A Framework for the Cryptographic Enforcement of Information Flow Policies

James Alderman
Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK
James.Alderman@rhul.ac.uk

Jason Crampton
Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK
Jason.Crampton@rhul.ac.uk

Naomi Farley
Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK
Naomi.Farley.2010@live.rhul.ac.uk

ABSTRACT

It is increasingly common to outsource data storage to untrusted, third party (e.g. cloud) servers. However, in such settings, low-level online reference monitors may not be appropriate for enforcing read access, and thus cryptographic enforcement schemes (CESs) may be required. Much of the research on cryptographic access control has focused on the use of specific primitives and, primarily, on how to generate appropriate keys and fails to model the access control system as a whole. Recent work in the context of role-based access control has shown a gap between theoretical policy specification and computationally secure implementations of access control policies, potentially leading to insecure implementations. Without a formal model, it is hard to (i) reason about the correctness and security of a CES, and (ii) show that the security properties of a particular cryptographic primitive are sufficient to guarantee security of the CES as a whole.

In this paper, we provide a rigorous definitional framework for a CES 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 (e.g. key assignment schemes) as components of a CES.

KEYWORDS

Cryptographic Enforcement Scheme; Information Flow Policy; Access Control; Cryptography; Key Assignment Scheme; Attribute-based Encryption

ACM Reference format:
James Alderman, Jason Crampton, and Naomi Farley. 2017. A Framework for the Cryptographic Enforcement of Information Flow Policies. In *Proceedings of SACMAT'17, Indianapolis, IN, USA, June 21-23, 2017,* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3078861.3078868

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

SACMAT'17, June 21-23, 2017, Indianapolis, IN, USA

ACM ISBN 978-1-4503-4702-0/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3078861.3078868

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 authorized 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 authorized to read. Furthermore, changes to the policy, such as extending or retracting the access rights of a user, or changing the security level of an object should be supported by the CES; 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 is *guaranteed* to permit only authorized requests, efficient cryptographic primitives are usually *computationally* secure (due to their probabilistic nature). Further, there may be real-world concerns to be addressed by an implementation that are not required in idealized, theorectical models. Thus, as observed by Ferrara *et al.* [12], 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, 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 emphasize the gap between policy specification and cryptographic enforcement mechanisms, let us consider Key Assignment Schemes (KASs) [3] 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 — e.g. 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 users may locally store their keys and could continue to decrypt objects for which they are no longer authorized. 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.

1.1 Related Work and Motivation

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) [2, 9] and functional encryption schemes, especially *Attribute-based Encryption* (ABE) [6, 17]. Throughout this paper, 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 than read-access to cryptographically enforce and typically requires additional assumptions on the trustworthiness and capabilities of the storage provider, or additional trusted entities [10]. In particular, whilst one can often use cryptographic primitives that provide data origin authentication to *detect* data originating from an unauthorized writer [14, 22], it can be difficult to *prevent* unauthorized writes to the (externally controlled) file-system in the first place. Furthermore, to ensure correctness of the system following an unauthorized write, one must ensure the storage provider maintains the ability to 'roll-back' data objects or to otherwise ensure that legitimate writes are maintained. In this paper, like most related work, we focus our attention on read-only policies, with the observation that *detection* mechanisms should be a simple future extension to this work if required.

Key Assignment Schemes (KASs) [2, 9] are symmetric cryptographic primitives that can be used to enforce read-only information flow policies. Security notions for KASs [3] capture the requirements that no (collusion of) users may compute a key for which they are unauthorized (*key recovery*), and the stronger notion that no information is leaked about keys for which users are unauthorized (*key indistinguishability (KI)*).

While the above security notions capture the required security of generated keys (i.e keys do not reveal information about other keys), they do not capture the *distribution, use* and *update* of such keys. Furthermore, when considering the use of a KAS *within* a CES, it becomes clear that key recovery is not a suitable property and that key indistinguishability alone is not sufficient. Indeed, the security requirements of a KAS and CES are intrinsically different.

Key indistinguishability of a KAS states that a user who is not authorized to hold a *key* cannot learn anything about the key even having learned the keys of other unauthorized users. We argue that a secure CES requires that an unauthorized user attempting to access a particular *object* cannot learn anything about the *data* written to that object[1] even if it can learn the keys of other unauthorized 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 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 [5] can be trivially insecure if, for example, the file-system leaks information about other keys defined by the KAS when writing 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.* [12] emphasize the importance of providing a formal model for secure Cryptographic Role-based Access Control. They describe how cryptographic access-control schemes often only informally analyze 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 [15] 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. In concurrent work, Damgård *et al.* introduced the notion of *Access Control Encryption* [10] 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 entity known as the *Sanitizer* to process all data sent over public channels.

1.2 Contributions

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 paper 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.

Ferrara *et al.* [12] studied the setting of *role-based access control* (RBAC). In this paper, we consider CESs for read-only information flow policies. Crampton [8] 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.* to bridge the gap between the specification of access control models and the capabilities of cryptographic primitives. Indeed, as future work, Ferrara *et al.* [12]

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

suggested modeling general access control frameworks; one can view our work as a step towards this goal.

Whilst there is a wealth of work considering cryptographic access control requirements [1, 2, 9, 11, 14, 17–21], 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 analyze specific CES instantiations to ensure correctness and security.

We begin in Section 2 by introducing some notation and recalling basic concepts related to information flow policies. In Section 3, we introduce our model of CESs and classify the required functionality, before defining correctness and security in Section 4. In Section 5, we discuss some example schemes, highlighting their shortcomings in the context of our model. We conclude the paper with a summary of our contributions and some ideas for future work.

