XACML Based Access Control For Key Management With OASIS KMIP

Master Thesis

Divay Bansal
bansaldi@student.ethz.ch

ETH Supervisor
Prof. Dr. David Basin
basin@inf.ethz.ch

IBM Supervisors
Dr. Guenter Karjoth
gka@zurich.ibm.com
Dr. Christian Cachin
cca@zurich.ibm.com

Department of Computer Science
ETH Zurich

November, 2010 - May, 2011
Abstract

Key Management systems are widely used for managing the life cycle of cryptographic keys. These systems work in a distributed client-server environment, providing services such as creation, distribution, derivation, and certification of keys. Keys are highly valuable secrets for an organization. A key management system should provide an efficient access control mechanism to protect the integrity of the stored keys and prevent any possible attacks.

This thesis addresses access control for a key management system. We present a novel approach for implementing the access control policies in XACML. We consider the KMIP specification for building our access control model and discuss the semantics of various KMIP operations and their side effects. Based on the above knowledge, we design our policies using a logic-based Authorization Specification Language. We discuss the strict form of access control in key management context and propose the design considerations for implementing such policies in XACML. We present a proof of completeness and consistency for our policies.

In the second part of the thesis, these policies are implemented in XACML. We present a semantic-based translation from ASL primitives into XACML. In addition, we present a service-oriented architecture for implementing the authorization mechanism. We discuss the design considerations of components involved in the authorization mechanism and its integration with KMIP prototype built at IBM Research. To give an idea regarding the performance we provide experimental results of load tests performed on our solution.

Keywords: Access Control, Key Management, KMIP, Strict Access Control, XACML.
Acknowledgment

I would like to offer my sincere gratitude to Dr. Christian Cachin and Dr. Guenter Karjoth who have guided me through every aspect of this thesis work. I am grateful for their encouragement, knowledge and time for long discussions, which helped me immensely in solving the problems addressed in this thesis. It was a wonderful learning experience that has given me knowledge and skills required for solving complex problems and writing scientific reports.

I would like to thank Prof. Dr. David Basin for his valuable suggestions and time during the course of this thesis. The suggestions helped me into looking at the problem from different perspectives and thinking about different possible solutions.

I am thankful to Dr. Robert Haas and Mathias Bjorkqvist form IBM Research Zurich for their guidance in understanding KMIP specification and its prototype built at the lab.

I finish my master’s program at ETH Zurich with this thesis. I am grateful to ETH for providing me with financial assistance to support my studies and stay in Switzerland. I have gained immense knowledge in different aspects of computer sciences and life in general during my stay at ETH.

I would like to use this opportunity to thank my parents for their love, support and motivation.
# Contents

1 Introduction .......................... 1
   1.1 Motivation .................................. 1
   1.2 Goals, Design Considerations And Our Contributions .......... 3

2 Background .............................. 5
   2.1 Key Management Interoperability Protocol ......................... 5
   2.2 Authorization Specification Language ............................... 7
   2.3 Extensible Access Control Markup Language ....................... 9
      2.3.1 XACML Policy Model .................................. 10
      2.3.2 XACML Request/Response Model ............................ 13
      2.3.3 XACML Data Flow Model ............................... 14

3 KMIP Access Control Model ............. 16
   3.1 KMIP Access Control Data Model ................................ 16
      3.1.1 Subject .......................................... 16
      3.1.2 Resource ........................................ 16
      3.1.3 Action ........................................... 18
      3.1.4 Access Request ................................... 20
      3.1.5 Access Control Policy ................................ 20
   3.2 KMIP Access Control Policy Design ................................ 20
      3.2.1 Authorization Specification Language extension for KMIP .... 20
      3.2.2 KMIP Server Functionality and Authorization Policies ....... 21
      3.2.3 KMIP Service Semantics and Access Control Policies ....... 22
   3.3 KMIP Strict Access Control Policy Design ......................... 28
      3.3.1 ASL Primitive Extension for Strict Access Control .......... 29
      3.3.2 KMIP Services and Strict Access Control Policies .......... 29
   3.4 Completeness and Consistency of KMIP Policies .................. 31

4 KMIP XACML Policy Implementation ..... 34
   4.1 Translating ASL Primitives Into XACML Policy Components ....... 34
      4.1.1 ASL Constant Symbols .................................. 34
      4.1.2 ASL Predicate Symbols .................................. 34
      4.1.3 ASL Rules ............................................ 36
   4.2 KMIP XACML Policy Hierarchy ................................... 42
   4.3 KMIP XACML Data Flow Model ................................... 43
5 System Design and Implementation
   5.1 System Architecture ........................................... 45
   5.2 Sequence Diagram ............................................. 47
   5.3 Performance Evaluation ....................................... 49
       5.3.1 Test Bed ................................................. 50
       5.3.2 Experimental Results ................................... 50
       5.3.3 Discussion ............................................... 53
   5.4 Related Work .................................................. 54
       5.4.1 Comparison With Related Work .......................... 54

6 Conclusion and Future Work ....................................... 56

A KMIP Default Operation Policy .................................... 58
   A.1 Operations Outside of Operation Policy ...................... 58

B KMIP XACML Policies ............................................... 65
# List of Figures

1.1 Key Management System ........................................... 2
2.1 KMIP Architecture .................................................. 6
2.2 XACML Policy Model ................................................. 11
2.3 XACML Request Model ............................................... 14
2.4 XACML Data Flow Model ........................................... 15
3.1 KMIP Resource Generalization Hierarchy ......................... 17
3.2 Derived Hierarchical Key Relation ................................. 29
4.1 KMIP Action Hierarchy ............................................... 35
4.2 KMIP XACML Policy Implementation ............................... 38
4.3 ASL Primitives to XACML Components Translation ............... 41
4.4 KMIP XACML Hierarchical Policy Model Diagram ................. 42
4.5 KMIP Authorization Request Evaluation Using XACML ............ 44
5.1 KMIP XACML Data Flow Architecture .............................. 45
5.2 PEP XML Access Request Schema ................................ 47
5.3 Sequence Diagram .................................................. 48
5.4 Latency Vs Load ..................................................... 51
5.5 Throughput Vs Load ................................................ 52
5.6 Indirect Operation Performance .................................. 52
5.7 XACML Engine Performance ....................................... 53
List of Tables

3.1 Managed Objects and Attributes Mapping .............................................. 18
3.2 Operation on Managed Objects ............................................................. 19
5.1 Pros/Cons of Access Control Integration Designs ................................. 46
5.2 Operation Components ................................................................. 53
A.1 Operation on Secret Managed Objects .................................................. 59
A.2 Operation on Public Managed Objects .................................................. 61
A.3 Default Operation Policy for Private Template Objects .......................... 63
A.4 Default Operation Policy for Public Template Objects ............................ 64
Chapter 1

Introduction

1.1 Motivation

A key management system provides infrastructure for secure generation, storage, management, distribution and destruction of cryptographic material. A key management system, as shown in Figure 1.1, is designed to work in a distributed client server environment. The clients access the key management server using a secure protocol and request to perform various operations on their cryptographic material managed by the server. The clients in a key management system can be categorized as follows:

- devices such as storage tapes which can generate encryption keys at the server to encrypt their content.
- electronic applications which need to provide secure communication channels, transactions or privacy services.
- users or enterprises who can generate private-public key pairs, store private keys for digitally signing their emails, distribute public keys with certificates, which are used to verify signatures and receive emails encrypted using their public keys.

