Design and Implementation of a Key-Lifecycle Management System

Mathias Björkqvist Christian Cachin Robert Haas Xiao-Yu Hu
Anil Kurmus René Pawlitzek Marko Vukolić

IBM Research - Zurich, CH-8803 Rüschlikon, Switzerland
{mbj,cca,rha,xhu,kur,rpa,mvu}@zurich.ibm.com
September 23, 2009

Abstract

Key management is the Achilles' heel of cryptography. This work presents a novel *Key-Lifecycle Management System (KLMS)*, which addresses two issues that have not been addressed comprehensively so far.

First, KLMS introduces a *pattern-based* method to *simplify* and to *automate* the *deployment* task for keys and certificates, i.e., the task of associating them with endpoints that use them. Currently, the best practice is often a *manual* process, which does not scale and suffers from human error. Our approach eliminates these problems and specifically takes into account the lifecycle of keys and certificates. The result is a centralized, scalable system, addressing the current demand for automation of key management.

Second, KLMS provides a novel form of *strict access control* to keys and realizes the first cryptographically sound and secure access-control policy for a key-management interface. Strict access control takes into account the cryptographic semantics of certain key-management operations (such as key wrapping and key derivation) to prevent attacks through the interface, which plagued earlier key-management interfaces with less sophisticated access control.

Moreover, KLMS addresses the needs of a variety of different applications and endpoints, and includes an interface to the Key Management Interoperability Protocol (KMIP) that is currently under standardization.

1 Introduction

Cryptography is used to secure many information-technology systems, ranging from encrypting data on storage and establishing virtual private networks to protecting communication with mobile devices and using SSL certificates for e-commerce over the Internet. All uses of cryptography rely on the proper keys being present. Key management deals with the *lifecycle* of cryptographic keys, with operations for creating, importing, storing, reading, updating, exporting, and deleting them, and with distributing keys before they are used in cryptographic functions. An important aspect is to manage the attributes of keys that govern their usage and their relation to other keys.

Complications with key distribution are seen as the source of most operational problems with secure systems using cryptography. A key-management system must *provision* the appropriate keys and *deploy* them to *endpoints*, the entities that consume keys and use them for cryptographic functions. Managing a large number of keys manually does not scale, suffers from human error, and is prohibitively expensive. As a result, there is a great demand for *automated* key-management today, provided by centralized, scalable systems.

In an enterprise context, multiple users associated with many endpoints access the key-management system and perform operations on the objects that it maintains. These objects include *symmetric keys*,

public keys, *private keys*, and *certificates*. A key-management system focuses on attribute handling rather than on cryptographic functions. But a comprehensive key-management system will also support a small set of cryptographic operations, including creating a key, issuing a certificate, to *derive* a new key (a deterministic operation that creates a symmetric key from an existing one), and to *wrap* or *unwrap* a key with another key (wrapping means to encrypt a target key with another key for export and transfer to another system).

In this paper, we describe the design and implementation of a prototype *Key-Lifecycle Management System (KLMS)*. It unifies key management for local and remote endpoints and handles many different types of cryptographic objects in a flexible way. KLMS addresses enterprise-level key management and covers many endpoints that require cryptographic keys, from heterogeneous applications, servers, and network devices to storage devices and media (e.g., tape cartridges). The system can provision keys to any application that can receive keys through the Java KeyStore (JKS) interface, as for example, file-based keystores in several formats, and it provides a prototype server for the Key Management Interoperability Protocol (KMIP), which is under standardization by OASIS [16].

KLMS introduces two novel features of a key-management system: first, the notion of *deployment patterns* to *automate* the administration and the deployment tasks for keys and certificates; and second, a *strict* implementation of *access control* to keys, which takes into account the cryptographic semantics of certain key-management operations and realizes the first cryptographically sound access-control policy for a key-management interface. We now briefly describe these two contributions.

Automated deployment. Often multiple related keys must be administered by the management system. To simplify this task, several keys can be grouped and *deployed* together to one or more *endpoints*. Such deployments can be structured according to certain *patterns*. In the example of a TLS-protected web server connected to the Internet, which is clustered for high availability, the same private key and certificate should be deployed to all nodes in the cluster. On the other hand, for a communication setup where multiple servers identify each other using their client-certificates through TLS-connections, every server should receive its own private key plus the public keys of all servers.

For supporting such scenarios, KLMS provides a novel pattern-based method for automated key and certificate deployment. A flexible *deployment manager (DM)* automates deployment and shields the system administrator from the lower-level key creation and distribution tasks. Once a suitable pattern and the supporting policies for a particular application are defined, KLMS automatically generates, distributes, and maintains as many keys or key pairs as necessary, and responds dynamically to changes of the topology, when new endpoints are added to the application.

KLMS also takes care of automatically managing the lifecycle of keys. It may create keys ahead of time and only maintain them internally, provisioned for a certain application. At the time of activation of a key, KLMS automatically deploys it to endpoints for the duration of its active life-time, and withdraws the key again from all endpoints when it expires. The key-lifecycling logic is tightly coupled with the deployment manager.

Strict access control. Every key-management system serves keys to users, the principals that invoke its operations. In the usual *basic* form of access control, the system decides about access to a key only by consulting an *access-control list (ACL)* associated with the key. But because the operations of the system allow users to create complex relationships between keys, through key derivation and key wrapping, basic access control may have security problems. For example, if there exists a key k_1 that some user is not allowed to read, but the user may wrap k_1 under another key k_2 and export the wrapped representation, the user may nevertheless obtain the bits of k_1. Another example is a key that was derived from a parent key; when a user reads the parent key, the user implicitly also obtains the cryptographic material of the derived key.

In general, a cryptographic interface that manages keys and allows the creation of such dependencies among keys poses the problem that access to one key may give a user access to many another keys. This issue has been identified in the APIs of several cryptographic modules [2, 7, 9, 11] and may lead to serious security breaches when one does not fully understand all implications of an API.

Therefore, KLMS provides a *strict* mode of access control, in which decisions take the semantics of the key-management API into account, and implements a cryptographically sound access-control policy on all symmetric keys and private keys. The above issues with basic access control are eliminated with strict access control. Our strict access-control policy builds on the work of Cachin and Chandran [8], which describes a secure cryptographic token interface and introduces a cryptographically strong security policy. A strict access-control decision not only depends on the ACL of the corresponding key, but also takes into account the ACLs of related keys and the history of past operations executed on them. It prevents *any* unauthorized disclosure of a symmetric key or a private key.

1.1 Related Work

The need for building enterprise-scope key management systems has been widely recognized [6], and the US National Institute of Standards and Technology (NIST) has issued a general recommendation for key management [4]. Several commercial enterprise key-management systems are on the market, including HP™ StorageWorks™ Secure Key Manager, IBM® Distributed Key Management System (DKMS), IBM Tivoli® Key Lifecycle Manager, NetApp® Lifetime Key Management™, Sun™ StorageTek™ Crypto Key Management System, and Thales®/nCipher® keyAuthority®.

In order to integrate these proprietary solutions, multiple efforts are currently underway to build and standardize key-management standards for open networks: the W3C XML Key Management Specification (XKMS) [19], the IEEE P1619.3 Key Management Project [14], and the Key Management Interoperability Protocol (KMIP) standardization under the auspieces of OASIS [16] are some of the most prominent ones. Cover [10] gives an up-to-date summary of the current developments.

There is a rich literature on using *patterns* in software engineering and in systems management. One representative work that links this area with deployment on a system are the *deployment patterns for a service-oriented architecture* of Arnold et al. [3]; these patterns distribute applications to servers automatically by an algorithm that is available with the IBM Rational® Software Architect product.