2 PRELIMINARIES

We write $a \leftarrow x$ to denote the assignment of x to variable a, whilst $a \xleftarrow{\$} X$ denotes a being assigned a value selected uniformly at random from the set X. We write $a \leftarrow B(c)$ to denote a polynomial time algorithm B being run on input c and the output being assigned to a, and write $a \xleftarrow{\$} B(c)$ if B is probabilistic polynomial time (PPT). We denote a security parameter by ρ and its unary representation by 1^ρ. A function f is *negligible* if, for every polynomial $p(\cdot)$, there exists an N such that for all integers $n > N$, $f(n) < \frac{1}{p(n)}$.

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 $A[0], \ldots, A[n-1]$.

A *partially ordered set* or poset is a pair $(L, \leqslant)$, where $\leqslant$ is a binary, reflexive, anti-symmetric, transitive relation on L. For a poset $(L, \leqslant)$, we write $x < y$ if $x \leqslant y$ and $x \neq y$ and may write $x > y$ if $y < x$. The empty set is denoted $\emptyset$.

A *read-only information flow policy* is a tuple $P = ((L, \leqslant), U, O, \lambda)$, where (i) $(L, \leqslant)$ is a partially ordered set of *security labels*; (ii) U is the set of users; (iii) O is the set of data objects; and (iv) $\lambda : U \cup O \rightarrow L$ is a function mapping users and objects to security labels in L. We say $u \in U$ is *authorized* 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 $\sqcup$. For any object o that is "dormant" or "inactive", we set $\lambda(o)$ equal to $\sqcap$; and for any user u that is dormant, 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 a dormant 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 $\sqcap$.

Traditionally, access control policies can be enforced by intercepting all attempts by users to interact with protected objects and determining whether the interaction is authorized. These functions are performed by what is known as a reference monitor (or, in more modern settings, the policy enforcement and policy decision points), a trusted software component that implements the logic of the authorization policy to evaluate a request from u to read o. Roughly speaking, the reference monitor instructs an unintelligent storage system to release an object to the user if the interaction is found to be authorized.

3 CRYPTOGRAPHIC ENFORCEMENT OF INFORMATION FLOW POLICIES

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. Conversely, the data owner may not wish the storage provider to have read access to the data. Thus, 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 keys to authorized users.

As mentioned, we focus on *read* access in this paper. 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 authorized 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.

As mentioned in the introduction, we believe 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.

3.1 State Requirements

In a CES, data objects are encrypted using some kind of cryptographic primitive and 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

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

Table 1: Notation used for modeling states of entities

owner) and its state in the system (in a protected format with any necessary metadata). We will then, in Section 3.2, consider the algorithmic requirements to use, maintain and update these states, which will lead us to consider a classification of CESs according to their functional requirements. Table 1 summarizes the notation we shall introduce in the next section to describe states in a CES.

3.1.1 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 $\alpha(l)$ (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 $\alpha(l)$ in a KAS).

Each user u must be provided with a means to learn some or all of $\alpha(l)$ for all $l \leqslant \lambda(u)$. Similarly, each object o must be protected using some or all of $\alpha(\lambda(o))$.

The manager must store (or be able to efficiently regenerate) $\alpha(l)$ 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:

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

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:

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

We refer to the state of the system as a whole as $\mathrm{st}_{\mathcal{M}}$ and Π 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).

Example 3.1. Consider a CES instantiated using the ABE scheme of Goyal *et al.* [17], where each attribute corresponds to a security label. Then, for each label $l \in L$, the manager must define a secret exponent $\alpha(l) \in \mathbb{Z}_p$ and compute a public group element $\pi(l) = g^{\alpha(l)}$. Furthermore, the manager must store additional secret information $\phi \in \mathbb{Z}_p$ (the system-wide secret exponent). Finally, Π must additionally store the masking term $\psi = e(g, g)^{\phi}$.

3.1.2 Objects. Each 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 modeled *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 ($\alpha(\lambda(o))$ 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 labeling function λ;
- $d(o)$ is the data written to the object o and 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 pairs of the form $(o, d(o))$. 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. 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.

[2]Note that in this work, 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.

3.1.3 Users. A user u is authorized 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 key assignment schemes [3], a user $u \in U$ is given a single secret $\sigma(\lambda(u))$ enabling the derivation of decryption keys associated to any $y \leqslant \lambda(u)$. We may assume that st_u contains the label $\lambda(u)$.

3.2 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.

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

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

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

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 4.2, 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 Π) to attempt to remove the protection mechanism applied to the data $d(o)$. It outputs $d(o)$ (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 authorized. 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.

3.2.1 Writeable. Although Setup writes the initial data $d(o)$ specified by the policy for each object in O, in many systems one may wish to update the data stored in objects over the course of the system lifetime. A *writeable* CES allows the manager to update objects and supports the following algorithm:

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

This algorithm takes as input the object identifier o 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 includes $(o, \overline{d(o)'})$ in FS.

3.2.2 Refreshability. Over time, cryptographic material may need to be *refreshed* if material is compromised or lost, or following the removal of an authorized user. Computing advances 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 more efficient to provide a targeted Refresh algorithm (to be run by the manager):

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

Refresh takes a label $l \in L$, the state $\mathrm{st}_{\mathcal{M}}$ of the manager and Π as input (which, together, contain the material $\alpha(l)$ and $\pi(l)$ associated to the target label), and outputs updated values of $\mathrm{st}_{\mathcal{M}}$ and Π, along with a set of messages $\{msg_u\}_{u\in U}$, which may contain updated cryptographic material for authorized users.

We say that a CES is *refreshable* if it uses a 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 key assignment scheme [3], for example, $L' = \{l' \in L : l' \leqslant l\}$.) Following a refresh, therefore, we may need to update Π, st_u for some users (typically those where $\lambda(u) \in L'$) and $\overline{d(o)}$ for objects o where $\lambda(o) \in L'$.

