based on chain partitions which enables users to iteratively derive intermediate secrets and keys down chains. The existing literature on chain-based schemes did not consider how to best partition a given policy poset into chains. We provided an efficient polynomial time algorithm for partitioning any policy poset into chains, as to minimise the number of intermediate secrets required by each user and in total. Informally, we identified that the number of chains in a given partition

was an upper bound on the number of intermediate secrets required by any user and, by Dilworth's theorem [42], the minimal number of chains in any chain partition is w, where w is the width of the underlying poset. By careful construction and transformation of the problem of minimising the number of intermediate secrets into a network flow problem, we were able to construct a chain partition of w chains of any given (policy) poset which minimises the total number of intermediate secrets required to be distributed to the user population in the corresponding chain-based KAS.

In Chapter 4, we identified that one could work with trees instead of chains, whilst still enabling users to iteratively derive keys down paths in a 'tree representation' of the policy poset without requiring public derivation information to assist key derivation. We thus proposed a new key assignment scheme which represents the policy poset as an out-tree and provided an efficient polynomial time algorithm for finding an optimal tree partition of the poset which minimises the total number of intermediate secrets required by the user population (at setup). Unfortunately, unlike the chain-based KAS described in Chapter 3, it is not possible to always find a tree partition that both minimises the number of intermediate secrets required by each user and in total. We did, however, identify that the number of leaves in the resulting tree partition was an upper bound on the number of intermediate secrets required by any user. Thus, we proposed a 'tweak' to our algorithm which finds a tree partition that minimises the total number of intermediate secrets required for the entire user population, such that if there is more than one choice of tree with minimal number of intermediate secrets, then it finds one with a minimal number of leaves.

Finally, we proposed a third KAS in Chapter 5 based on binary trees. Such a KAS is rather different to existing schemes in the literature in that the policy poset is mapped to a binary tree structure instead of (a subgraph of) the transitive closure of the Hasse diagram of the policy poset. By defining key derivation along paths in the binary tree, key derivation is logarithmically bounded, unlike in the tree and chain-based schemes described in Chapters 3 and 4 respectively. Additionally, we showed that the use of a binary tree structure enables us to eliminate *all* public information (other than the public filesystem), i.e. the policy poset need not be public. Unfortunately, users may require a substantial number of intermediate secrets in our binary tree KAS. We thus developed several heuristics to minimise the average number of intermediate secrets required by each user.

In Chapter 5, we also provided some experimental results to compare how both the KASs proposed in this thesis, and existing KASs in the literature (for example, the Iterative scheme proposed by Atallah *et al.* [6]) performed in practice. Such experiments suggested that, for the policies considered in the experiments, the average number of intermediate secrets required by each user in the binary tree KAS is typically much less than the theoretical upper bound of $\lceil \frac{n}{2} \rceil$ (where n is the number of security labels in the policy). Furthermore, the results suggested that the binary tree KAS performed better than the chain-based scheme in terms of average and maximum number of key derivations required by any user, and in terms of the average number of intermediate secrets required by any user. Of course, the chain-based KAS has the nice guarantee that no user will require more than w intermediate secrets, where w is the width of the policy poset, compared to the large upper bound of $\lceil \frac{n}{2} \rceil$ for binary tree KAS. Thus when user storage is a priority, the chain-based KAS may still be preferred over binary tree KAS. The experiments also showed that the tree-based KAS proposed in Chapter 4 performed best (compared to the chain and binary tree KAS) in terms of the average number of intermediate secrets required by any user.

In Chapter 6 we provided a concrete security framework for the cryptographic enforcement of read-only information flow policies that is independent of any cryptographic primitive that may be used to instantiate a CES. We provided several classes of CES and gave formal definitions of security and correctness for a CES for read-only information flow policies. As suggested by Ferrara *et al.* [45], there is often a gap between the theoretical policy specification and a cryptographic implementation of an enforcement mechanism, and thus this framework acts as a template for designing a CES for read-only information flow policies that is correct and secure.