Access control and management of cryptographic keys are related in many ways. One prominent line of research, starting with Akl and Taylor [1], has concentrated on representing a hierarchy (modeled as a partially ordered set) of users and their access privileges in a directed graph of keys, where a user inherits the privileges of all users below; this and subsequent work focuses on minimizing the number of keys stored by a user.

Because the key-management server provides the above-mentioned cryptographic functions, it represents a *cryptographic security API* accessible over a network. Security APIs stand at the boundary between untrusted code and trusted modules capable of maintaining internal state. Cryptographic security APIs are typically provided by cryptographic tokens [2], hardware-security modules (HSM) like IBM's 4764 cryptoprocessor that supports the IBM CCA interface [12, 15] and generic PKCS #11-compliant [17] modules, smartcards, or the Trusted Platform Module [18]. The study of cryptographic security APIs has so far been limited to programming APIs and to libraries; this paper extends their study to protocols for open networks.

1.2 Organization of the Paper

Section 2 describes the architecture of KLMS and its data model, and motivates some design choices. The main contributions, automated deployment and strict access control, are described in Sections 3 and 4, respectively. We conclude the paper with Section 5 by providing further information about the implementation and discussing experimental integration of support for KMIP.

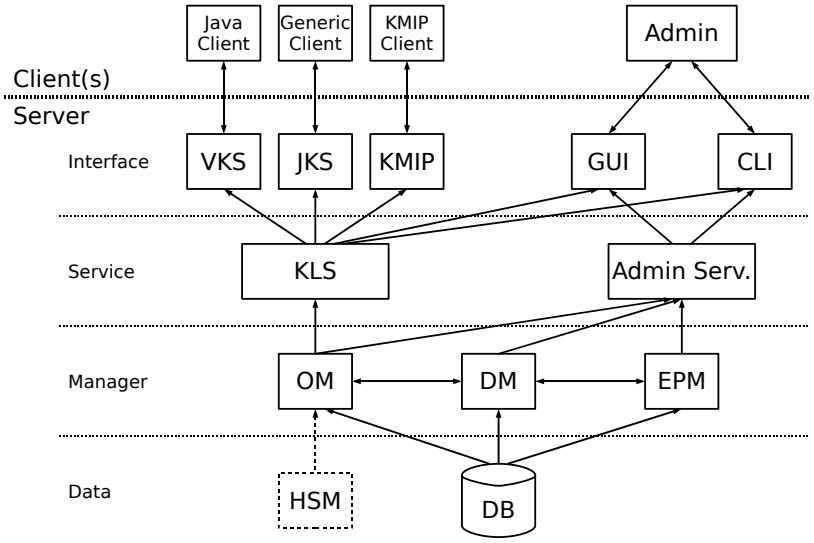

Figure 1: Key-Lifecycle Management System architecture (see text).

2 Model

2.1 System Architecture

The architecture of KLMS is shown in Figure 1. It is foreseen that the server is implemented in Java and runs on a web-application server. The KLMS server interacts with the clients (endpoints) through several types of interfaces, as described below. Administrators use a different interface than the clients to access the server. The server itself is structured in four layers; bottom-up, these are a data layer, a manager layer, a service layer, and an interface layer, as described next.

Data Layer. The data layer stores all information in a persistent *Database (DB)*. Internally, DB accesses a standard SQL database through the JDBC interface. All state information of KLMS is maintained by DB, so that KLMS does not lose any data because of a system crash. Meta-data about keys and certificates is stored in DB, as well as the cryptographic material itself. Some corporate security policies mandate that certain keys exist in cleartext only in a hardware-security module (HSM); the architecture supports this feature, by including a master key stored in an HSM and using it to encrypt all cryptographic material in DB.

Manager Layer. KLMS contains three components that provide low-level functionalities: an *Object Manager (OM)*, a *Deployment Manager (DM)*, and an *Endpoint Manager (EPM)*.

First, OM provides a simple interface to manipulate the *cryptographic objects* supported by KLMS. OM can add new objects, read, modify, search, and delete them in the DB, and maintains an in-memory object cache that is used to speed up read operations.

Second, DM takes care of administering *deployments* and *deployment bundles*. A deployment is an association between an object and an endpoint in the sense that KLMS provisions the object for use in cryptographic operations by the endpoint. The deployment policy realized by the DM dictates when and under which condition a deployed object finally becomes available to an endpoint through an interface; the deployment policy is described in Section 3. A *deployment bundle* is a set of deployments, which are grouped to support a given application.

Finally, EM controls the endpoints in the interface layer of the server, registering them in KLMS, potentially creating new file-backed JKS endpoints, and listening to protocol ports to which KMIP clients connect. EM unifies the different types of endpoints towards the rest of the server.

Service Layer. The service layer provides two modules: a *Key-Lifecycle Service (KLS)*, which is used by endpoints and by an administrator, and an *Admin Service*, which is only accessed by the administrator.

KLS represents the core of the server. It implements all operations related to keys and certificates that are available to endpoints and to users, drives automated deployment and lifecycle operations in conjunction with DM, and enforces access control. KLS can distinguish between different *users*, the principals that access it; every invocation of an operation occurs in the context of a *session*, which represents a user that has been securely authenticated by KLMS. The data model and the operations of KLS are described in the next section.

The Admin Service controls the allocation of endpoints and deployments through EPM and DM, respectively. Access to its operations also occurs in the context of a session, but is restricted to users with the corresponding permission. The Admin Service also allows archive and recovery operations for individual keys and for the whole database. Both modules, KLS and Admin Service, generate audit events.

Interface Layer. Three types of *endpoint interfaces* interact with the clients. The *Virtual Keystore (VKS)* interface emulates the provider of a Java KeyStore, for applications that are hosted by the same application server as KLMS. The client reads and writes keys via VKS by issuing the "get" and "set" operations of the Java KeyStore interface. VKS is a *pull-style* synchronous interface, i.e., KLS can forward client calls to VKS directly to OM and DM.

The *Java Keystore (JKS)* interface accesses a named Java KeyStore as a client. A Java KeyStore is usually passive and its default implementation is a file, but depending on the installed Java Cryptography Extension (JCE) provider, many different entities may receive key material through the JKS interface (in particular, such clients need not be implemented in Java). JKS is a *push-style* asynchronous interface, because KLS calls the Java KeyStore interface and clients may retrieve keys from JKS at a later time.

A protocol interface provides an experimental implementation of the *Key Management Interoperability Protocol (KMIP)* draft standard [16]. KMIP is mostly a client-to-server protocol that offers rich functionality to manipulate keys and certificates. Many of its operations can be forwarded directly to KLS, but other operations are realized by an adapter module inside the KMIP interface. Ignoring the (optional) server-to-client operations in KMIP, the protocol interface is again *pull-style* and synchronous, similar to VKS. Clients connecting through KMIP need not be implemented in Java.

For the two keystore-based interfaces, EPM statically configures the user with which KLS is accessed. For the protocol-based interface, it is possible to take the user from the client context. For the pull-style interfaces, access control occurs when the client calls KLS; for the push-style JKS interface, on the other hand, access control must be enforced at the time when the deployment occurs.

Administrators access KLMS through a web-based *Graphical User Interface (GUI)* (built using the Hamlets framework [13]) or through a *Command-Line Interface (CLI)*; they both provide operations to deal with endpoints and to manage deployments. Note that clients who access the system through one of the endpoint interfaces cannot deploy keys or certificates in KLMS.