3.2.3 Dynamic Policy. In some settings, it may be that the sets of objects and users 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 or an object becomes declassified). A basic solution to fulfilling this requirement is to re-run the Setup algorithm with a modified labelling function.

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

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

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, for example, ChUsL may affect the states of other users (or the secret information $\alpha(y)$ associated to labels $y \neq l'$) e.g. if the access rights of u are *decreased* then the cryptographic material for all labels that u is no longer authorized for may need to be changed; subsequently, objects protected using keys that have

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

Table 2: Algorithms required in different classes of CES

been updated may require re-protecting. Typically, ChObL could be implemented by decrypting $\overline{d(o)}$, calling Refresh on $\lambda(o)$ and re-encrypting $d(o)$ using $\alpha(l')$.

Recall that we assume a large population of users, many of which may be assigned to $\sqcup$. The "creation" of a user may be modeled as the activation of a user that has been assigned to $\sqcup$, whilst user deletion can be modeled 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.

3.2.4 Decentralized 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 st_u for all u that are affected, and distribute msg_u containing st_u. We call this a *centralized 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 could use some key derivation function to update their own user state using a counter value or nonce broadcast by the manager. Hence, we introduce a final algorithm UserUpdate, run by the user:

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

3.2.5 Classes of CES. We have seen that CESs in different settings may require different functionality. In Table 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 e.g. if the inputs are invalid.

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 see that to initialize 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 5.

3.2.6 System State Transition. The evolution of a CES over time can be modeled as a series of state transitions, $S_t \xrightarrow{a} S_{t+1}$, where a is an algorithm run by the manager that results in a change to the policy.[3] In a CES for a static, refreshable policy, for example, the Setup and Refresh algorithms cause a transition to another state – Read does not change the state of the system and thus produces a trivial or null state transition.

We now attempt to specify the minimal sets of items within the system that must be updated in some way to ensure that the enforcement scheme reflects the updated system following a command. The specific forms of updates will be very dependent on the specific implementation. Some schemes may choose to update additional items (e.g. non-refreshable schemes may update *all* user states following an update). Here, we simply attempt to identify the *minimal* sets of items that are affected and that any implementation *must* deal with. For our purposes, we assume that all necessary updates are performed immediately i.e. we do not employ a lazy update mechanism.

Note that a transition from a state $S_t = (\mathrm{st}_{\mathcal{M}}, \Pi)$ to another state $S_{t'} = (\mathrm{st}'_{\mathcal{M}}, \Pi')$ *only* occurs if the associated conditions hold.

- $\mathsf{Write}(o, d(o)', \mathrm{st}_{\mathcal{M}}, \Pi)$: if $o \in O$, the manager protects $d(o)'$ (using cryptographic material related to o and $\lambda(o)$) and updates $\overline{d(o)} \in \Pi$.
- $\mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, \Pi)$: If $l \in L$, then let L' be the set of labels whose cryptographic material depends on that of l (e.g. in an iterative KAS [9], $L' = \{l' \in L : l' \leqslant l\}$). Then $\{\alpha(l), \pi(l) : l \in L'\}$ gets updated. All objects that are protected under cryptographic material that has been updated will need re-protecting under the refreshed material. Let O' be the set of such objects, then $\{\overline{d(o)} : o \in O'\}$ must be updated. In addition, a set of users, U', whose cryptographic material has been updated will also need to be issued material to update their user states.
- $\mathsf{ChUsL}(u, l', \mathrm{st}_{\mathcal{M}}, \Pi)$: If $l' \in L \setminus \sqcap$, $u \in U$ and $\lambda(u) \neq l'$, set $L' = \left\{l \in L : l \leqslant \lambda(u), l \not\leqslant l'\right\}$ (this is precisely the set of labels for which u is no longer authorized). Then we need to update the set $C = \{\alpha(l), \pi(l) : l \in L'\}$, and set $\lambda(u) \leftarrow l'$. For every object $o \in O$ protected using material in C, $\overline{d(o)}$ is updated. The smallest set of users whose state needs updating is the set of users $\{u' \in U : \lambda(u') \in L'\}$.
- $\mathsf{ChObL}(o, l', \mathrm{st}_{\mathcal{M}}, \Pi)$: Let $l = \lambda(o)$. If $l' \in L \setminus \sqcup$, $o \in O$ and $l \neq l'$, set $\lambda(o) \leftarrow l'$ and update $\overline{d(o)}$. κ_l and $\pi(l)$ should

[3] Since user states can be computed from the manager state and public information, they need not form part of the system state; therefore we do not consider UserUpdate as an algorithm that causes a system state transition.

be refreshed[4]. Such key refreshes are required to prevent the following scenario: suppose a user u locally stores $\overline{d(o)}$, where $\lambda(o) = l$, and u is not authorized for l. Suppose o is reassigned to label l', where $l' > l$, or $l' \parallel l$, and ChUsL is run such that $\lambda(u) = l$. Now, if κ_l has not been updated, then u can access the contents of his stored copy of $\overline{d(o)}$, although u is not authorized for o since $\lambda(o) \not\leqslant \lambda(u)$.

4 CORRECTNESS AND SECURITY

The security properties of a system employing cryptographic primitives are often defined using games. A game is "played" between a *challenger* and an *adversary*, $\mathcal{A}$, and seeks to model the actions of the adversary and its interactions with the system (represented by the challenger). A game typically comprises an interleaving of calls made by the challenger to algorithms provided by the system, and calls made by the adversary to "oracles". An adversary given oracle access is denoted $\mathcal{A}^{O}$, where the O denotes the set of oracles to which the adversary has access. We assume that all data sent amongst entities is done so via confidential and authenticated channels; the adversary is given access to publicly observable information, and oracles model his ability to act as the storage provider and to corrupt users to learn any confidential information available to attackers in a real system that take similar actions.