The cryptographic material in a key management system mainly consists of public keys, private keys, symmetric keys, split keys, or certificates. The cryptographic keys are used to provide security services such as: confidentiality, data integrity, authentication, privacy and non-repudiation. The cryptographic keys are highly valuable secrets that need to be safeguarded from any unauthorized access to prevent the information protected by them from being disclosed.

A key management system manages the lifecycle of the stored cryptographic keys by performing create, register, destroy, revoke, and archive operations. These operations affect the attributes of the cryptographic keys having effect on their usage and the relation to other keys. In addition to managing the cryptographic keys, key management systems also perform cryptographic operations such as:

- certification of a public key.
- wrap or unwrap a key with another key to securely install it at the client endpoint.
- deriving keys from a symmetric key to be used by multiple clients.
- performing secure backups (escrow) or recovery for the various keys.
The above cryptographic operations change the access constraints of involved keys by creating complex relations among keys. For example, derive operation create dependency among keys where a parent symmetric key is used to create a new symmetric key using a key derivation scheme, which implements a pseudo-random function. This dependency relation creates a hierarchical structure among the keys. The owner of a parent key can regenerate a derived key by applying various key derivation schemes on his key even if access is not granted to the derived key.

The access control policies must also take into account the integrity of the involved entities for deciding authorization. The integrity constraints involve considering the history of accesses and verifying that the involved resources including their dependency relations has not been compromised by any API attacks \[5,6\]. The security policies must consider the strict form of access control for keys taking in to account the semantics of the key management operations and the integrity constraints. For example, weak security policies for key management system might allow an unauthorized user $U$ to import a key $K$, due to $K$’s dependency on $U$’s key $L$, created by the above cryptographic operations. In another example, if protection policies do not consider the relation between wrapped keys, then an unauthorized user can import another key by wrapping it with his own key. Hence, the access control in a key management system must consider the relation among the keys while deciding about granting the access.

An access control system ensures that only authorized operations get executed. It acts as a reference monitor mediating access request by a user to resources inside the system and decides whether the access request should be granted or denied using the system security policy. An access control system is considered to be composed of two elements: security policies and access control mechanism. A security policy represents high level guidelines that determine how access is controlled and access decisions are determined \[15\]. The access control mechanism is a software program which implements the policies inside a system.

The access control systems are based on the distinction between subjects and objects. Actions are performed on behalf of subjects in accordance with authorizations established in the system. Authorizations are access rights which can be specified as rules. A rule can be defined as a tuple containing subject, resource and action. It specifies which operations can be performed
on an object by which subjects. An access request is granted if it matches the rules that satisfies
the subject and object conditions for access. A basic form of access control system for a key
management system will decide access for a request based on security policy, which specifies:

- any authenticated user can access the public keys and certificates
- owners are allowed to access the private keys, symmetric keys, split keys and secret data.

This access control system restricts access to keys to privileged entities. However, the system
does not consider the complex relation between keys created by cryptographic operations such
as derivation, wrapping etc., adding more constraints to a access control policy. In certain sce-
narios, a user will like to delegate his ownership of a key to other users to perform cryptographic
operations on his behalf. We can not reason about joint authority, delegation of authority, roles
in the above access control system. The access control policy does not take in to account the
dynamic behavior of keys in terms of their states. For example, a key which has been revoked
can be recovered and re-activated. This behavior raises the problem of dealing with the authori-
izations specified for a key and whether specified authorizations should be considered valid if
the key is modified, recovered or its state changes.

### 1.2 Goals, Design Considerations And Our Contributions

In addition to Samarati et al. [14], our work focuses on building stateful access control policies
which takes dependency relation among keys and access history in to account. For example, the
policy should specify that all the users who have used a wrapping key $W$ in the past must have
read access to the dependent keys of the key $K$ requested for exporting using $W$. This access rule
of propagating privileges among the keys, if not satisfied, can lead to unauthorized access. We
use OASIS Key Management Interoperability Protocol (KMIP) [13] framework for designing
our access control model for key management system.

We use the Authorization Specification Language (ASL) [10] to design our authorization
policies and XACML [13] to implement those policies. ASL is a logic-based language, which
provides expressibility to express authorizations on subjects and keys using a conjunction of
its symbols. The symbols represent the rules for authorizations, derived authorizations, conflict
resolutions and access decisions. The rules provide flexibility of specifying authorizations either
at an abstract level in terms of groups, roles, resource types or at a specialized level in terms of
single resource and a user. We can specify the proofs for correctness and consistency of a policy
formed by the combination of multiple policies using ASL model. In addition, we develop a
mechanism for translating KMIP ASL policies in to XACML.

Using XACML we can separate authorization mechanisms from authorization policies and
server functionality in a distributed environment. This way, changed protection requirements
can be easily incorporated involving only the policy change. Also, the XACML policies can be
easily distributed, combined and evaluated by various authorization mechanism.

We designed and implemented XACML based policies for KMIP server, which is a part of
IBM Trivoli Key Lifecycle Manager (TKLM) product. The current implementation of autho-
ration mechanism in TKLM involves tight coupling with server functionality, which makes
it difficult to change the authorization policies. In a distributed environment, an authorization
mechanism must be able to handle multiple policies based on the requirement of different en-
terprises. This condition can be satisfied if the authorization mechanism is independent of the
policies and is separate from server functionality. We propose a service oriented architecture for
This thesis report can be summarized as follows. Chapter 2 provides some background on KMIP, ASL and XACML. It gives an overview of the various features of KMIP system, discusses the various constructs of ASL model and introduces the syntax and semantics of various elements of XACML model. In Chapter 3, we use the above knowledge to design access control policies for KMIP system. To design the policies for KMIP, we first define its various elements and their relations by building a KMIP Data Model. We formalize the complex relations or constraints related to authorizations for various elements of KMIP Data Model as additional primitives in ASL. Using ASL symbols we model authorization policies for various KMIP operation considering their inputs, outputs and side effects, for different objects and users. We define the notion of strict access control for key management systems and design the access control policies for corresponding KMIP operations. At the end of this section, we develop the ground rules for combining the various policies to make our policies complete and consistent. In Chapter 4 we provide translation of the ASL symbols into XACML components, which is used to implement the policies in XACML. In Chapter 5 we discuss the authorization mechanism architecture, design considerations and its integration with the KMIP prototype built at IBM Research. We discuss the performance evaluation criteria of the design and provide experimental results. Using the evaluation results we find the performance bottleneck in the system and look into the optimization techniques. We compare our approach with the related work discussed in Section 5.4. In Chapter 6, we conclude by providing a summary of our work and a discussion on problems that can be a part of the future work.
Chapter 2

Background

2.1 Key Management Interoperability Protocol

The Key Management Interoperability Protocol (KMIP) [8] is an OASIS standard, which defines the communication between clients and a key management server to perform various key management operations on the cryptographic objects managed by the server. KMIP classifies objects in a key management system into base objects and managed objects. Base objects deal with communication between client and server. Managed objects represent the cryptographic objects, templates for creating those objects and their meta data. The cryptographic objects are of type symmetric keys, private keys, public keys, certificates, secret data, split keys and opaque objects.