2.2 Data Model and Operations

KLMS manages *symmetric keys*, *public keys*, *private keys*, and *certificates*, which we summarily call *cryptographic objects* or simply *objects*. All key formats supported by the underlying Java platform and the JCE are available, including RSA, Diffie-Hellman, ElGamal, DSA, and EC-variants for public-key algorithms and symmetric keys of arbitrary length. Objects are composed of attributes and (possibly)

cryptographic material. Attributes contain meta-data about the use of the cryptographic material, and they are the main concern of key management. Attributes may be read and sometimes also modified by clients in KLMS. KLMS also provides templates that simplify the handling of attributes for multiple objects, but we do not describe them in detail here, as they are not our primary concern.

Key lifecycle		Access control	
State	*Deactivation time*	*Usage*	*Creator*
Initialization time	*Compromise time*	*Digest*	*Dependents*
Activation time	*Destroy time*	*Strict*	*Ancestors*
		ACL	*Readers*

Table 1: Object attributes relevant for key lifecycle and access-control features.

KLMS currently supports close to 50 different object attributes. Rather than listing them all, we focus on the subset that is relevant for lifecycle operations, for automated deployment, and for access control (see Table 1). Every object has a unique *identifier*. A crucial attribute is the *state* of an object in its lifecycle. Objects in KLMS follow a lifecycle according to NIST [4, Section 7]. NIST distinguishes between using a key for *protect* purposes when applying cryptographic armor (through encryption, wrapping, signing, and so on), and *process* purposes when consuming previously protected data (through decryption, unwrapping, verification, and so on). A key may at certain times be used for one or both purposes, or for none at all. The lifecycle of a cryptographic object progresses from a *Pre-Active* state, where it is not to be used for any cryptographic operation, through an *Active* state, where it may be used to protect and to process data, to a *Deactivated* state, where it may only be used to process data (see Figure 2). State transitions may be triggered directly by modifications to the lifecycle-relevant attributes, such as *state*, *activation time*, and *deactivation time*, or indirectly, as a side-effect of operations (e.g., when destroying an object). State transitions may cause actions by the automated deployment mechanism, as described in Section 3.

The operations of KLMS fall in two categories: those that manipulate objects, provided by KLS, and those that affect deployments, provided by Admin Service.

The most important operations on objects are: (1) *create*, which generates a new key or certificate and stores it, with attributes supplied by the client; (2) *store*, which stores a key or certificate and uses the cryptographic material supplied by the client in cleartext; (3) *import*, which stores a key and uses the cryptographic material supplied by the client in wrapped form (i.e., encrypted with another key); (4) *derive*, which creates a new symmetric key from an existing symmetric key; (5) *read*, which returns

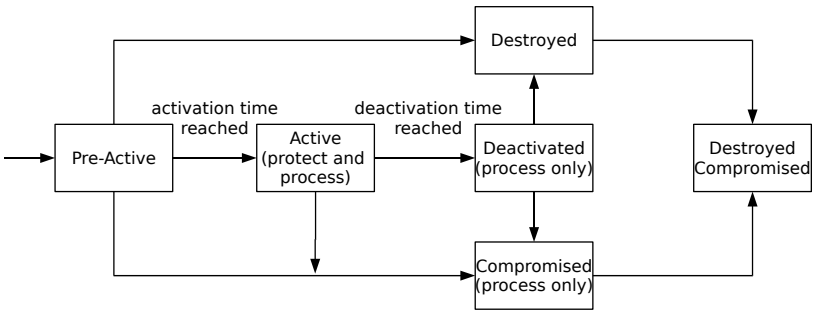

Figure 2: Key states and transitions. Transitions are triggered when an appropriate attribute is set or at the time specified by a time-related attribute.

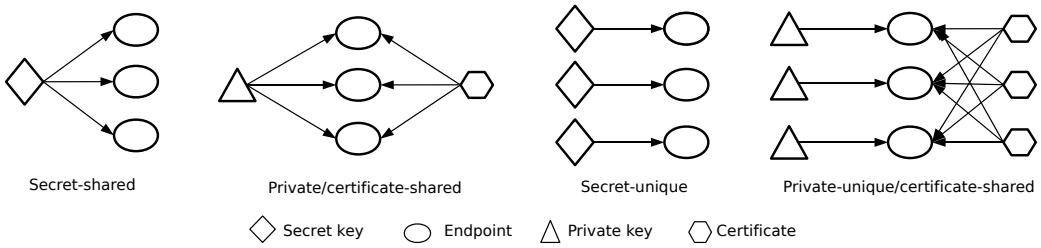

Figure 3: Deployment patterns.

the key or certificate with a given identifier to the client, including attributes and cryptographic material in cleartext; (6) *export*, which returns the key with a given identifier to the client, including attributes and cryptographic material in wrapped form; (7) *read attributes*, which is the same as *read*, except that it omits the cryptographic material; (8) *set attributes*, which modifies the attributes of an object; (9) *search*, which locates all objects matching a given search condition and returns their identifiers; (10) *destroy*, which deletes the cryptographic material of an object, but leaves its attributes intact; (11) *delete*, which deletes the entire object; (12) *archive*, for writing some objects to off-line storage; and (13) *recover*, for reading objects back from off-line storage.

The relevant operations of Admin Service on deployments are: (1) *specify*, which creates a deployment or a deployment bundle; (2) *activate*, which executes a deployment or a deployment bundle and distributes all objects to the specified endpoints; (3) *withdraw*, which reverses the effects of *activate* on a deployment or on a deployment bundle; and (4) *remove*, which removes a deployment or a deployment bundle from the system.

3 Automated Deployment

This section describes the pattern-based automated deployment in KLMS and the interaction between key-lifecycle management and key deployment. Recall that a deployment is an association between an object and an endpoint and that a deployment bundle is a set of deployments.

3.1 Deployment Patterns

A *deployment pattern* is a rule for generating deployment bundles with a defined structure. A deployment pattern is described in terms of an *object list*, an ordered set of keys and/or certificates to be deployed, and an *endpoint list*, an ordered set of endpoints, to which the objects are to be deployed. The pattern defines how the objects relate to the endpoints. Given an object list and an endpoint list, a deployment pattern yields a unique deployment bundle that complies with the pattern. Deployment patterns enable an administrator to focus on the requirements of an application, without having to worry about deploying individual keys and certificates.

We now describe four different deployment patterns, depicted in Figure 3.

Secret-shared: This pattern associates each element of the object list with every element of the endpoint list. For example, it is used to deploy a (set of) symmetric key(s) to multiple endpoints, such that they all share the same key(s).

When KLMS instantiates a *secret-shared* pattern, only the endpoint list is a mandatory input. The object list may be left out and a desired number n of symmetric keys can be given instead. In this case, KLMS generates n symmetric keys on the fly, taking their attributes from a template that is

also included, and deploys all keys to each endpoint. (The generated keys are also stored in the DB.)

Private/certificate-shared: This pattern associates each private-key/public-key pair or private-key/certificate pair in the object list with every element of the endpoint list. It is similar to the *secret-shared* pattern, but applies to asymmetric key pairs only (where public keys and certificates are used interchangeably).

A typical use-case for this pattern arises in a cluster of nodes that implement the same service, using replication and/or a fail-over strategy for increasing throughput and fault-tolerance. For example, when the cluster nodes serve web content over HTTPS for multiple domains, one private-key/certificate pair for TLS per domain must be available to every cluster node.