We gave two instantiations, one based on symmetric primitives and the other on public key primitives, both of which we proved secure against our security framework. The first instantiation used symmetric primitives, and combined an IND-CPA secure symmetric encryption scheme with a KAS which is secure in the sense of Key Indistinguishability. Whilst such an instantiation is secure, the perhaps somewhat limited functionality of a KAS restricts the classes of CES that one may construct when solely combining a KAS with a secure encryption scheme. For example, since a 'refresh' algorithm is not defined within a KAS[1], one cannot simply construct a *refreshable* CES using a KAS and

[1]Some of the literature (e.g. [6]) discusses how one might handle updates in a KAS, however these are

secure encryption scheme alone. Whilst KASs may be adequate for constructing basic CESs, we hope that future work considers expanding the current definition of a KAS to include more functionality, such that it can be used to enforce more expressive classes of CES (e.g. refreshable CESs). It would also be interesting to implement various CES constructions using different primitives to see how they compare in practice. Future work should also consider expanding our CES framework to incorporate write access information flow policies.

We hope that future work also continues to develop KASs that do not require public information. As mentioned briefly in Chapter 4, unlike Chain-based schemes, it did not seem possible to always minimise the number of intermediate secrets, both in total and on a per-user basis for tree-based KASs. From a practical perspective, it would thus be interesting to find an algorithm that can compute an out-tree such that: (i) no user requires more than w intermediate secrets (or keys), where w is the width of the poset, and (ii) the total number of intermediate secrets is as small as possible. In addition, one parameter that our chain and tree-based KASs did not address is the number of derivation steps d required by a user in the worst case. If we compare the chain partitions in Figure 3.2c, for example, we see that a user assigned to the maximum vertex (label) will need to perform at most 3 derivations. Moreover, for any other chain partition of cardinality 2 of the same poset, there must be a chain of depth 4. Thus, in this case, the chain partition in Figure 3.2c is optimal. We thus hope that future work considers attempting to find an efficient algorithm that takes an information flow policy as input and outputs a chain or tree partition which also minimises d.

As with regards to the binary tree KAS described in Section 5, we hope that future work will also consider other enforcement structures to target different design goals of KASs and develop interesting optimisation strategies for the mappings. For example, one could generalise our construction based on binary trees to n-ary trees or trees with varying degrees. Finally, we hope that the KAS described in this chapter spurs the development of efficient constrained PRFs (i.e. ones based on symmetric primitives) tailored to enforcing access control policies.

Whilst we ran experiments to compare the performance of the KASs proposed in this thesis for randomly generated posets/policies, it would also be interesting to consider how

not captured within the current definition of a KAS.

they compare for real-life policies; unfortunately we were unable to obtain such policies and thus one might argue that our experimental comparisons are inconclusive. Thus we hope that experimental analysis of KASs and CESs for information flow policies is further explored in future work.

Bibliography

[1] M. Abadi and B. Warinschi. Security analysis of cryptographically controlled access to XML documents. *Journal of the ACM (JACM)*, 55(2):6, 2008.

[2] M. Abdalla, C. Cid, B. Gierlichs, A. Hlsing, A. Luykx, K. G. Paterson, B. Preneel, A.-R. Sadeghi, T. Spies, M. Stam, M. Ward, B. Warinschi, and G. Watson. Algorithms, key size and protocols report. Technical report, ECRYPT CSA, 01 2015.

[3] 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.

[4] J. Alderman and J. Crampton. On the Use of Key Assignment Schemes in Authentication Protocols. In J. Lopez, X. Huang, and R. Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 607–613. Springer Berlin Heidelberg, 2013.

[5] J. Alderman, N. Farley, and J. Crampton. Tree-based cryptographic access control. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 47–64, 2017.

[6] 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.

[7] 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.

[8] M. J. Atallah, M. Blanton, and K. B. Frikken. Incorporating temporal capabilities in existing key management schemes. In J. Biskup and J. Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2007.

[9] 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.

[10] N. Attrapadung and H. Imai. Attribute-based encryption supporting direct/indirect revocation modes. In M. G. Parker, editor, *IMA Int. Conf.*, volume 5921 of *Lecture Notes in Computer Science*, pages 278–300. Springer, 2009.