KMIP defines a set of attributes such as unique identifier, usage, owner that are used for managing the cryptographic objects. For example, unique identifier distinguish each cryptographic object globally and can be used to reference the object. Each cryptographic object is associated with a subset of these attributes. KMIP defines the data structure, format and corresponding legal values for these attributes. KMIP also specifies who can read, modify and delete these attributes.

KMIP defines the various states representing the life cycle of a cryptographic object. A cryptographic object in a key management system shall be in one of the following states at any given time.

1. **Pre-Active**: A managed cryptographic object is in Pre-Active state when it is created and not yet usable.

2. **Active**: A managed cryptographic object is in Active state when it is being used for cryptographic purposes. For example, a symmetric key is active when it is being used for performing encryption or decryption.

3. **Deactivated**: A managed cryptographic object in Deactivated state can not be used for performing cryptographic protection operation such as encryption, signing but may be used for to perform operations involving processing cryptographically protected information such as decryption, verification.

4. **Compromised**: A managed cryptographic object is in Compromised state when it has been compromised and can only be used to process cryptographically protected information by a trusted client.

5. **Destroyed**: A managed cryptographic object is in Destroyed state when it has been destroyed and is no longer usable. When object is destroyed its meta data is kept in the key...
management system since other objects might be dependent on the destroyed object.

6. **Destroyed Compromised:** The object is no longer usable for any purpose but the compromised status may be used for auditing and security purposes.

KMIP defines a set of operations dealing with the *lifecycle* of the cryptographic objects such as *Create, Register, Destroy, Archive, Revoke, Activate, Deactivate and Recover*. In addition to key management operations, KMIP defines a set of operations to search and retrieve cryptographic objects and their attributes, which are used to distribute the objects to various clients. KMIP supports a set of cryptographic operations that include certification of a public key, derivation of a symmetric key from an existing key and wrapping the keys to securely distribute them to various clients or end points connected to the key management server.

Based on research prototype built by IBM Research Zurich, KMIP is implemented as a client server architecture as shown in Figure 2.1. At client side, a KMIP proxy is deployed, which has a layered architecture containing a KMIP API layer, encoding-decoding layer and a transport layer. To perform an operation at the KMIP server, client creates a request using KMIP API containing his credentials, operation he wants to perform and the corresponding resource parameters. The request is encoded to make it secure by the encoding-decoding layer and forwarded to the transport layer. Transport layer provides functionality for sending the request to the KMIP server over TCP/IP.

At the server side, KMIP receives the request from the Tomcat container and forwards it to the encoding-decoding layer to decode the message from the request. The request is forwarded to a dispatcher layer, which first authenticates the user using his KMIP credentials from the request and after authentication forward the request to the corresponding key management operation defined. The KMS Adapter layer provides functionality for integrating the different implementation of a KMIP operation. The response of the operation is decoded and send back to the client.

![Figure 2.1: KMIP Architecture](image)

The specification of KMIP divides the managed objects in to three domains i.e Secret Objects, Public Objects and Template Objects for access control. KMIP defines default operation
policies for each of the three domains. Each operation policy is associated with every managed object in the KMIP system depending on its type. The default operation policies assign access based on ownership and type of the object. It does not consider complex relations between objects, users and operations. The specification considers only one to one relationship between users and cryptographic objects.

Managed cryptographic objects in KMIP are dynamic objects which have an associated state. Each cryptographic object is associate to at least one subject who is the owner of the object. The owner represents an entity who created the object and has the special privilege to perform all the key management operations on it. This entity can represent an enterprise, a user of an enterprise, the role or an endpoint who has been delegated authority for creating the object and managing it.

The objects are related to the other objects through cryptographic relations which include a public private key pair, a certificate for a public key, wrapping where one key is used to wrap the another key and derivation where an existing symmetric key is used to derive other keys. These cryptographic relations give rise to the problem of assigning authorization to a user for related keys which also raises the question of whether to assign authorization based on key values or principals associated to the key. For example, a user who is the owner of a base key which is used to derive other keys can implicitly obtain the cryptographic material for the derived keys by reading the base key. Any operation which can change the state or modify the attributes of an object can affect the validity of the authorizations for the current and related objects. The other issues which an authorization model for KMIP should take into consideration are joint ownership which may arise in the case of split keys, delegation of authority to an administrator who is allowed to perform certain privileged operation like revoke, archive, recover, locate on cryptographic objects and defining access based on roles.

2.2 Authorization Specification Language

A system is composed of different resources. A user interacts with the system to perform different actions on system resources. For example in a file system, files and directories are resources and a user can perform read, write and modify operation on these resources. An authorization can be defined as a permission given either to an individual user, group of users or the role for accessing a resource or a type of resource inside a system. The resources inside the system are protected by specifying the set of authorizations inside a policy which is used to evaluate access decision for different access requests. An authorization language provides a mechanism for specifying the policies and verifying whether they satisfy the various constraint of a system. An access control policy can be implemented either as a closed policy, an open policy or a hybrid policy. In a closed policy, all the permitted authorizations for the entire system are fully specified. The default access decision is deny in case there are no matching authorizations for an access request. In an open policy, all denied authorizations are fully specified and in case of no match not applicable is returned.

The Authorization Specification Language [10] provides a logical framework to specify multiple policies for different objects inside a system. The rationale behind using a logic-based language are as below:

1. The semantics of logic-based language are unambiguous, simple and suitable for formal policy verification and validation.

2. The declarative nature of a logic-based language provides equal expressiveness and flexibility, which are compromised in other languages.
3. A logic-based language can capture the complex relations such as implied relations by modeling separate policies for entities involved in the relation. These relations are difficult to specify in a single policy for the entire system.

A policy in ASL is composed of rules which enforce authorizations, derivation of implicit authorizations from those explicitly specified; determining the overriding policy to be applied in case of conflicts; integrity constraints and access decision. The advantage of using ASL is that multiple policies can be specified containing its rules. These policies can be modeled as open, closed and hybrid of both. They can be combined together using rules which implement conflict resolution. The final combined policy representing the access control policy of entire system can be proved to be complete and consistent.

ASL syntax consist of following three types of symbols.

**Constant Symbols** consist of every member of $O \cup T \cup U \cup R \cup I \cup N$. $O$ represents the set of objects, $T$ the set of object types, $U$ the set of subjects, $G$ the set of users, $R$ the set of roles, $+$ and $-$ set of signed actions representing permit and deny authorizations respectively. $N$ represents the set of natural numbers. ± represents either a positive authorization or a negative authorization.

**Variable Symbols** consist of $o, t, s, g, r, V_R, +action, -action$ representing the subset of each of the above sets. $V_R$ represents a subset of the set containing various combination of roles.