Secret-unique: This pattern associates the i-th key of the object list (containing only symmetric keys) with the i-th endpoint in the endpoint list, for $i = 1, \ldots, n$. When KLMS instantiates a *secret-unique* pattern, the object list can be omitted and a template describing attributes for the keys can be given; KLMS then generates as many keys automatically as there are endpoints in the list, and deploys them. This pattern can be used for generating unique master keys for a range of secure devices, each of which is identified by a symmetric master key.

Private-unique/certificate-shared: This pattern takes an object list of n asymmetric key pairs (i.e., private key/public key or private key/certificate pairs) and a list of n endpoints as inputs, and associates the i-th private key with the i-th endpoint, for $i = 1, \ldots, n$ and each public key/certificate with every endpoint. As with other patterns, when KLMS instantiates the pattern and the object set is omitted, then KLMS automatically generates the necessary key pairs from the attributes given in a template (such automatically generated certificates can only be self-signed).

This pattern addresses a typical infrastructure key-distribution model, where every entity is identified by a private key/public key pair, and every entity must know the public keys of all others. This could be a cluster of J2EE servers, whose communication is secured using those keys. Since the certificates are self-signed, all servers must have a copy of every other server's self-signed certificate in their keystore.

Note that the above list of four patterns is not exhaustive: one can define more patterns analogously, for example, a *private-unique/certificate-unique* pattern similar to *secret-unique*, by extending the generic association rules between object list and endpoint list above.

3.2 Administering Deployments

Recall that deployments are specified by an administrator using the Admin Service. Information on all objects deployed to endpoints is kept in a deployment table. Every deployment has a state that is either *OnHold* or *Active*.

When a deployment is in state *OnHold*, the deployment information is present in the deployment table, but the deployment should not yet or no longer take effect. Only during the time when a deployment is in *Active* state should the object be distributed to the endpoint and available to clients at the endpoint.

The administrator schedules the transition of a deployment from *OnHold* to *Active* state and vice versa by invoking the *activate* and *withdraw* operations of the Admin Service, respectively. After such a state change has been registered in the deployment table, it is the responsibility of a *distribution process* in DM to move or remove objects to or from the affected endpoints. Because its operations take time and may fail because of network failures, the distribution process operates asynchronously in the background. This design shields the administrator from the different semantics of the endpoints. When

DM distributes deployed objects to endpoints, it respects a *deployment policy* that affects its operation as described in Section 3.3.

In order to fully realize the power of automated deployment, the Admin Service allows dynamic modification of all pattern-based deployment bundles even after they have been created and activated. It is possible to add and to delete objects and endpoints to and from an existing deployment bundle. For example, when a deployment bundle d has been created by instantiating a pattern p, then an endpoint e can be added to it, and this will *specify* and *activate* a new deployment in d that affects e according to p. Likewise, when an endpoint e is deleted from d, this will *withdraw* and *remove* all deployments from d that contain e. Hence, the operator can manipulate deployments in a convenient way.

Note that a deployment of an object o to an endpoint e may be created through multiple ways, through individual deployments, deployment bundles, and pattern-based deployments. In this case, DM regards the deployment (o, e) to be in *Active* state whenever at least one of the sources is in *Active* state.

3.3 Deployment Policy

The policy followed by DM is called the *key-lifecycle deployment policy* and affects the behavior of the distribution process. The policy distinguishes *state-aware* endpoints that can interpret the *state* attribute of an object (such as those connecting through the KMIP interface) from *state-oblivious* endpoints that are not capable of expressing the notion of a lifecycle state (those connecting via VKS and JKS interfaces). The policy acts as follows. When DM distributes a deployment that associates an object o with an endpoint e, and when e is state-aware, then DM always distributes o to e; on the other hand, when e is state-oblivious, then DM only distributes o to e if o is in *Active* state. This ensures that a state-oblivious endpoint never uses a key in *Pre-Active* state for a cryptographic operation, as this undermines the idea of managing the lifecycle of keys.

In order to support this policy, DM includes a *key lifecycle scheduler*, which executes the time-triggered state changes that can be specified by setting certain object attributes, like *activation time* or *deactivation time*. For example, a key with an *activation time* in the future can already be deployed to a state-oblivious endpoint; but the key is not distributed to the endpoint until the *activation time* is reached, when the key lifecycle scheduler executes that action.

Moreover, the policy involves access control when the deployment specifies the push-style endpoint, as explained in Section 2.1.

Two screens from the GUI available to the administrator are shown in Appendix A. They show the interfaces for manual and automated deployment of symmetric keys, respectively.

4 Strict Access Control

The difficulty of enforcing strict access control comes from relations between keys, which are introduced through *wrapping* and *key derivation*, such that a simple ACL no longer adequately represents a permission on a key. This section explains the details behind our implementation in KLMS of the theoretical model for *strict* access control of [8].

In short, *strict* access control guarantees that a user may only retrieve the information she is authorized to, i.e., that she cannot abuse the API to violate the access control policy. To achieve this, traditional access control (with ACLs independent among different objects) is not sufficient, since the interdependencies among different keys, arising from *key wrapping* and *key derivation* operations, may open security holes in cryptographic APIs [2, 7, 9, 11].

KLMS server supports *key wrapping* (encrypting a target key under a different key, called wrapping key) through the *export* and *import* operations of KLS. To enforce strict access control, only key wrapping schemes are allowed that also provide strong integrity [8], such as authenticated symmetric-key

encryption [5] as realized by block-cipher encryption in *CCM* or *GCM* padding mode. When a wrapping in another format is supplied to an *import* operation, it must be treated like a *store* operation (with a cleartext key).

Through *key derivation*, a new symmetric key is generated from a parent key. Multiple keys derived from each other form a hierarchy, where knowledge of one key implies knowledge of all keys that are below in the hierarchy. Our system supports key derivation through the *derive* operation of KLS, and only key derivation functions with strong one-wayness are considered secure under the strict access-control policy.

Users. Recall from Section 2 that KLS distinguishes between different users who access it. For this section, a user u represents an entity that may invoke an operation of KLS, or is one of the special values `any` and `creator`. Users `creator` and `any` exist to simplify the formulation of ACLs. KLMS authenticates the user that invokes an operation and supplies the identity of the user to an operation as the *authenticated user*; it is different from `any` and `creator`.

Access-control policy. For most operations, KLS can take the access-control policy from the attributes of the affected object. But since KLS provides operations that do not refer to any existing object (e.g., *create*), it must take the access-control policy for those operations from attributes associated with the *user* that executes the operation. Therefore, there is a *user permission list* associated with every user u, denoted $u.$*UPL*, which is a set of permissions.

Permissions. *Permissions* are the elements that appear in the access-control list attribute of an object and in the user permission list. The permissions that may appear in an object ACL are: *Admin* (permits all operations), *Derive* (permits key derivation using the key as a parent key), *Destroy* (permits *destroy* and *delete* operations), *Export* (permits a key to be exported in a wrapped form), *Read* (permits reading the key in cleartext), *ReadAttributes* (permits gaining knowledge about key attributes), *Unwrap* (permits a key to be used for unwrapping in the *import* operation), and *Wrap* (permits a key to be used for wrapping in the *export* operation). The permissions that may appear in a user permission list (UPL) are *Create* and *Store* (for executing the respective operations).

Access-Control Lists. An *access-control list value* is an unordered set of pairs (u, p), where u is a user and p is a permission. An ACL attribute usually contains at least the entry $(\texttt{creator}, \textit{Admin})$.

4.1 Attributes for Access Control

We now describe the relevant attributes that govern the strict access-control policy.