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 the manager — for example, an adversary may take a course of action (e.g. placing an order with a company) which it suspects will cause some data to be written to the file-system, and it may have some guess about the contents of that data; in the cryptographic game, this is modeled by allowing the adversary to request data (its guess) to be written (via a Write oracle), even though the adversary does not have the capability (e.g.the necessary access rights or encryption keys) to write to the file-system in the real system. If an adversary can glean any *additional* information from seeing protected objects (where the adversary knows the contents) in the game, it may be able to determine such information about the contents of data objects in a real file-system.

Most oracles include a call to a system algorithm and take as input a subset of the inputs to that algorithm. We do not provide oracles for Setup or any user-run algorithms as the adversary can run these 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).

[4] An efficient instantiation may add l to a refresh list and only update its key and public information when necessary (e.g. use lazy update mechanisms).

For the purposes of this framework, we make the assumption that all updates following a state transition occur immediately. In practice, one may need to lock files whilst updates are performed [14].

4.1 Correctness

Informally, an information flow policy is correctly enforced if all authorized 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 whereby we ensure that it is not possible for the system to enter a state in which an authorized 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 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 been last written to this object. We must ensure that the protection mechanism can be applied to, and removed from, data correctly by authorized users, and that the algorithms specified in the CES do not interfere with this operation. Recall that the storage provider is modeled 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 $\mathbf{Exp}^{\text{Correctness}}_{CES,\mathcal{A}}(1^\rho, P)$ in Figure 1, begins with the challenger setting up the system and initializing 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 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 Corrupt 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 decentralized CES). The challenger runs the UserUpdate algorithm to update the state of all *non-corrupted* users so that they remain synchronized 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

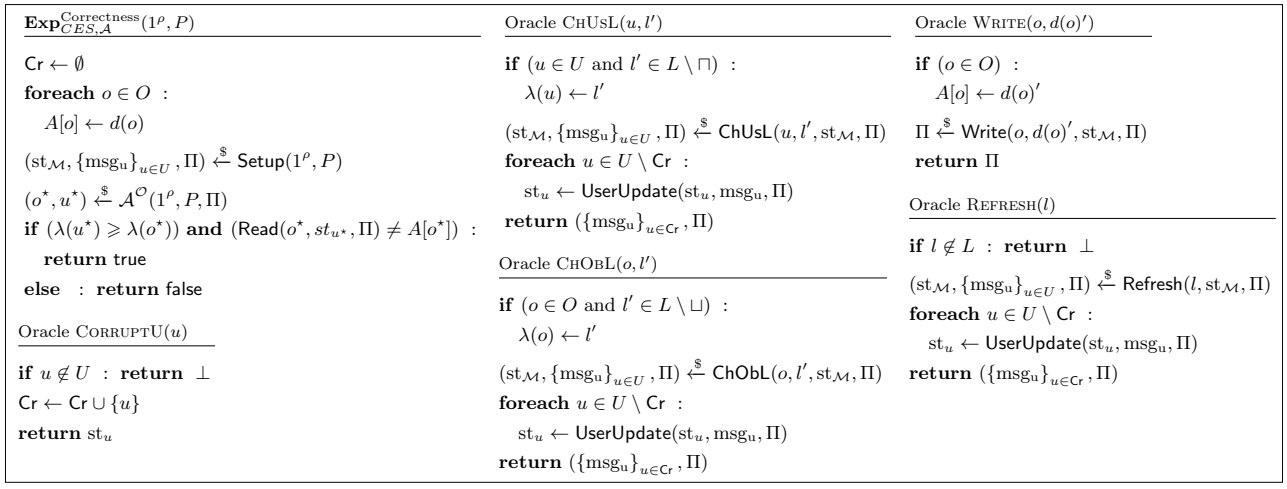

Figure 1: Correctness of a CES

$u^\star$ is authorized 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 authorized user not gaining the correct data.

Definition 4.1. Let $P = ((L, \leqslant), \mathcal{U}, O, \lambda)$ be an information flow policy. A CES for P is *correct* if for all probabilistic polynomial-time schedulers $\mathcal{A}$, all valid policies P and all security parameters ρ:

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

4.2 Security

Informally, a CES for a read-only information flow policy is *secure* if it denies all unauthorized read requests e.g. a user u cannot learn $d(o)$ if $\lambda(u) \not\geqslant \lambda(o)$. A stronger cryptographic notion of security may require that unauthorized users can learn *nothing* about the contents of objects for which they are unauthorized.[5] Unlike an enforcement mechanism based on a reference monitor, there are often no absolute guarantees of security in a CES because cryptographic primitives are probabilistic. Thus, security is defined in terms of the probability of an adversary learning something about an object that they are not authorized to read.

An ideal notion of security may be *semantic security* [16]. 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 [5] in which the adversary can choose data to be written (a chosen plaintext attack).

In our indistinguishability game for a CES, the adversary chooses a challenge object (for which it is unauthorized) and two data values. The challenger chooses one of the data values at random and writes it to the chosen object. To win, the adversary, having observed the file-system, must state which data item was written. The adversary can clearly 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.

This notion of indistinguishability implies (is stronger than) the notion that a user is not able to decrypt $\overline{d(o)}$ 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 maximize 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 $P = ((L, \leqslant), \mathcal{U}, O, \lambda)$ is captured in $\mathbf{Exp}^{\text{Ind}}_{CES,\mathcal{A}}(1^\rho, P)$ in Figure 2. The challenger C randomly chooses a bit $b \in \{0, 1\}$ and a challenge object identifier $o^\star$, and initializes an empty list Cr of corrupted users. C then initializes 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). C checks that no corrupted user is authorized 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.