[11] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, volume 4189 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 2006.

[12] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2nd edition, 2009.

[13] A. Beimel. Secure schemes for secret sharing and key distribution. *PhD thesis, Israel Institute of Technology, Technion*, 1996.

[14] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corporation, 1973.

[15] D. Bell and L. LaPadula. Computer security model: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., 1975.

[16] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 394–403. IEEE Computer Society, 1997.

[17] M. Bellare and P. Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 207, 2005.

[18] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE Symposium on Security and Privacy*, pages 321–334. IEEE Computer Society, 2007.

[19] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., 1977.

[20] M. Bishop. *Introduction to Computer Security*. Addison-Wesley, 2005.

[21] 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.

[22] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In Y. Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[23] D. Boneh and B. Waters. Constrained pseudorandom functions and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 280–300. Springer, 2013.

[24] E. Boyle, S. Goldwasser, and I. Ivan. Functional signatures and pseudorandom functions. In *International Workshop on Public Key Cryptography*, pages 501–519. Springer, 2014.

[25] A. Castiglione, A. D. Santis, and B. Masucci. Key indistinguishability versus strong key indistinguishability for hierarchical key assignment schemes. *IEEE Trans. Dependable Sec. Comput.*, 13(4):451–460, 2016.

[26] 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.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[28] 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.

[29] J. Crampton. Trade-offs in cryptographic implementations of temporal access control. In A. Jøsang, T. Maseng, and S. J. Knapskog, editors, *Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems, NordSec 2009, Oslo, Norway, 14-16 October 2009. Proceedings*, volume 5838 of *Lecture Notes in Computer Science*, pages 72–87. Springer, 2009.

[30] 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.

[31] J. Crampton. Practical and efficient cryptographic enforcement of interval-based access control policies. *ACM Trans. Inf. Syst. Secur.*, 14(1):14, 2011.

[32] 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.

[33] 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.

[34] 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.

[35] J. Crampton, N. Farley, G. Gutin, M. Jones, and B. Poettering. Cryptographic enforcement of information flow policies without public information via tree partitions. *Journal of Computer Security*, 25(6):511–535, 2017.

[36] 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.

[37] I. Damgård, H. Haagh, and C. Orlandi. Access control encryption: Enforcing information flow with cryptography. In M. Hirt and A. D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 547–576, 2016.

[38] P. D'Arco, A. De Santis, A. L. Ferrara, and B. Masucci. Security and tradeoffs of the akl-taylor scheme and its variants. In R. Královic and D. Niwinski, editors, *MFCS*, volume 5734 of *Lecture Notes in Computer Science*, pages 247–257. Springer, 2009.

[39] P. D'Arco, A. D. Santis, A. L. Ferrara, and B. Masucci. Variations on a theme by akl and taylor: Security and tradeoffs. *Theor. Comput. Sci.*, 411(1):213–227, 2010.

[40] A. De Santis, A. L. Ferrara, and B. Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. *Theor. Comput. Sci.*, 407(1-3):213–230, 2008.

[41] S. D. C. Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *Proceedings of the 33rd international conference on Very Large Data Bases*, pages 123–134. VLDB endowment, 2007.

[42] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 1(50):161–166, 1950.

[43] D. Ferraiolo and R. Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[44] A. L. Ferrara, G. Fuchsbauer, B. Liu, and B. Warinschi. Policy privacy in cryptographic access control. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 46–60. IEEE Computer Society, 2015.

[45] A. L. Ferrara, G. Fuchsbauer, and B. Warinschi. Cryptographically enforced RBAC. In *CSF*, pages 115–129. IEEE, 2013.

[46] A. L. Ferrara and B. Masucci. An information-theoretic approach to the access control problem. In C. Blundo and C. Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 342–354. Springer, 2003.

[47] E. S. V. Freire and K. G. Paterson. Provably secure key assignment schemes from factoring. In U. Parampalli and P. Hawkes, editors, *Information Security and Privacy - 16th Australasian Conference, ACISP 2011, Melbourne, Australia, July 11-13, 2011. Proceedings*, volume 6812 of *Lecture Notes in Computer Science*, pages 292–309. Springer, 2011.