Usage: The *usage* attribute determines in which cryptographic operations a key may be used by the endpoints. Its value is a subset of $\{\textit{Sign}, \textit{Verify}, \textit{Encrypt}, \textit{Decrypt}, \textit{Wrap}, \textit{Unwrap}, \textit{Derive}\}$. In our context, the only relevant function is to distinguish whether a key may be used for wrapping and unwrapping in *export* and *import* operations. Under strict access control, a symmetric key used for wrapping or unwrapping another key must *never* be used for a different purpose, as this may violate the specification of the interface [8]. Hence, every symmetric key must be used either for wrapping and/or unwrapping or for other cryptographic operations, like encryption, authentication, key derivation and so on. The strict policy enforces that these two kinds of key usage are mutually exclusive.

Digest: The server uses a *digest* attribute to avoid that the same cryptographic key exists as more than one object in its database. The *digest* is computed as the SHA-256 hash value of the cryptographic material, and must be available at least for all *symmetric key* objects and *private key* objects.

Strict: The *strict* attribute contains a boolean value and denotes whether the object falls under the strict access-control policy and benefits its guarantees. The attribute can only be *true* for symmetric keys and private keys. When it is *true*, it may be set to *false* by an explicit *set attributes* operation. Once the *strict* attribute is set to *false*, it cannot be set to *true* again.

ACL: The *ACL* attribute contains an access-control list value and denotes the ACL of the object. We assume that every *ACL* value satisfies the following invariant (the corresponding checks are omitted below):

1. if for some user u, an entry $(u, \mathit{Admin}) \in \mathit{ACL}$, then for every permission p, also $(u, p) \in \mathit{ACL}$; **and**
2. if for some user u, an entry $(u, \mathit{Export}) \in \mathit{ACL}$ or $(u, \mathit{Read}) \in \mathit{ACL}$, then also $(u, \mathit{ReadAttributes}) \in \mathit{ACL}$; **and**
3. if for some user u, an entry $(u, \mathit{Read}) \in \mathit{ACL}$, then also $(u, \mathit{Export}) \in \mathit{ACL}$.

The invariant ensures, first, that the *Admin* permission implies every other permission as well, second, that the *Export* and *Read* permissions imply permission to read the attributes, and, third, that the *Read* permission implies the *Export* permission.

Creator: The *creator* attribute contains a user that is different from `creator` and `any`. It denotes the authenticated user for the *create*, *store*, *import*, or *derive* operation that generated the object.

Dependents: The *dependents* attribute contains a set of strings that serve as *identifiers* for those objects whose cleartext value can be computed from knowledge of the cleartext value of the object itself (these are usually symmetric keys and private keys). It contains at least the identifier of the object itself.

Ancestors: The *ancestors* attribute contains a set of strings that serve as *identifiers* for those objects on which the given object depends, i.e., all objects whose *dependents* attribute contains the *identifier* of the given object. It contains at least the identifier of the object itself.

Readers: The *readers* attribute contains a set of users (except `creator`). It denotes those users who have, or may potentially have, obtained the cleartext value of the given object because they have either executed a *read* operation for the given object and obtained its cleartext value or because they have executed a *read* operation for another object and the *identifier* of the given object is contained in the *dependents* attribute of the other object.

4.2 Authorization of Operations

This section describes the authorization of operations by KLS and specifies the strict access-control policy. An operation is executed by an authenticated user u on an object o, which is named by its *identifier*. An attribute *attr* of o is denoted by $o.\mathit{attr}$. We use the following macro to express the basic access-control policy for user u, permission p, and object o:

$$\begin{aligned} \mathit{BASICAUTH}(u, p, o) \;= {} & (\texttt{any}, p) \in o.\mathit{ACL} \textbf{ or} \\ & \big(u = o.\mathit{creator} \textbf{ and } (\texttt{creator}, p) \in o.\mathit{ACL}\big) \textbf{ or} \\ & (u, p) \in o.\mathit{ACL}. \end{aligned}$$

In other words, *BASICAUTH* gives a permission p to user u for an operation o, if (a) any user has a permission p, or (b) if the creator of o has permission p and u is the creator of o, or (c) if ACL of o lists permission p for u.

Create. This operation generates a new key or certificate and adds it to KLS; it takes the cryptographic specification of the key as input, together with the attributes that govern the object. For certificates, KLS generates a certificate request and completes the *create* operation only later, when the certificate arrives from a certification authority. The condition *Create* $\in u.UPL$ must be true. After creating the object, the operation initializes the access-control-related attributes of all objects created during the operation as follows (we will refer to this algorithm again for the *store*, *import*, and *derive* operations).

1. It sets the *strict* attribute to *false*, unless another value is given; *strict* may only be *true* for *symmetric keys* and for *private keys*, but not for any other objects.

2. It sets the *ACL* attribute of all new objects to the value supplied with the operation. The *ACL* must contain at least the entry $(\texttt{creator}, \mathit{Admin})$.

3. For every new object o, it sets the *creator* attribute to the authenticated user, sets *dependents* and *ancestors* to the singleton set containing the *identifier* of o, and sets *readers* to the empty set.

The assumption that the default ACL gives *Admin* permission to the creator may be omitted when there exists a security officer or a server administrator outside the model, who can control the object.

Store. This operation initializes a new object in KLS with the cryptographic material of a key or a certificate given in cleartext, and with attribute values also supplied by the client, and stores the object. The following condition must be true:

1. *Store* $\in u.UPL$; **and**

2. there is no object already present at the server whose *digest* attribute is equal to the SHA-256 hash of the cryptographic material of the object being stored.

The operation initializes all access-control-related attributes of the stored object as described for the *create* operation, with the difference that it sets *strict* to *false*. This reflects the fact that the key value has not been generated in a controlled environment, which may lead to a violation of the strict access-control policy.

Import. Given a key supplied in wrapped form, this operation adds it to the server, with the attribute values taken from the wrapped representation. The operation specifies the *identifier* of the *unwrapping key* that KLS should use to unwrap the key.

The condition *Store* $\in u.UPL$ must be true for the operation to begin. If false, the operation aborts. Otherwise, the operation continues to check the following condition that determines if the unwrapping operation satisfies the strict access-control policy:

1. for the unwrapping key w, it holds: $\mathit{BASICAUTH}(u, \mathit{Unwrap}, w)$ **and** $w.\mathit{readers} = \emptyset$ **and** $w.\mathit{strict} = \mathit{true}$ **and** w is a *symmetric key* **and** $\{\mathit{Sign}, \mathit{Verify}, \mathit{Encrypt}, \mathit{Decrypt}, \mathit{Derive}\} \cap w.\mathit{usage} = \emptyset$ **and** $\mathit{Unwrap} \in w.\mathit{usage}$; **and**

2. there is no object already present at the server whose *digest* attribute is equal to the SHA-256 hash of the cryptographic material of the object being imported.

If true, the operation proceeds according to the strict policy. It unwraps the cryptographic material and the accompanying attribute values, uses them to initialize the object, and adds the object to KLS. Furthermore, the operation adds the *identifier* of o to the *dependents* attribute of w, and adds the set $w.\mathit{ancestors}$ to the *ancestors* attribute of o.

If the condition is not met, then the operation proceeds as for the *store* operation with a cleartext key. In particular, it sets the *strict* attribute of the imported object to *false*.

We remark that wrapping with public keys and unwrapping with private keys is not supported in strict mode, according to the model of Cachin and Chandran [8]. The reason is that encryption with a public key is not a secure wrapping method supporting the authentication of attributes; in fact, there is no authentication of the exported (wrapped) data at all, as everyone with the public key may create such a wrapping. Of course, one could require proper authentication of public-key-wrapped keys by a digital signature under the private key of the entity that exported and wrapped the key. But then, the exporter and the importer would each have to know the public key of the other one, and this is equivalent to the case where they would share a symmetric key. Hence, it is simpler for them to wrap with a symmetric key in this situation.