**Predicate Symbols** represent various access control relations such as authorizations, derived authorizations, integrity constraints, access decision as discussed below.

- **done(o, s, R, action, t)**: A 5-term predicate symbol consisting of object $o$, user $s$, roles $R$, action and a natural number representing time. A done rule represents the fact that an action has been performed by a subject on an object before time $t$. For example, a doctor can access the medical records of his patient twice a day. To evaluate the decision access policy needs to check whether the requesting doctor has accessed the records before the current request. A done rule can be specified as below

  $$\text{done}(o, s, R, \text{action}, t) \leftarrow .$$

  where $o, s$ and $\text{action}$ are elements of $O, S$, and $A$ respectively.

- **typeof(o', o)**: A binary predicate that captures the grouping relationship between objects. For example, the requested object should be of type secret object. A typeof rule is specified as below

  $$\text{typeof}(o', o) \leftarrow .$$

  where $o'$ and $o$ are elements of $O$.

- **cando(o, s, ±action)**: A ternary predicate symbol that represents explicit authorization policies stated by the system administrator. The arguments consist of an object $o$, subject $s$ and the corresponding action. A cando rule is specified as below

  $$\text{cando}(o, s, \pm \text{action}) \leftarrow L_1 \& \cdots \& L_n.$$ 

  where $o, s$ and $\text{action}$ are elements of $O, S$ and $A$ respectively. $L_i$ can be either typeof.
• **dercando(o, s, \(\pm\)action)**: A ternary predicate symbol that represents the implied authorizations derived using the logical inference rules. The arguments consist of an object \(o\), subject \(s\) and the corresponding action. It is used for expressing propagation of authorizations among relations such as dependency, hierarchy between subjects, objects or actions. For example, if a user has access to read a file related to medical records of a patient, he should not have access to the file containing patient personal information. A *dercando* rule is specified as below

\[
dercando(o, s, \pm \text{action}) \leftarrow L_1 \& \cdots \& L_n.
\]

where \(o\), \(s\) and \(\text{action}\) are elements of \(O\), \(S\) and \(A\) respectively. \(L_i\) can be either *cando*, *dercando*, *done*, or *typeof* literal.

• **do(o, s, \(\pm\)action)**: A ternary predicate symbol that represents the authorizations that hold for each subject on each object enforcing conflict resolution. A conflict occurs if either system can derive multiple decisions or no decision for a single access request. It is used to force a decision in case of conflicts by defining a default conflict resolution policy. The default conflict resolution policy consists of *Derived Override*, *Permit Override* rule. The arguments consist of an object \(o\), subject \(s\) and the corresponding action. A *do* rule is specified as below

\[
do(o, s, \pm \text{action}) \leftarrow L_1 \& \cdots \& L_n.
\]

where \(o\), \(s\) and \(\text{action}\) are elements of \(O\), \(S\) and \(A\) respectively. \(L_i\) can be either *cando*, *dercando*, *done*, or *typeof* literal.

• **grant(o, s, R, \(\pm\)action)**: A 4-term predicate symbol that represents the access decision to allow or deny access to an object \(o\). The arguments consist of an object \(o\), a subject \(s\), a role-set \(R\) containing roles in which subject is currently active and the corresponding action. A grant with negative action means subject \(s\) with active roles \(R\) will not be allowed to perform the action on object \(o\). A *grant* rule is specified as below

\[
\text{grant}(o, s, R, \pm \text{action}) \leftarrow L_1 \& \cdots \& L_n.
\]

where \(o\), \(s\) and \(\text{action}\) are elements of \(O\), \(S\) and \(A\) respectively. \(L_i\) can be either *cando*, *dercando*, *do*, *done*, *typeof* literal. A grant rule contains a collection of rules. Each rule specifies the authorizations on subjects and objects required for performing the requested action.

### 2.3 eXtensible Access Control Markup Language

XACML Version 2.0 [13], or eXtensible Access Control Markup Language, is a declarative, XML-based, standard and generic access control specification language that describes:

- an access control policy language to specify authorizations.
- an access request/response language to create access decision requests and receive response.
- a processing framework specifying data flow entities required to make an access decision.
XACML defines a set of language primitives for describing rules, policies, function, data types etc. Using these primitives, XACML can support a wide variety of access control policies such as Access Control List (ACL) in operating system, JAVA security policies and fine grained policies for distributed hierarchical environments. XACML provides a common policy format, which can be used to share or exchange policies across multiple application domains.

XACML implements context based security where rules and functions used to express a policy are specified using attributes. Attributes are named values representing the identification information for the subjects, resources, actions and environment of a system. Attributes are associated with a data type from the set of data types defined in XACML standard. An access request in XACML consists of number of these attributes that makes up the request context. The attributes in a request are matched with those specified in the rules contained inside a policy to take the access decision. The matching is performed using the match functions defined in XACML as mentioned below:

\[
\text{MatchFcn} ::= \text{type-equal} | \text{type-greater-than} |
\text{type-greater-than-or-equal} | \text{type-less-than} |
\text{type-less-than-or-equal} | \text{type-regexp-match} \\
\text{type-any-of-any} | \text{type-all-of-any} | \text{type-all-of-all} \\
\text{type-one-and-only} | \text{type-bag-size} | \text{type-is-in}
\]

The type in the match functions represents one of the below data types defined in XACML.

\[
\text{Type} ::= \text{string} | \text{boolean} | \text{integer} | \text{double} | \text{datetime} | \text{anyuri}
\]

XACML is an extensible language allowing users to define their own custom syntax and semantics for functions and data types.

### 2.3.1 XACML Policy Model

An XACML policy consists of a single policy or a set of policies contained inside a PolicySet as shown in Figure 2.2. A PolicySet can contain multiple policies or PolicySets either using a reference or a single complete policy. A policy is expressed using a set of rules, each of which evaluates to different access control decision.

This section describes the syntax and semantics for the various elements of XACML policy model. The elements are represented using a scheme-like syntax instead of XML based syntax for readability.

- **Subject**

  In XACML a subject SUB is defined as a conjunctive set of subject attribute values AV, attribute designator AD or attribute selector AS pairs that matches user identification attributes or role assigned to him. Subject attribute designator or selector can be defined as a function which retrieves a Bag of matching subject attribute values related to subject id, name, role or a specific attribute not defined in XACML attribute identifiers using an XPath query from the request context. The subject attribute id attr-ID in Attribute Designator can be subject id, subject role, subject name defined in XACML. In XACML we can represent a set of subjects who may have a joint authority using the conjunction property of subject match.
2.3. EXTENSIBLE ACCESS CONTROL MARKUP LANGUAGE

**Policy/Rule Combining Algorithm**

deny-overrides
permit-overrides
first-applicable
only-one-applicable

![Figure 2.2: XACML Policy Model](image)

**SUB** ::= (Conjunctive SubjectMatch+)

SubjectMatch ::= (AV AD MatchFcn)
∥ (AV AS MatchFcn)

AV ::= (AttributeValue value Type)

AD ::= (SubjectAttributeDesignator attr-ID Type Issuer mustBePresent³)

AS ::= (SubjectAttributeSelector contextPath Type mustBePresent³)

• **Resource**

In XACML, a resource RES is defined as a conjunctive set of attribute value, attribute
designator or attribute selector pairs that matches a resource representing an object in the
system.