[48] 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.

[49] K. Fu, S. Kamara, and Y. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proceedings of the Network and Distributed System*

Security Symposium, NDSS 2006, San Diego, California, USA. The Internet Society, 2006.

[50] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, Mar. 1986.

[51] T. Gallai and A. N. Milgram. Verallgemeinerung eines Graphentheoretischen Satzes von Rédei. *Acta Sci. Math.*, 21:181–186, 1960.

[52] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.

[53] V. K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

[54] W. C. Garrison III, A. Shull, S. Myers, and A. J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 819–838. IEEE Computer Society, 2016.

[55] D. K. Gifford. Cryptographic sealing for information secrecy and authentication. *Communications of the ACM*, 25(4):274–286, 1982.

[56] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[57] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[58] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 89–98. ACM, 2006.

[59] E. Gudes. The design of a cryptography based secure file system. *IEEE Trans. Softw. Eng.*, 6(5):411–420, Sept. 1980.

[60] G. Gutin, I. Razgon, and E. J. Kim. Minimum leaf out-branching and related problems. *Theor. Comput. Sci.*, 410(45):4571–4579, 2009.

[61] S. Halevi, P. A. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. *IACR Cryptology ePrint Archive*, 2005:169, 2005.

[62] A. Harrington and C. Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, pages 158–165. ACM, 2003.

[63] S. Hohenberger, V. Koppula, and B. Waters. Adaptively secure puncturable pseudorandom functions in the standard model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 79–102. Springer, 2015.

[64] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations. Technical Report NIST Special Publication 800-162, NIST, 2014.

[65] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC Press, 2007.

[66] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In N. P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2008.

[67] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 669–684. ACM, 2013.

[68] F. Kuo, V. Shen, T. Chen, and F. Lai. Cryptographic key assignment scheme for dynamic access control in a user hierarchy. *Computers and Digital Techniques, IEE Proceedings* -, 146(5):235 –240, sep 1999.

[69] J. Lai, R. H. Deng, Y. Li, and J. Weng. Fully secure key-policy attribute-based encryption with constant-size ciphertexts and fast decryption. In S. Moriai, T. Jaeger, and K. Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 239–248. ACM, 2014.

[70] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 62–91. Springer, 2010.

[71] A. B. Lewko and B. Waters. Unbounded HIBE and attribute-based encryption. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 547–567. Springer, 2011.

[72] B. Liu and B. Warinschi. Universally composable cryptographic role-based access control. Cryptology ePrint Archive, Report 2016/902, 2016. `http://eprint.iacr.org/2016/902`.

[73] H. K. Maji, M. Prabhakaran, and M. Rosulek. Attribute-based signatures. In A. Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2011.

[74] R. Ostrovsky, A. Sahai, and B. Waters. Attribute-based encryption with non-monotonic access structures. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 195–203. ACM, 2007.

[75] J.-l. Qian and X.-l. Dong. Fully secure revocable attribute-based encryption. *Journal of Shanghai Jiaotong University (Science)*, 16:490–496, 2011.

[76] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[77] A. Sahai and B. Waters. Fuzzy identity-based encryption. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, 2005.

[78] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[79] R. S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Inf. Process. Lett.*, 27(2):95–98, 1988.

[80] 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.

[81] A. D. Santis, A. L. Ferrara, and B. Masucci. Cryptographic key assignment schemes for any access control policy. *Inf. Process. Lett.*, 92(4):199–205, 2004.

[82] 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.

[83] A. D. Santis, A. L. Ferrara, and B. Masucci. Efficient provably-secure hierarchical key assignment schemes. *Theor. Comput. Sci.*, 412(41):5684–5699, 2011.

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

[85] B. Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In *Public Key Cryptography*, pages 53–70, 2011.

[86] S. Yamada, N. Attrapadung, G. Hanaoka, and N. Kunihiro. Generic constructions for chosen-ciphertext secure attribute based encryption. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2011.