The condition that $w.\mathit{readers} = \emptyset$ under the strict policy means that no user in the model is supposed to have read the wrapping key w. Of course, some security officer or server administrator may have read the key in practice, but this occurs outside of the model, and that entity has to be trusted not to break the strict access-control policy.

Derive. A new symmetric key n is derived from an existing symmetric key o and some auxiliary input using a key derivation scheme, which essentially implements a pseudo-random function. Possible derivation schemes are PBKDF2 according to PKCS #5 (RFC 2898), HMAC (RFC 2104), encryption of the auxiliary data with a block cipher in CBC-mode, etc. The operation proceeds differently depending on the *strict* attribute of o.

If $o.\mathit{strict} = \mathit{false}$, the condition $\mathit{BASICAUTH}(u, \mathit{Derive}, o)$ must be true. In this case, the operation sets the *strict* attribute of n to *false*.

If $o.\mathit{strict} = \mathit{true}$, the operation checks the following condition implementing the strict access-control policy:

1. $\mathit{BASICAUTH}(u, \mathit{Derive}, o)$; **and**
2. $o.\mathit{usage} = \{\mathit{Derive}\}$.

The operation initializes the access-control-related attributes of the newly created object n as described for the *create* operation, with the following addition: If the $n.\mathit{strict}$ is *true*, then it adds the *identifier* of n to the *dependents* attribute of every object q whose *identifier* is in $o.\mathit{ancestors}$, and adds the set $o.\mathit{ancestors}$ to the *ancestors* attribute of n.

Read. Given the *identifier* of an object, this operation returns the attributes of the object and its cryptographic material in cleartext. The following condition must be true:

> $o.\mathit{strict} = \mathit{false}$ **and** $\mathit{BASICAUTH}(u, \mathit{Read}, o)$; **or**
>
> $o.\mathit{strict} = \mathit{true}$ **and** for all objects k such that the *identifier* attribute of k is in $o.\mathit{dependents}$, it holds $\mathit{BASICAUTH}(u, \mathit{Read}, k)$.

If the operation proceeds and $o.\mathit{strict} = \mathit{true}$, then it adds u to the *readers* attribute of all k as quantified in the condition.

Export. The *export* operation creates an external representation of an object o specified by an *identifier* and wrapped with a symmetric key w, suitable for transfer to another key-management system. KLS supports a number of key wrapping methods, but defines its own format for representing attribute values in the wrapped representation. As mentioned before, only wrapping schemes with strong integrity are suitable for use under the strict policy.

The operation proceeds differently depending on the *strict* attribute of o. If $o.\mathit{strict} = \mathit{false}$, only the condition $\mathit{BASICAUTH}(u, \mathit{Read}, o)$ must be true. As the operation uses only basic access control, it does not involve any extra checks.

If $o.strict = true$, the operation checks the following condition implementing the strict access-control policy:

1. *BASICAUTH*$(u, Export, o)$ **and** *BASICAUTH*$(u, Wrap, w)$; **and**
2. for each user v in $w.readers$ and for each object q whose *identifier* is in $o.dependents$, it holds *BASICAUTH*$(v, Read, q)$; **and**
3. w is a *symmetric key* **and** $w.strict = true$ **and** $w.identifier \notin o.dependents$; **and**
4. $\{Sign, Verify, Encrypt, Decrypt, Derive\} \cap w.usage = \emptyset$ **and** $Wrap \in w.usage$.

If the condition holds, then the operation adds the *identifier* of o to the *dependents* attribute of w, and adds the *identifier* of w to the *ancestors* attribute of o. All attributes of o are included together with the wrapped key in the external representation, in particular all access-control-related attributes mentioned in Section 4.1.

Note that by our assumption on ACL values, every user with *Read* permission also has *Export* permission. Hence, the *Export* permission is weaker than *Read*, but only applies to objects that fall under the strict access-control policy. For objects under basic access control, the *Read* permission is needed for the *export* operation.

Read Attributes. The operation returns all attributes of an object o specified by its identifier, but not the cryptographic material of o. The condition *BASICAUTH*$(u, ReadAttributes, o)$ must be true.

Set Attributes. The operation sets all attributes of an object o specified by its identifier. The condition *BASICAUTH*$(u, Admin, o)$ must be true.

A client cannot modify any of the following attributes using the *set attribute* operation: *identifier*, *digest*, *creator*, *dependents*, *ancestors*, and *readers*. Moreover, no operation may set the *strict* attribute to *true*.

When a *set attribute* operation changes the *ACL* attribute of o to a value *ACL'*, the following condition must be true:

1. for all $(v, Read) \in ACL'$ with $v \neq \texttt{creator}$ and for each object $q \neq o$ whose *identifier* is in $o.dependents$, it holds *BASICAUTH*$(v, Read, q)$; **and**
2. for all $(\texttt{creator}, Read) \in ACL'$ and for each object $q \neq o$ whose *identifier* is in $o.dependents$, it holds *BASICAUTH*$(o.creator, Read, q)$.

In other words, before *Read* permission for o can be given to some user, the user must also be permitted to *read* all keys that depend on o.

Note that according to this specification, only a user with *Admin* permission who has complete control over an object, including reading its cleartext value, may trigger life-cycle state changes. One could instead introduce a separation of the *Admin* permission into one for proper *administration* that allows all operations (as for the *Admin* permission above) and one for *life-cycle administration* that only allows modification to the life-cycle related attributes, such as *state* and the related time values.

Search. The *search* operation returns identifiers of all objects matching a given search condition on attribute values. A user may invoke this operation without any particular permission. But since the presence of an object in the reply discloses information about the attributes of the object, the user must at least have permission to read the attributes of the returned objects.

Hence, the condition *BASICAUTH*$(u, ReadAttributes, o)$ must be true for every object o whose *identifier* is returned in the reply from *search*.

Archive and Recover. The *archive* and *recover* operations save and read back an object to and from off-line storage. The condition *BASICAUTH*(u, Admin, o) must be true. The attributes of o must remain accessible online, however, in order to be used in decisions on strict access control.

Destroy and Delete. The *destroy* operation deletes the cryptographic material of an object o and leaves its attributes intact; the *delete* operation removes the entire object from KLS. For either one, the condition *BASICAUTH*$(u, \mathit{Destroy}, o)$ must be true.

5 Implementation

Our implementation of the KLMS server, together with the prototype support for KMIP, measures over 70k lines of Java code. Out of this, the core of the server (below the Interface layer) takes roughly 20k lines of Java code, with automated deployment taking slightly over 5.5k lines, and strict access control support around 1.5k lines. Currently, KLMS supports the four automated deployment patterns presented in Section 3.1, yet additional patterns can be added in a modular manner. The implementation of strict access control takes into account the possible size of the object attributes *dependents* and *readers*; these grow with the system's age and may pose performance issues if implemented sub-optimally. To cope with this, our implementation uses two separate global tables in the DB layer for these two attributes. For the *readers* table, as for the representation of *ACL*, DB maintains the identity of a user determined from an LDAP directory server in the form of the string representation of the user's LDAP Distinguished Name.