RES ::= (Conjunctive ResourceMatch+)

ResourceMatch ::= (AV AD MatchFcn)
∥ (AV AS MatchFcn)

AV ::= (AttributeValue value Type)

AD ::= (ResourceAttributeDesignator attr-ID Type Issuer mustBePresent³)

AS ::= (ResourceAttributeSelector contextPath Type mustBePresent³)

• **Action**

³Indeterminate result will be returned if mustBePresent attribute is true and there is no attribute in the request that matches it.

Divay Bansal  

ETH Zurich, IBM Research
In XACML, an action $ACT$ is defined as a conjunctive set of attributes value, attribute
designator or attribute selector pair that matches an action requested by a subject to be
performed on a resource.

$$ACT ::= (\text{Conjunctive ActionMatch}^+)$$

$$ActionMatch ::= (AV AD MatchFcn)$$

$$|| (AV AS MatchFcn)$$

$$AV ::= (\text{AttributeValue value Type})$$

$$AD ::= (\text{ActionAttributeDesignator attr-ID Type Issuer mustBePresent}^3)$$

$$AS ::= (\text{ActionAttributeSelector contextPath Type mustBePresent}^3)$$

- **Target**

A Target $T$ is a set of subjects $SUB$, resources $RES$ and actions $ACT$ to match the subject $s$, resource $r$ and action $a$ in the request context.

$$T ::= ((\text{Disjunctive SUB}^*), (\text{Disjunctive RES}^*), (\text{Disjunctive ACT}^*))$$

The set of subjects, resources and actions are disjunctive and used to define the various rules in a policy. A target $T$ is also used as an index to identify a policy or a PolicySet.

- **Rule**

A rule is an elementary unit of a policy, evaluated based on its content [13]. A rule consists of a target $T$, describing the attributes that must be present in the context in order for it to be applicable, a $Condition$ specifying constraints that must be fulfilled and an $Effect$ which is either $Permit$ or $Deny$. The effect indicates the consequence of a "true" evaluation of the rule, and both the Target and Condition must be true for the Rule to be true.

$$Rule ::= (T \text{ Condition Effect})$$

$$Condition ::= (\text{Conjunctive Expression}^*)$$

$$Expression ::= \text{Apply} | \text{AS} | \text{AV} | \text{AD}$$

$$| \text{Function} | \text{VariableReference}$$

- **Policy**

A policy is a set of rules, a rule combination algorithm $C_A$ and a single target $T$ on which access control policy applies. The target also acts as an index for the policy.

$$Policy ::= (T \text{ Rule}^* \text{ } C_A)$$

- **PolicySet**

A PolicySet is defined as a set of policies, PolicySets, a target $T$ and a policy combining
algorithm $P_A$. The target also acts as an index for the PolicySet.

$$PolicySet ::= (Policy^* \text{ PolicySet}^* \text{ } P_A \text{ } T)$$

A PolicySet represents a combination of policies for various objects based on the generalization hierarchy of the objects. A PolicySet can also reference other policy or PolicySet using their uniform resource identifier specified using $PolicySetReferenceId$ or $PolicyReferenceId$ element.
2.3. EXTENSIBLE ACCESS CONTROL MARKUP LANGUAGE

An access request can match a different rules inside a policy or different policies inside a PolicySet. The result of matching is the effect defined for the corresponding rule. To arrive at an access decision, XACML define results combining algorithms, which are also classified as rule-combining or policy-combining algorithms. A combining algorithm specifies the procedure by which results of evaluation of a set of rules or policies are combined. XACML defines following combining algorithms for rules and policies:

- **Deny-overrides(Ordered and Unordered)**: If a single policy or rule evaluates to Deny, then the combined result is Deny regardless of evaluation result of other rules or policies.

- **Permit-overrides(Ordered and Unordered)**: If a single policy or rule evaluates to Permit, then the combined result is Permit regardless of evaluation result of other rules or policies.

- **First-applicable**: The result is the same as the result of evaluating the first rule, policy or PolicySet in the set of rules.

- **Only-one-applicable**: This algorithm states that only one policy or PolicySet is applicable for an access decision. If more then one policy or PolicySet are applicable then result is Indeterminate.

### 2.3.2 XACML Request/Response Model

- **AccessRequest**

In XACML an access request $Q$ can be defined as a set of attribute values pair (AVP) which identifies a subject $s$ who has requested to perform an action $a$ on resource $r$.

$\{(s)^*,(r)^*,(a)^*\} \in Q$

$s \equiv (AVP)^*$

$r \equiv (AVP)^*$

$a \equiv (AVP)^*$

XACML request can have an unbounded number of subjects, an unbounded number of resources and only one action. XACML specifies a multiple resource profile [3], as shown in Figure 2.3, for modeling request containing more than one resource in a single request. Each resource can be considered either as an individual resource, children or descendant by adding a `scope` attribute with in the resource. In case of individual resources, each resource is evaluated as separate access request for the specified action. The final access decision for every resource is combined by the context handler.

- **AccessDecision**

The access decision for an access request in XACML can be expressed as:

**Definition 2.3.1.** If an access request $Q$ matches a target $T$ inside a PolicySet\policy and the policy\rule combining algorithm of the PolicySet\policy, combining the effects of various matched rules defined inside the matched policies\policy implies permit, then the subject will be allowed to perform the requested action on the resource. In case it implies deny, then the subject is not allowed to perform the request action. If an error occurs during matching process such as incorrect data type then the PolicySet\policy returns indeterminate. In case of no matching rule the PolicySet\policy returns not applicable.
2.3.3 XACML Data Flow Model

XACML provides a mechanism of evaluating a policy based on the values of the attributes associated with the request. The process of evaluating and enforcing XACML policies includes following entities:

- **Policy Administration Point**: It is used to store, manage and author access control policies in XACML. It acts as policy store from where policies can be loaded or referenced by an authorization mechanism.

- **Policy Enforcement Point**: PEP enforces access decision inside a system by performing specified obligations required for each access decision. It acts as an entry point for an authorization request from a system. It forwards the request to the context handler and receives the authorization response from the context handler.

- **Context Handler**: It converts the access request received from PEP in the native system format to XACML canonical form and XACML response from PDP back to native form, which is forwarded to PEP. It acts as a gateway managing the communication between various entities.

- **Policy Decision Point**: PDP is the main entity that evaluates an access request with the corresponding applicable policies and decides whether authorization shall be granted or denied.

- **Policy Information Point**: This entity acts as a source of attribute values. It has privileges to access various resource repositories of a system. It provides all the information that might be required by the PDP to make the access decision. Figure 2.4 represents the various entities of XACML and data flow between these entities.