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 authorized for $o^\star$;
(2) change the challenge object's label such that a corrupted user is now authorized for $o^\star$;
(3) change a corrupted user's label such that the user is now authorized 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;

[5] Whilst a user u who was authorized 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 authorized for o, the user should not be able to read any further writes to o.

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

$b \xleftarrow{\$} \{0,1\}; o^\star \leftarrow \perp; \mathsf{Cr} \leftarrow \emptyset$

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

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

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

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

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

    return false

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

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

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

else return false

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** $l' \in L \setminus \sqcup)$:

  $\lambda(o) \leftarrow l'$

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

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

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

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

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

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

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

  $\lambda(u) \leftarrow l'$

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

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

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

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

Oracle $\textsc{Refresh}(l)$

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

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

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

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

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

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

return Π

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

Figure 2: Security of a CES

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 the new policy as this may include an alternative poset (permitted policy changes can be effected through other oracles).

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.

Definition 4.2. A CES for an information flow policy is *secure* if for all probabilistic polynomial-time adversaries $\mathcal{A}$, all valid policies P and all security parameters ρ:

$$\left|\Pr\left[\text{true} \leftarrow \mathbf{Exp}^{\text{Ind}}_{CES,\mathcal{A}}(1^\rho, P)\right] - \frac{1}{2}\right| \leqslant f(\rho)$$

where f is a negligible function.

One may observe that a *secure* CES, in accordance with Definition 4.2, must employ some form of *foward-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 authorization for l, and being able to derive the old key for l to enable successful decryption of such ciphertexts.

5 EXAMPLE INSTANTIATIONS

A Key Assignment Scheme (KAS) [3] is defined by:

- $(\{\kappa_x, \sigma_x\}_{x\in L}, Pub) \xleftarrow{\$} \mathsf{KAS.Setup}(1^\rho, (L, \leqslant))$ takes a security parameter and a poset and outputs a key κ_x and secret σ_x for each label $x \in L$, along with some public derivation information Pub; and
- κ_x or $\perp \leftarrow \mathsf{KAS.Derive}(x, y, \sigma_y, Pub)$ takes labels $x, y \in L$, the secret for y and Pub, and outputs the key for label x if and only if $x \leqslant y$, else it outputs $\perp$.

Each user is given a secret associated to their security label and can derive all keys for which they are authorized. Figure 3 gives

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

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

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

foreach $x \in L$:

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

$\phi \leftarrow P$

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

foreach $u \in U$:

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

foreach $o \in O$:

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

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

$\psi \leftarrow (Pub, (L, \leqslant))$

$\Pi \leftarrow (\psi, FS)$

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

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

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

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

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

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

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

return $\perp$

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

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

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

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

$\Pi \leftarrow (\psi, FS)$

return Π

Figure 3: A Writeable, Centralized CES using a KAS

an example CES instantiation using a KAS (KAS) and a symmetric encryption scheme (SE) where the key space for KAS and SE is the same. The manager state includes all generated keys and secrets; each user state includes the secret assigned to the user's security label, and Π includes the public information output by the KAS.

Theorem 5.1. *Let* KAS *be secure in the sense of key indistinguishability and let* SE *be IND-CPA secure. Then the instantiation in Figure 3 is a secure static, writeable, centralized, non-refreshable CES.*

The proof of Theorem 5.1 can be found in Appendix A. 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 [4], 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.

Our second example uses a large-universe key-policy attribute-based encryption (KP-ABE) [17] scheme:

- $(MK, PP) \xleftarrow{\$} \mathsf{Setup}(1^\rho)$ takes a security parameter and outputs a master secret and public parameters;
- $C \xleftarrow{\$} \mathsf{Encrypt}(m, \gamma, PP)$ takes a message m, a set of attributes γ and PP, and outputs a ciphertext;
- $k_{\mathbb{A}} \leftarrow \mathsf{KeyGen}(\mathbb{A}, MK, PP)$ takes as input an access structure (policy) $\mathbb{A}$, the master secret key and public parameters, and outputs a key for the policy; and
- $(m \text{ or } \perp) \leftarrow \mathsf{Decrypt}(C, k_{\mathbb{A}}, PP)$ takes a ciphertext C encrypting m using an attribute set γ, a key $k_{\mathbb{A}}$ for a policy $\mathbb{A}$ and PP. It outputs the encrypted message m if $\gamma \in \mathbb{A}$ (the policy is satisfied) or $\perp$ otherwise.

Then, Figure 4 gives an instantiation of a dynamic, centralized, 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 [24]), 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).

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 [24, 25], it seems more difficult to reduce access rights rather than remove the user completely without assigning an entirely new user identifier.

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, to exemplify the utility of our framework. Further work should develop the definitions for key assignment schemes to meet the requirements of our framework for richer classes of CES. One could also expand our framework to consider other policies, including write-access, and security goals such as hiding the labels of users and objects.

REFERENCES

[1] Martín Abadi and Bogdan Warinschi. 2008. Security analysis of cryptographically controlled access to XML documents. *Journal of the ACM (JACM)* 55, 2 (2008), 6.

[2] Selim G. Akl and Peter D. Taylor. 1983. Cryptographic Solution to a Problem of Access Control in a Hierarchy. *ACM Trans. Comput. Syst.* 1, 3 (1983), 239–248.

[3] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. 2009. Dynamic and Efficient Key Management for Access Hierarchies. *ACM Trans. Inf. Syst. Secur.* 12, 3 (2009).

[4] Mikhail J. Atallah, Marina Blanton, and Keith B. Frikken. 2007. Efficient techniques for realizing geo-spatial access control. In *ASIACCS*, Feng Bao and Steven Miller (Eds.). ACM, 82–92.

[5] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. 1997. 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*. IEEE Computer Society, 394–403. https://doi.org/10.1109/SFCS.1997.646128

[6] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 321–334.