KLMS support for hardware-security modules (HSM) is foreseen by the architecture, but has not been implemented yet. Currently, the DB layer is based on a small-footprint Apache Derby database. The experimental integration of support for KMIP includes the portable portion of KMIP client/server code (around 28.5k lines of Java code) and the KMIP/KLMS adapter code (slightly over 4k lines). With this architecture, the support for KMIP can be easily transferred to a different key server core.

6 Evaluation

6.1 Automated Deployment

We compare automated pattern-based deployment with the KLMS GUI to manual deployment operations in a theoretical usability study. The two main screens in the administrator interface of the GUI for configuring manual and automated key deployment are shown in Figures 5 and 6, respectively (Appendix A).

It is obvious that manual deployment is not only more error-prone but also less efficient than automated deployment for handling a large number of keys. To be concrete, we examine the two deployment methods in a realistic scenario, where n private keys and the corresponding certificates are deployed to n endpoints such that every private key is deployed to one endpoint and all certificates are deployed to all endpoints. This corresponds to the *private-unique/certificate-shared* pattern.

We count the number of steps (user interactions in the form of mouse clicks and keyboard inputs) that the administrator has to perform for accomplishing the deployment task using either method.

With both methods, the administrator needs five interactions initially for navigating to the deployment screen in order to create and select the appropriate deployment method. Once the interface is set up like this, the administrator creates the deployments.

For manual deployment, it takes $4n(n+1)$ steps to create the described deployment. The administrator selects an "Add" dialogue for every deployment, choses an object and an endpoint to include in the deployment, and submits the operation. The administrator must repeat this these operations $n(n+1)$

times to deploy each private key to one endpoint and every one of the n certificates to all n endpoints. Finally, activating the deployment requires one additional user interaction. In total, the administrator needs $6 + 4n(n + 1)$ steps in manual deployment.

For automated deployment, the interface requires $4 + n$ user interactions to create the described deployment. After the administrator has selected the "Add" dialogue, it must select a template for automatically creating the required number of keys and check the option to "generate keys automatically." Furthermore, the administrator must select the n endpoints. To activate the deployment, one additional step is necessary. In total, $10 + n$ user interactions are necessary with automated deployment.

Table 2 summarizes these results and demonstrates that manual deployment requires $O(n^2)$ user interactions and automated deployment requires $O(n)$ user interactions for the selected task. It is clear automated pattern-based deployment scales much better than manual deployment.

	Manual deployment	Automated deployment
Setup	5	5
Configuration	$4n(n + 1)$	$4 + n$
Deployment	1	1
Total	$O(n^2)$	$O(n)$

Table 2: Number of user interactions for deployment.

6.2 Performance of Strict Access Control

We have measured the performance difference between operations under the strict access-control policy and operations under the basic access-control policy, to determine the cost of the additional protection.

System setup. The benchmarking server is an IBM x345/6 system with two hyper-threaded Intel® Xeon™ CPUs running at 3.06GHz and 2GB RAM. It runs our Java prototype of KLMS on an IBM J9 JVM (build 2.3, J2RE 1.5.0) with Apache Derby 10.3 as the database.

Experiments. The first experiment consists of running 100 *create* operations and measuring the elapsed time or latency. The operations are executed and measured at the service layer (in KLS). The created keys are then used to separately measure 100 *search*, 100 *read* and 100 *delete* operations. Because each operation takes only a few milliseconds, measurements are done over 100 operations. Each measurement was also performed 100 times (i.e. a total of 100×100 operations each, for strict and for basic access control). Table 3 summarizes the average time taken by one operation.

Our second experiment measures the scalability of the implementation of the strict access-control policy in KLMS, when a tree of dependencies grows deeper. The experiment starts by creating a new key and deriving from it a linear hierarchy of 10 keys, yielding 11 keys in total. We measure the time taken by the *derive* operation to derive one key, depending on the depth where this occurs. We chose maximal depth 10 because it is the maximum depth allowed by certain key-management products; in particular, the IBM CCA interface [15] allows maximal derivation depth 10. As the latencies are small (a few dozens of milliseconds), each data point is obtained by taking an average over 100 identical successive operations, and a total of 100 data points are collected per derivation level ($10 \times 100 \times 100$ derivations in total). Then the ACL of every key in the hierarchy is modified by adding the *read* permission for a fixed user, starting from the deepest key in the hierarchy up to the initial key. We measure the time taken by the *set attributes* operation. The measurements are done as previously for the *derive* operation.

Operation	Policy	Average latency	95%-confidence interval
Create	basic	4.81	[4.44; 5.19]
	strict	5.10	[4.69; 5.51]
Search	basic	2.50	[2.39; 2.61]
	strict	2.61	[2.50; 2.72]
Read	basic	2.64	[2.60; 2.69]
	strict	3.75	[3.67; 3.83]
Delete	basic	9.38	[8.97; 9.79]
	strict	9.97	[9.49; 10.44]

Table 3: The table shows the average times taken by four representative operations and the corresponding 95%-confidence intervals, in milliseconds.

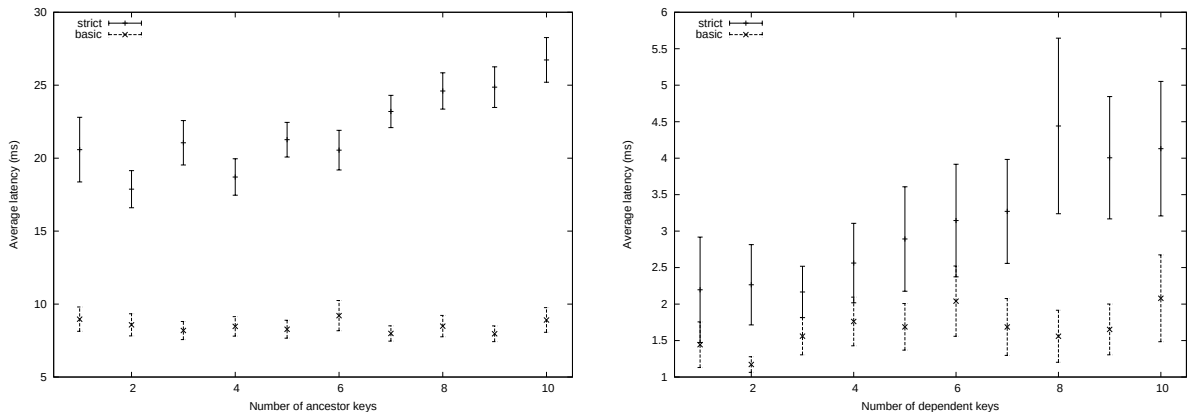

Figure 4: The graphs show the average times for the *derive* operations (left) and the *set attributes* operations (right), for varying tree depth.

Results. Table 3 shows the overhead of the strict access-control policy for the four operations in the first experiment. One can see that most operations perform comparably except for *read*, which takes on average about 41% longer with strict access control. The reason is that, unlike with basic access control, the strict policy requires the *read* operation to modify the *Readers* attribute of an object. This explains the extra time taken by the operation.

Figure 4 shows the results of the second experiment. The measurements demonstrate that the strict policy scales well with the depth of the derivation tree for the *derive* operation, showing, roughly, a constant twofold overhead with respect to basic access control, with a slight increase for derivation trees deeper than six levels. For the *set attributes* operation, we observe a increasing overhead for modifying the ACL of those keys that have six or more dependent keys.

Acknowledgements

We thank Gordon Arnold, Todd Arnold, Tim Hahn, Ilias Iliadis, Glen Jaquette, Tony Nadalin, John Peck, Roman Pletka, Bruce Rich, and Krishna Yellepeddy for interesting discussions and valuable comments.