In XACML, we can define authorization based on user, roles or group membership instead of resource types. XACML is a powerful authorization language providing flexibility and expressibility to model complex hierarchical relations. It provides a clean way of separating the
security code from the implementation of various actions as shown in Figure 2.4. This way, XACML helps to reduce redundancy as policies can be defined at a very abstract level, provides a simple attribute based mechanism for matching the context required for different actions and a simple mechanism for combining different policies into PolicySets. It also provides a clean separation between access control policies and authorization mechanism. The other advantage of this approach is managing change in the policy, which requires no change in authorization mechanism and server implementation code.
Chapter 3

KMIP Access Control Model

3.1 KMIP Access Control Data Model

This section describes the various elements of KMIP which are used to formalize the authorization specification for access control.

3.1.1 Subject

In KMIP [8], a subject $s$ corresponds to an entity that can send a request to KMIP server. This entity is as an authenticated KMIP user whose credentials can be verified by the KMIP server, which is a necessary condition for performing authorization. Each subject is associated with credentials, which are used to identify him from others. A credential can be a role, a group or a user identifier. Currently, we don’t model roles, which are dependent on an enterprise setting and outside the scope of KMIP system. We model joint authority over a resource using conjuncted set of subjects. We don’t model group of users since users inside a group operates individually. We consider a subject operates individually and is associated to a single key through his credentials. A subject can have multiple keys using different credentials. For example, a subject representing user bob can have multiple keys using different credentials like [bob_home], [bob_office]. In later section, we will discuss ownership, authority relations between subjects and the KMIP managed objects and develop predicates to formalize them.

Definition 3.1.1. A subject $s$ represents a single authenticated user belonging to the set of all authenticated users $S$ of the KMIP system, having credentials that are used to uniquely identify him in the KMIP system.

3.1.2 Resource

A resource represents a managed object or the attributes of a managed object in KMIP. In KMIP, we consider each resource to be associated to at least one subject. One of the associated subjects representing a single user is the owner who created the resource. In some cases, owner is the only subject associated to the resource. An owner is allowed to perform all the KMIP operation on the resource. We don’t consider joint ownership of a resource because a key can only have one associated owner. Since, owner can perform all the operations on a key, it is not feasible to model multiple owner for a single resource. Multiple ownership would mean different values of a key for different user, which need to be synchronized for each user when performing an operation. However, we assume joint authority over a resource, which could be a case for resources of type split keys.
KMIP defines the concept of an administrator for the resources who has the authority to perform operations like revoke, recover and destroy on them. We assume authority and ownership can not be shared for resources, which means a single subject can not be both an owner and an administrator. This segregation is necessary in an enterprise context since enterprise users can create keys and become their owner but the enterprise should have the authority over the created keys. In addition to relation between resources and subjects, resources are also dependent on other resources represented by the link attribute of a resource.

**Definition 3.1.2.** A resource $o$ represents either a single object of type managed object or a subset of attributes, which are part of set of all the managed objects and attributes $O$ in KMIP. A resource is associated with a subset of subjects $s$ from the set of subjects $S$, having at least one subject who is the owner of the resource.

We can categorize KMIP objects in to a hierarchy as shown in Figure 3.1. All the KMIP objects are inherited from the Object class. For completeness we have shown the base objects as children of Object class. Base objects are used within the message of the KMIP protocol and are not managed by the key management system. We consider access control only for managed objects as they are the subjects of key management operations.

![Figure 3.1: KMIP Resource Generalization Hierarchy](image)

Table 3.1 lists the attributes and the various managed objects to which an attribute belongs marked by $\exists$. The table also specifies attributes that can be modified and deleted after an object is created inside the KMIP system. An attribute can be read by any one having the required authorization to perform the read attribute operation. The attribute can be modified or deleted by any KMIP operation only if the corresponding modify and delete columns are marked with $\exists$ in the table. The rest of the attributes are read only which are initialized when the object is created and can not be modified or deleted after that.

$^4$For definition of Secret object, Public object, Template object refer to Sections 3.13.2.1-3.13.2.3 of KMIP Spec, Page 40. [8]

---

Divay Bansal

ETH Zurich, IBM Research
3.1.3 Action

**Definition 3.1.3.** An action \( a \) represents a key management operation and is part of the set of all key management \( A \) operation defined in KMIP.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Certificate</th>
<th>Symmetric Key</th>
<th>Public Key</th>
<th>Private Key</th>
<th>Split Key</th>
<th>Template</th>
<th>Secret Data</th>
<th>Opaque Object</th>
<th>Modify</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique Identifier</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Name</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Object Type</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Cryptographic Algorithm</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Cryptographic Domain Parameters</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Cryptographic Length</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Cryptographic Parameter</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Certificate Type</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Certificate Identifier</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Certificate Issuer</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Certificate Subject</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Digest</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Cryptographic Usage Mask</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Lease Time</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Usage Limits</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>State</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Initial Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Activation Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Process Start Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Protect Stop Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Deactivation Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Destroy Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Compromise Occurrence Data</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Compromise Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Revocation Reason</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Archive Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Object Group</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Link</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Last Change Date</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Contact Information</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Custom Attribute</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
</tbody>
</table>

Table 3.1: Managed Objects and Attributes Mapping
The operations defined above are the only permitted operations allowed by KMIP if a subject is authorized to perform them. KMIP special operations Validate, Query, Cancel, Poll will not be considered for access control policies as they deal with network communication between client and server. The operations Recover, Locate can also be performed on revoked or archived managed objects. ReadAttribute is not a part of KMIP specification. It is used to formalize read authorization on the set of KMIP attributes specified in Table 3.1. Table 3.2 lists operations that can be performed on various managed objects and their attributes in KMIP.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Managed Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Certificate</td>
</tr>
<tr>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>CreateKeyPair</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>X</td>
</tr>
<tr>
<td>Re-Key</td>
<td>X</td>
</tr>
<tr>
<td>Derive Key</td>
<td>X</td>
</tr>
<tr>
<td>Certify</td>
<td>X</td>
</tr>
<tr>
<td>Re-Certify</td>
<td></td>
</tr>
<tr>
<td>Locate</td>
<td>X</td>
</tr>
<tr>
<td>Check</td>
<td></td>
</tr>
<tr>
<td>Get</td>
<td>X</td>
</tr>
<tr>
<td>ObtainLease</td>
<td>X</td>
</tr>
<tr>
<td>GetUsageAllocation</td>
<td></td>
</tr>
<tr>
<td>Activate</td>
<td>X</td>
</tr>
<tr>
<td>Revoke</td>
<td></td>
</tr>
<tr>
<td>Destroy</td>
<td>X</td>
</tr>
<tr>
<td>Archive</td>
<td>X</td>
</tr>
<tr>
<td>Recover</td>
<td>X</td>
</tr>
<tr>
<td>GetAttributes</td>
<td>X</td>
</tr>
<tr>
<td>GetAttributeList</td>
<td>X</td>
</tr>
<tr>
<td>AddAttribute</td>
<td>X</td>
</tr>
<tr>
<td>ModifyAttribute</td>
<td>X</td>
</tr>
<tr>
<td>DeleteAttribute</td>
<td>X</td>
</tr>
<tr>
<td>ReadAttribute</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.2: Operation on Managed Objects
3.1.4 Access Request

In KMIP, access request can be interpreted as a triple of the form (resource, subject, action) stating a subject $s$ as defined in Definition 3.1.1 has requested to perform an action $a$ as defined in Definition 3.1.3 on a KMIP managed resource $o$ as defined in Definition 3.1.2.