[7] Arcangelo Castiglione, Alfredo De Santis, and Barbara Masucci. 2016. Key Indistinguishability versus Strong Key Indistinguishability for Hierarchical Key Assignment Schemes. *IEEE Trans. Dependable Sec. Comput.* 13, 4 (2016), 451–460. https://doi.org/10.1109/TDSC.2015.2413415

[8] Jason Crampton. 2010. Cryptographic Enforcement of Role-Based Access Control. In *Formal Aspects in Security and Trust (Lecture Notes in Computer Science)*, Vol. 6561. Springer, 191–205.

[9] Jason Crampton, Keith M. Martin, and Peter R. Wild. 2006. On Key Assignment for Hierarchical Access Control. In *CSFW*. IEEE Computer Society, 98–111.

[10] Ivan Damgård, Helene Haagh, and Claudio Orlandi. 2016. Access control encryption: Enforcing information flow with cryptography. In *Theory of Cryptography Conference*. Springer, 547–576.

[11] Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. 2007. Over-encryption: management of access control evolution on outsourced data. In *Proceedings of the 33rd international conference on Very Large Data Bases*. VLDB endowment, 123–134.

[12] Anna Lisa Ferrara, Georg Fuchsbauer, and Bogdan Warinschi. 2013. Cryptographically Enforced RBAC. In *CSF*. IEEE, 115–129.

[13] Eduarda S. V. Freire, Kenneth G. Paterson, and Bertram Poettering. 2013. Simple, Efficient and Strongly KI-Secure Hierarchical Key Assignment Schemes. In *CT-RSA (Lecture Notes in Computer Science)*, Vol. 7779. Springer, 101–114.

[14] William C Garrison, Adam Shull, Steven Myers, and Adam J Lee. 2016. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 819–838.

[15] David K Gifford. 1982. Cryptographic sealing for information secrecy and authentication. *Commun. ACM* 25, 4 (1982), 274–286.

[16] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *Journal of computer and system sciences* 28, 2 (1984), 270–299.

[17] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security*. ACM, 89–98.

[18] Ehud Gudes. 1980. The design of a cryptography based secure file system. *IEEE Transactions on Software Engineering* 5 (1980), 411–420.

[19] Shai Halevi, Paul A Karger, and Dalit Naor. 2005. Enforcing Confinement in Distributed Storage and a Cryptographic Model for Access Control. *IACR Cryptology ePrint Archive* 2005 (2005), 169.

[20] Anthony Harrington and Christian Jensen. 2003. Cryptographic access control in a distributed file system. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*. ACM, 158–165.

[21] Bin Liu and Bogdan Warinschi. 2016. Universally Composable Cryptographic Role-Based Access Control. Cryptology ePrint Archive, Report 2016/902. (2016). http://eprint.iacr.org/2016/902.

[22] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. 2011. Attribute-Based Signatures. In *CT-RSA (Lecture Notes in Computer Science)*, Aggelos Kiayias (Ed.), Vol. 6558. Springer, 376–392.

[23] Matthew G. Parker (Ed.). 2009. *Cryptography and Coding, 12th IMA International Conference, Cryptography and Coding 2009, Cirencester, UK, December 15-17, 2009. Proceedings*. Lecture Notes in Computer Science, Vol. 5921. Springer.

[24] Nutta pong Attrapadung and Hideki Imai. 2009. Attribute-Based Encryption Supporting Direct/Indirect Revocation Modes, See [23], 278–300.

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

Parse $P = ((L, \leqslant), U, O, \lambda)$
$(\mathrm{MK}, \mathrm{PP}) \xleftarrow{\$} \mathsf{ABE.Setup}(1^\rho)$
for $l \in L$:
  $A[l] \leftarrow 0$
$k_M \xleftarrow{\$} \mathsf{ABE.KeyGen}((\bigvee_{l\in L} l||A[l]), \mathrm{MK}, \mathrm{PP})$
$\phi \leftarrow (A, k_M, P, \mathrm{MK})$
$\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
foreach $u \in U$:
  $k_u \xleftarrow{\$} \mathsf{ABE.KeyGen}((\bigvee_{l\leqslant\lambda(u)} l||A[l]), \mathrm{MK}, \mathrm{PP})$
  $\mathrm{st}_u \leftarrow (k_u, \lambda(u))$
foreach $o \in O$:
  $c_o \xleftarrow{\$} \mathsf{ABE.Encrypt}(d(o), \{\lambda(o)||A[\lambda(o)]\}, \mathrm{PP})$
  $\overline{d(o)} \leftarrow (c_o, o, \lambda(o))$
$FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
$\psi \leftarrow (PP, (L, \leqslant))$
$\Pi \leftarrow (\psi, FS)$
return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, \Pi)$

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

if $o \in O$:
  Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
  Parse $\mathrm{st}_u = (k_u, \lambda(u))$
  return $\mathsf{ABE.Decrypt}(c_o, k_u, \mathrm{PP})$
return $\perp$

$(\mathrm{st}_{\mathcal{M}}, \{\mathrm{msg}_u\}_{u\in U}, \Pi) \xleftarrow{\$} \mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, \Pi)$

if $l \in L$:
  $A[l] = A[l] + 1$
  $k'_M \xleftarrow{\$} \mathsf{ABE.KeyGen}((\bigvee_{l\in L} l||A[l]), \mathrm{MK}, \mathrm{PP})$
  foreach $u \in \{u \in U : l \leqslant \lambda(u)\}$:
    $\mathrm{st}_u \xleftarrow{\$} (\mathsf{ABE.KeyGen}((\bigvee_{l'\leqslant\lambda(u)} l'||A[l']), \mathrm{MK}, \mathrm{PP}), \lambda(u))$
  foreach $o \in \{o \in O : \lambda(o) = l\}$:
    Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
    $d \leftarrow \mathsf{ABE.Decrypt}(c_o, k_M, \mathrm{PP})$
    $\overline{d(o)} \xleftarrow{\$} (\mathsf{ABE.Encrypt}(d, \{\lambda(o)||A[\lambda(o)]\}, \mathrm{PP}), o, \lambda(o))$
  $\phi \leftarrow (A, k'_M, P, \mathrm{MK})$
  $\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $\Pi \leftarrow (\psi, FS)$
  return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, \Pi)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, \Pi)$