A Graphical User Interface

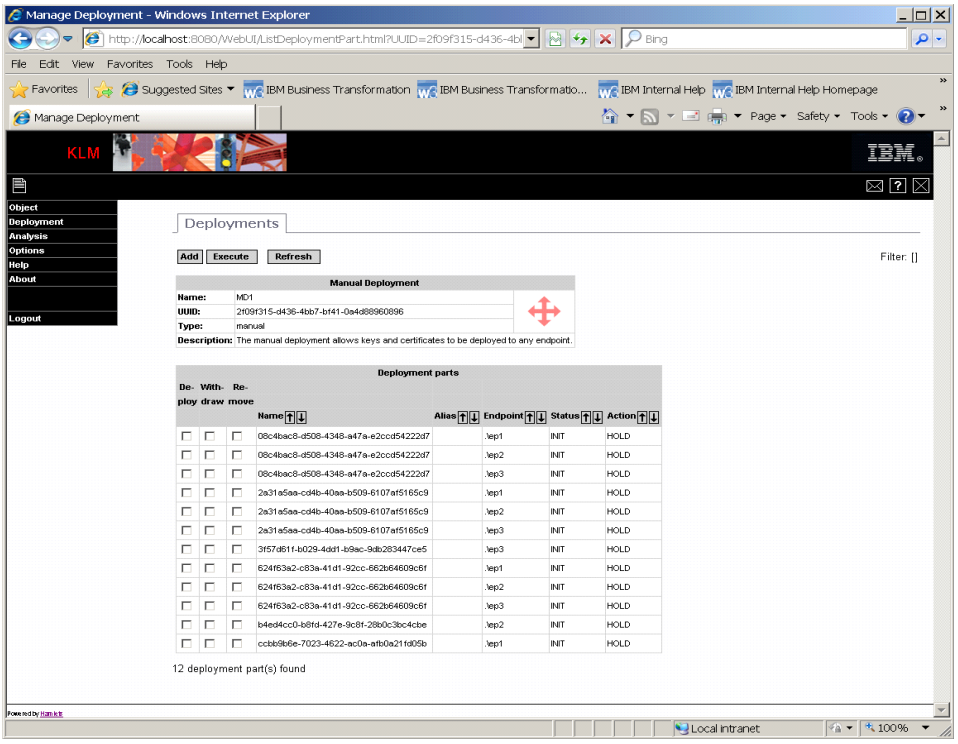

Figure 5: Screen for manual deployment in the KLMS server.

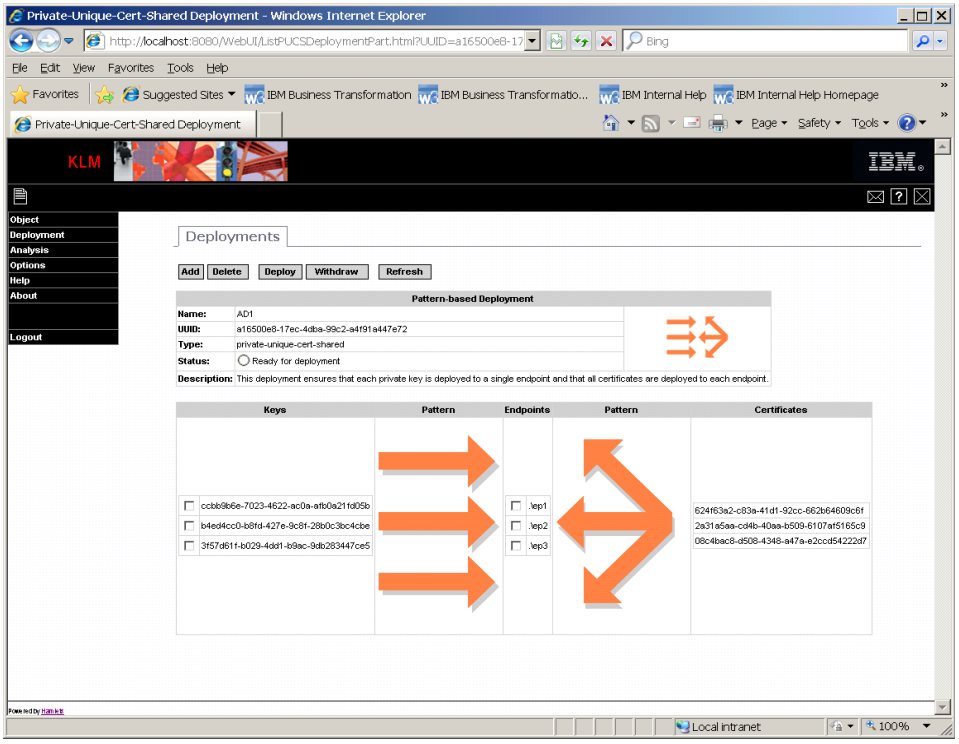

Figure 6: Screen for automated deployment in the KLMS server.

References

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

[2] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic processors — a survey," *Proceedings of the IEEE*, vol. 94, pp. 357–369, Feb. 2006.

[3] W. Arnold, T. Eilam, M. H. Kalantar, A. V. Konstantinou, and A. Totok, "Pattern based SOA deployment," in *Proc. Service-Oriented Computing (ICSOC)* (B. J. Krämer, K.-J. Lin, and P. Narasimhan, eds.), vol. 4749 of *Lecture Notes in Computer Science*, pp. 1–12, 2007.

[4] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management," NIST special publication 800-57, National Institute of Standards and Technology (NIST), 2007. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[5] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology: ASIACRYPT 2000* (T. Okamoto, ed.), vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, 2000.

[6] BITS Security Working Group, "Enterprise key management." Whitepaper, BITS Financial Services Roundtable, available from `http://www.bits.org/downloads/Publications%20Page/BITSEnterpriseKeyManagementMay2008.pdf`, May 2008.

[7] M. Bond, "Attacks on cryptoprocessor transaction sets," in *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 220–234, 2001.

[8] C. Cachin and N. Chandran, "A secure cryptographic token interface," in *Proc. Computer Security Foundations Symposium (CSF-22)*, IEEE, July 2009. To appear.

[9] J. Clulow, "On the security of PKCS#11," in *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 411–425, 2003.

[10] "Cover pages: Cryptographic key management." `http://xml.coverpages.org/keyManagement.html`, Apr. 2009.

[11] S. Delaune, S. Kremer, and G. Steel, "Formal analysis of PKCS#11," in *Proc. 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008.

[12] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor," *IEEE Computer*, vol. 34, pp. 57–66, Oct. 2001.

[13] "Hamlets." `http://hamlets.sourceforge.net`.

[14] IEEE Security in Storage Working Group (SISWG), "P1619.3/D6 draft standard for key management infrastructure for cryptographic protection of stored data." Available from `https://siswg.net/index.php`, 2009.

[15] International Business Machines Corp., *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, 19th ed., Sept. 2008. Available from `http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml`.

[16] OASIS Key Management Interoperability Protocol Technical Committee, "Key Management Interoperability Protocol," Apr. 2009. Editor's draft 0.98; available from `http://www.oasis-open.org/committees/documents.php?wg_abbrev=kmip`.

[17] RSA Laboratories, "PKCS #11 v2.20: Cryptographic Token Interface Standard." Available from http://www.rsa.com/rsalabs/, 2004.

[18] Trusted Computing Group, "Trusted platform module specifications." Available from http://www.trustedcomputinggroup.org, 2008.

[19] World Wide Web Consortium, XML Key Management Working Group, "XML Key Management Specification (XKMS 2.0)." Available from http://www.w3.org/2001/XKMS/, 2005.