3.1.5 Access Control Policy

In KMIP an access control policy can be stated as follows:

**Definition 3.1.4.** A subject $s$ is allowed/denied to perform an action $a$ on a managed object $o$ using the following policy:

- A subject $s$ is permitted to perform the requested action $a$ if it matches one of the subjects associated with the managed object $o$ and satisfy the conditions related to various relation such as ownership, joint authority, and dependency for managed object. The access is denied in all other scenarios.

The above definition represents closed policy for our access control model. All requests that do not satisfy the above condition will be denied.

3.2 KMIP Access Control Policy Design

3.2.1 Authorization Specification Language extension for KMIP

To express access control policies for KMIP access model we will be using logical language for authorization specifications defined in Section 2.2 which defines the following predicates $cando$, $dercando$, $do$, $done$, $typeof$ and $grant$. To express KMIP specific relations between subjects and resources we specify new predicate symbols discussed below in addition to ASL symbols.

- **owner($o, s$)**: A binary predicate that captures ownership of an object $o$ by a user $s$. A user who creates a managed object becomes its owner having privilege of performing all the actions listed in Table 3.2 for the created object type. An owner also has the privilege of granting authorization to other users for performing a subset of action on his managed object. It represents the fact that the requesting user is the owner of the requested object. A **owner** rule is specified as below

$$\text{owner}(o, s) \leftarrow .$$

where $o$ and $s$ are elements of $O$ and $S$ respectively.

- **publicKeyOf($o', o$)**: A binary predicates that captures the relation between a public key object $o'$ and a certificate $o$ or a private key $o$. It represents the fact that requested certificate is attached to a public key or the requested public key is associated to a private key object.

**Notations used in formalizing policies**

- $o$.Attribute represents the KMIP attribute associate with the requested object. The attribute is the part of the access request.

- $o.[A_1, A_2, ..., A_n]$ represents the list of KMIP attributes associated with the requested object. The attributes are also part of the access request.
• \{E_i\} represents a set or a subset of a set of elements representing different values of a KMIP attribute.

• \(\not\subset\) denotes that there is not a subset relation between two sets.

• \(=, \neq\) represents the equality and inequality relation between two sets. Two sets are said to be equal if they have same elements and their intersection returns the same sets.

• \& represents the conjunctive relation.

### 3.2.2 KMIP Server Functionality and Authorization Policies

In order to design the authorization policies we have to define the segregation between server implementation and authorization functionality since key management operations and access control decision are correlated with the environment. There are certain special operations that need to be completely implemented in the server code. Below are examples of the statements which represent these issues:

- **Create**: Any user \(s\) shall be allowed to create a symmetric key object \(o\) at any time. The request should contain the object type and template attribute. The user who creates the object becomes the owner of the object.

  **Input**: Object Type, Template-Attribute
  **Output**: Object Type, Unique Identifier of \(o\)
  **Sideeffect**: owner\((o, s)\)

  **Policy Statement**

  \(\text{cando}(o, s, +\text{Create}).\)

- **Create Key Pair**: Any user \(s\) shall be allowed to create a key pair \(o\) which consist of a private key object \(o'\) and a public key object \(o''\) at any time using create key pair action.

  **Input**: Private Key Template-Attribute, Public Key Template-Attribute
  **Output**: Private Key Identifier, Public Key Identifier
  **Sideeffect**: owner\((o', s), owner(o'', s)\)

  **Policy Statement**

  \(\text{cando}(o, s, +\text{CreateKeyPair}).\)

- **Register**: Any user shall be allowed to register a managed object \(o\) created by the client or obtained by the client through some other means. The request should contain object type and template attribute.

  **Input**: Object Type, Template-Attribute, Object
  **Output**: Unique Identifier of \(o\)
  **Sideeffect**: owner\((o, s)\)

  **Policy Statement**

  \(\text{cando}(o, s, +\text{Register}).\)
These are special operations which should be implemented by the KMIP server code, since the object does not exist when authorization is requested. For create key pair operation KMIP does not specify a single key pair object but a set of two objects which are of type public key and private key, both of which do not exist during authorization check.

In the two operation statements below, the server code shall perform the check regarding the environment like get usage allocation has not been performed or whether the object is in pre-active state. The authorization policy should check whether a subject has the required permissions for the requested managed object and other resources which are involved in performing the requested operation.

- **Modify UsageLimit**: A user $s$ who is the owner of the managed object $o$ shall be allowed to modify the usage limit of the object, if the get usage allocation action has not been performed on the object before modification action is requested.

  **Input** Unique Identifier, Usage Limit Attribute
  **Output** Unique Identifier, Modified Usage Limit Attribute
  **Sideeffect** Not Applicable

  Policy Statement

  $$cando(o, \text{UsageLimit}, \ s, \ +\text{ModifyAttribute}) \leftarrow \text{owner}(o, \ s) \ & \neg \text{typeof}(o, \ \text{SecretData}) \ & \neg \text{typeof}(o, \ \text{OpaqueObject}) \ & \neg \text{typeof}(o, \ \text{Certificate}) \ & \neg \text{done}(o, \ s, \ R, \ \text{GetUsageAllocation}, \ t).$$

- **Modify ProcessStartDate**: A user $s$ who is the owner of the managed object $o$ shall be allowed to modify the ProcessStartDate, if the object is in Pre-Active state.

  **Input** Unique Identifier, Process Start Date Attribute
  **Output** Unique Identifier, Modified Process Start Date Attribute
  **Sideeffect** Not Applicable

  Policy Statement

  $$cando(o, \text{ProcessStartDate}, \ s, \ +\text{ModifyAttribute}) \leftarrow \text{owner}(o, \ s) \ & \neg \text{typeof}(o, \ \text{PrivateKey}) \ & \neg \text{typeof}(o, \ \text{SecretData}) \ & \neg \text{typeof}(o, \ \text{OpaqueObject}) \ & \neg \text{typeof}(o, \ \text{Certificate}) \ & \neg \text{typeof}(o, \ \text{PublicObject}) \ & \neg \text{typeof}(o, \ \text{State}, \ \text{Pre-Active}).$$

### 3.2.3 KMIP Service Semantics and Access Control Policies

KMIP services refer to the various operations specified in Table 3.2 that can be performed by a user on the KMIP managed objects. We need to define protection requirement for these services and formalize them in to authorization policies. In this section we look in to various operations, specify their inputs, outputs and sideeffects, which can help us in formalizing the protection requirement for each operation.
• **Read Attribute:** At the implementation level, most of the operations involve reading the properties of a KMIP object. The read attribute operation is encapsulated from the KMIP services. We specify the protection requirements for a read attribute operation, which can be used to derive the protection requirements for various other operations in KMIP. The read attribute operation can only be applied on attributes which have *read* permission specified in Table 3.1.