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

if $o \in O$ and $l' \in L \setminus \sqcup$:
  $l \leftarrow \lambda(o)$
  Parse $\overline{d(o)} = (c_o, o, \lambda(o))$
  $d \leftarrow \mathsf{ABE.Decrypt}(c_o, k_M, \mathrm{PP})$
  $\overline{d(o)} \xleftarrow{\$} \mathsf{ABE.Encrypt}(d, \{l'||A[l']\}, \mathrm{PP}), o, l')$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $\lambda(o) \leftarrow l'$
  $\Pi \leftarrow (\psi, FS)$
  return $\mathsf{Refresh}(l, \mathrm{st}_{\mathcal{M}}, \Pi)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, \Pi)$

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

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 $\overline{d(o)} = (c_o, o, \lambda(o))$
      $d \leftarrow \mathsf{ABE.Decrypt}(c_o, k_M, \mathrm{PP})$
      $\overline{d(o)} \xleftarrow{\$} (\mathsf{ABE.Encrypt}(d, \{\lambda(o)||A[\lambda(o)]\}, \mathrm{PP}), o, \lambda(o))$
  if $X \neq \emptyset$:
    $k_M \xleftarrow{\$} \mathsf{ABE.KeyGen}((\bigvee_{l\in L} l||A[l]), \mathrm{MK}, \mathrm{PP})$
    $\phi \leftarrow (A, k_M, P, \mathrm{MK})$
    $\mathrm{st}_{\mathcal{M}} \leftarrow \phi$
    $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
    $\Pi \leftarrow (\psi, FS)$
    foreach $u' \in \{u' \in U \setminus u : \exists x \in X, x \leqslant \lambda(u')\}$:
      $st_{u'} \xleftarrow{\$} (\mathsf{ABE.KeyGen}((\bigvee_{x\leqslant\lambda(u')} x||A[x]), \mathrm{MK}, \mathrm{PP}), \lambda(u'))$
    $\lambda(u) \leftarrow l'$
    $\mathrm{st}_u \xleftarrow{\$} (\mathsf{ABE.KeyGen}((\bigvee_{x\leqslant l'} x||A[x]), \mathrm{MK}, \mathrm{PP}), l')$
  return $(\mathrm{st}_{\mathcal{M}}, \{\mathrm{st}_u\}_{u\in U}, \Pi)$
return $(\mathrm{st}_{\mathcal{M}}, \emptyset, \Pi)$

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

if $o \in O$:
  $\overline{d(o)} \xleftarrow{\$} (\mathsf{ABE.Encrypt}(d(o)', \{\lambda(o)||A[\lambda(o)]\}, \mathrm{PP}), o, \lambda(o))$
  $FS \leftarrow \left\{\overline{d(o)} : o \in O\right\}$
  $\Pi \leftarrow (\psi, FS)$
return Π

Figure 4: Construction of a Dynamic, Centralized, Refreshable, Writeable CES using Attribute-based Encryption

$\mathbf{Exp}^{\mathrm{S-KI-ST}}_{\mathcal{A},(L,\leqslant)}(1^\rho)$

$l^\star \xleftarrow{\$} \mathcal{A}(1^\rho, (L, \leqslant))$
$((\sigma_l, \kappa_l)_{l\in L}, Pub) \xleftarrow{\$} \mathsf{Setup}(1^\rho, (L, \leqslant))$
$b \xleftarrow{\$} \{0, 1\}$; **if** $b = 1$ **then** $\kappa^\star \leftarrow \kappa_{l^\star}$, **else** $\kappa^\star \xleftarrow{\$} K$
$b' \xleftarrow{\$} \mathcal{A}(Pub, Corrupt, Keys, \kappa^\star)$
return $b' = b$

Figure 5: Static Strong Key Indistinguishability of a KAS

[25] Jun-lei Qian and Xiao-lei Dong. 2011. Fully secure revocable attribute-based encryption. *Journal of Shanghai Jiaotong University (Science)* 16 (2011), 490–496.

A SECURITY PROOF OF THEOREM 1

A symmetric-key encryption scheme [5] comprises:

- $SK \xleftarrow{\$} \mathrm{KeyGen}(1^\rho)$ takes a security parameter and outputs a secret key.
- $c \xleftarrow{\$} \mathrm{Encrypt}_{SK}(m)$ takes as input a secret key SK and a message m and outputs a ciphertext c.
- $(m \text{ or } \perp) \leftarrow \mathrm{Decrypt}_{SK}(c)$ takes a key and a ciphertext, and outputs a message m or a failure symbol $\perp$.

A KAS is *Strongly Key Indistinguishable* (SKI) [13] if for all PPT adversaries $\mathcal{A}$ and posets $(L, \leqslant)$:

$$2\left|\Pr\left[\mathbf{Exp}^{\mathrm{S-KI-ST}}_{\mathcal{A},(L,\leqslant)}(1^\rho) = b\right] - \frac{1}{2}\right| \leqslant f(\rho),$$

where f is a negligible function, $\mathbf{Exp}^{\mathrm{S-KI-ST}}_{\mathcal{A},(L,\leqslant)}(1^\rho)$ is given in Figure 5 where K is the key space, $Corrupt = \left\{\sigma_l : l \in L, l < l^\star\right\}$ and $Keys = \left\{\kappa_l : l \in L, l \neq l^\star\right\}$.

Proof. We first define a modified game, **Game 1**, which is the same as that defined in Definition 4.2 (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 are 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.

Although Theorem 5.1 requires a Key Indistinguishable (KI) KAS, we instead use the Strong KI (SKI) [13] property instead which is polynomially equivalent [7] but provides the adversary with *all* keys (except the challenge key) to model key leakage. We find SKI more convenient for proving interactive reductions as all keys are immediately available.

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. Let 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 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 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 Refresh will be called before the challenge). Let r be a counter, initially 0, denoting the number of calls $\mathcal{A}_{CES}$ makes to Refresh. Thus, $\mathcal{A}_{KI}$ makes a guess $c \xleftarrow{\$} L$ for the challenge label and guesses $i \xleftarrow{\$} \{0, 1, \ldots, q\}$, for $q = poly(\rho)$, for the value of r when the challenge parameters are chosen.

$\mathcal{A}_{KI}$ sends c to C_{KI} as its SKI challenge label. C_{KI} runs $(\{\sigma_l, \kappa_l\}_{l \in L}, Pub) \xleftarrow{\$} \mathsf{KAS.Setup}(1^\rho, (L, \leqslant))$, and chooses a random bit $b \xleftarrow{\$} \{0, 1\}$; if $b = 0$, $\kappa^\star = \kappa_c$, else $\kappa^\star$ is chosen randomly from the key space. C_{KI} sends the KAS public information, the set of all keys except for the challenge key, the set of all secrets for labels $l' \leqslant c$, and the challenge key $\kappa^\star$ to $\mathcal{A}_{KI}$. $\mathcal{A}_{KI}$ initializes $\mathsf{Cr} = \emptyset$ and $o^\star = \perp$.

Now, if $i \neq 0$, then $\mathcal{A}_{KI}$ does not embed the challenger's outputs in the initial CES setup. Instead, it runs Setup as in Figure 3, running KAS.Setup itself. Else, when $i = 0$, $\mathcal{A}_{KI}$ sets $\mathrm{st}_{\mathcal{M}}$ to include $\{\{\sigma_{l'} : l' < c, l' \in L\}, \{\kappa_l : l \in L \setminus \{c\}\}\}$ and Π to include Pub. For each user, if $\lambda(u) < c$, $\mathcal{A}_{KI}$ defines $\mathrm{st}_u = \{\sigma_{\lambda_u}, \lambda_u\}$, and $\mathrm{st}_u = \{\cdot, \lambda_u\}$ otherwise.

$\mathcal{A}_{CES}$ is given Π and a set of oracles O as in Figure 2. If $\mathcal{A}_{CES}$ calls CorruptU on a user $u \in U$ where $\lambda(u) > c$, then $\mathcal{A}_{KI}$ loses the game (c would now be an invalid challenge and so the initial guess of c was wrong). Similarly, $\mathcal{A}_{KI}$ loses if $\mathcal{A}_{CES}$ chooses $o^\star$ such that $\lambda(o^\star) \neq c$. Whenever the Refresh oracle is called, r is increased by 1.

When $r = i$, $\mathcal{A}_{KI}$ runs Refresh but instead of running KAS.Setup, it uses the key material received from C_{KI}, and re-initializes the state of the manager, users, and objects as described above in Setup where $i = 0$. $\mathcal{A}_{KI}$ loses the game if r exceeds i and $\mathcal{A}_{CES}$ has not yet chosen a challenge object.

Eventually, $\mathcal{A}_{CES}$ guesses that it was playing **Game** b'. $\mathcal{A}_{KI}$ forwards b' to 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 $\frac{\mathsf{Adv}(\mathcal{A}_{CES})}{q|L|}$ where $q = poly(\rho)$ is the number of calls to the refresh oracle. 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 $\mathcal{A}_{IND}$ can use $\mathcal{A}_{CES}$ to win the IND-CPA game against a challenger C_{IND}.

C_{IND} randomly selects a key k from the key space, selects a random bit $b \xleftarrow{\$} \{0, 1\}$, and gives $\mathcal{A}_{IND}$ access to an encryption oracle $\eta_{k,b}$, 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 LoR IND-CPA game instead of Find-then-Guess [5] as it allows multiple challenges; thus we need only guess the challenge label and not the object itself.)

$\mathcal{A}_{IND}$ runs line 1 of the CES experiment and guesses the security label c of the challenge object $o^\star$ that $\mathcal{A}_{CES}$ will choose. All encryptions using the key κ_c will be replaced by encryptions under k. When an object o with label c is to be written, the adversary calls the encryption oracle $\eta_{k,b}$ on inputs $(d(o)', d(o)')$ to obtain an encryption under k. $\mathcal{A}_{IND}$ runs line 2 of the CES experiment and gives oracle access to $\mathcal{A}_{CES}$. If $\mathcal{A}_{CES}$ corrupts a user $u \in U$ such that $\lambda(u) \geqslant c$, the experiment fails (the guess of c was wrong). 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; else, $\mathcal{A}_{IND}$ calls $\eta_{k,b}(m_0, m_1)$, and writes the result to $\overline{d(o^\star)}$.

Eventually, $\mathcal{A}_{CES}$ sends b' to $\mathcal{A}_{IND}$ as its guess of b; $\mathcal{A}_{IND}$ forwards this to C_{IND}. If $\mathcal{A}_{CES}$ can correctly guess which data was written with non-negligible advantage $\mathsf{Adv}(\mathcal{A}_{CES})$, then $\mathcal{A}_{IND}$ wins the IND-CPA game with non-negligible advantage $\frac{\mathsf{Adv}(\mathcal{A}_{CES})}{|L|}$. This is a contradiction, since the encryption scheme is assumed IND-CPA secure. □