  - **Secret Object:** Owner of the managed secret object \( o \) shall be allowed to read its attributes.

    
    \[
    \text{cando}(o.\text{Attribute}, s, \text{+ReadAttribute}) \leftarrow \text{typeof}(o, \text{SecretObject}) \& \text{owner}(o, s).
    \]

  - **Public Object:** Any subject \( s \) can read the attributes of any public object \( o \).

    
    \[
    \text{cando}(o.\text{Attribute}, s, \text{+ReadAttribute}) \leftarrow \text{typeof}(o, \text{PublicObject}).
    \]

  - **Template Object:** Any subject \( s \) can read the attribute of a public template object and only an owner can read the attributes of a private template object.

    
    \[
    \begin{align*}
    \text{cando}(o.\text{Attribute}, s, \text{+ReadAttribute}) & \leftarrow \\
    & \text{typeof}(o, \text{PublicTemplate}). \\
    \text{cando}(o.\text{Attribute}, s, \text{+ReadAttribute}) & \leftarrow \\
    & \text{typeof}(o, \text{PrivateTemplate}) \& \\
    & \text{owner}(o, s).
    \end{align*}
    \]

• **Modify Attribute:** An owner of the requested managed object \( o \) shall be allowed to modify its requested attribute at any time. The modify attribute operation is allowed on attributes which are marked to be modifiable in Table 3.1.

  - **Input** Unique Identifier, Attribute
  - **Output** Unique Identifier, Modified Attribute
  - **Sideeffect** State of object may change

    
    \[
    \text{cando}(o.\text{Attribute}, s, \text{+ModifyAttribute}) \leftarrow \text{owner}(o, s) \& \\
    \neg \text{typeof}(o, \text{PublicTemplate}).
    \]

• **Get:** A subject \( s \) can request the KMIP server to return the managed object \( o \) specified by its unique identifier.

  - **Input** Unique Identifier of \( o \)
  - **Output** ObjectType, Unique Identifier, Managed Object
  - **Sideeffect** Not Applicable

    
    - **Secret Object:** An owner of a requested managed secret object \( o \) shall be allowed to get it at any time.

      
      \[
      \text{cando}(o, s, \text{+Get}) \leftarrow \text{typeof}(o, \text{SecretObject}) \& \text{owner}(o, s).
      \]

    
    - **Public Object:** Any user \( s \) can request to get a managed public object \( o \) at any time.

      
      \[
      \text{cando}(o, s, \text{+Get}) \leftarrow \text{typeof}(o, \text{PublicObject}).
      \]
– **Template Object**: Any user $s$ can request to get a public template object and only an owner can request private managed template object created by him at any time.

\[
cando(o, s, +Get) \leftrightarrow \text{type} o(f, \text{PublicTemplate}).
\]

\[
cando(o, s, +Get) \leftrightarrow \text{type} o(f, \text{PrivateTemplate}) & \text{owner}(o, s).
\]

- **Destroy**: An owner of the requested managed object $o$ can request the KMIP server to destroy its key material. Cryptographic objects may only be destroyed if they are in either Pre-Active or Deactivated state.

**Input** Unique Identifier

**Output** Unique Identifier of the destroyed object

**Sideeffect** Expired or Revoked related objects can’t be recovered.

\[
cando(o, s, +Destroy) \leftrightarrow \text{owner}(o, s) & \neg \text{type} o(f, \text{PublicTemplate}).
\]

- **Activate**: An owner of the requested managed object $o$ which is in pre-active state shall be allowed to bring it in active state using activate operation.

**Input** Unique Identifier

**Output** Unique Identifier of the activated object

**Sideeffect** Modify Object State and Activation Date

\[
cando(o, s, +activate) \leftrightarrow \text{owner}(o, s) & \neg \text{type} o(f, \text{ManagedCryptographicObject}).
\]

- **Revoke**: An owner of the requested Managed Object $o$ shall be allowed to revoke it. If compromise date is specified in the request, state of the object is changed to Compromised, else it is changed to Deactivated with deactivation date set to current date time.

**Input** Unique Identifier

**Output** Unique Identifier of the revoked object

**Sideeffect** Modify Object State, Compromise Date and Deactivation Date

\[
cando(o, s, +revoke) \leftrightarrow \text{owner}(o, s) & \neg \text{type} o(f, \text{TemplateObject}).
\]

- **Archive**: An owner of the requested Managed Object $o$ shall be allowed to archive it using archive operation.

**Input** Unique Identifier

**Output** Unique Identifier of the archived object

**Sideeffect** Not Applicable

\[
cando(o, s, +Archive) \leftrightarrow \text{owner}(o, s).
\]
3.2. KMIP ACCESS CONTROL POLICY DESIGN

- **Recover**: An owner of the requested Managed Object \( o \) which is archived shall be allowed to recover it.

  \( \text{Input} \) \quad \text{Unique Identifier} \\
  \( \text{Output} \) \quad \text{Unique Identifier of the archived object} \\
  \( \text{Side effect} \) \quad \text{Modify Object State} \\

\[
cando(o, s, +recover) \leftarrow \text{owner}(o, s).
\]

- **Re-Key**: An owner of the requested managed symmetric key object \( o \) shall be allowed to create its replacement key at any time. Re-Key should only be performed once on a given key. The link attribute of the existing key is set to point to the replacement key. The various date attributes are copied from the existing key to the replacement key adding the offset if specified in the request.

  \( \text{Input} \) \quad \text{Unique Identifier of existing object, Offset} \\
  \( \text{Output} \) \quad \text{Unique Identifier of new replacement object} \\
  \( \text{Side effect} \) \quad \text{Modify Link attribute of existing symmetric key object} \\

\[
dercando(o, s, +\text{Re-Key}) \leftarrow \text{typeof}(o, \text{SymmetricKey}) \&
dercando(o, \text{Link}, s, +\text{ModifyAttribute}) \&
dercando(o, \text{InitialDate}, s, +\text{ReadAttribute}) \&
dercando(o, \text{ActivationDate}, s, +\text{ReadAttribute}) \&
dercando(o, \text{ProcessStartDate}, s, +\text{ReadAttribute}) \&
dercando(o, \text{DeactivationDate}, s, +\text{ReadAttribute}).
\]

- **Certify**: A subject \( s \) can request to generate a Certificate object for a specified public key object \( o \) using certify operation. The request should contain certificate request type and certificate request.

  \( \text{Input} \) \quad \text{Unique Identifier of } o, \text{ Certificate Request Type} \\
  \text{Certificate Request, Template-Attribute} \\
  \( \text{Output} \) \quad \text{Unique Identifier of generated Certificate object} \\
  \( \text{Side effect} \) \quad \text{Modify Link attribute of public key object} \\

\[
dercando(o, s, +\text{Certify}) \leftarrow \text{typeof}(o, \text{PublicKey}) \&
dercando(o, \text{Link}, s, +\text{ModifyAttribute}).
\]

- **Re-Certify**: A subject \( s \) can request to renew an existing certificate object \( o \) for the same key pair at any time. Only a single certificate shall be renewed at a time and Re-Certify should only be performed once on a given certificate. The link attribute of the existing certificate is modified to point to the new certificate. The link attribute of the Public Object \( o' \) associated with existing certificate is also modified to point to the new certificate.

  \( \text{Input} \) \quad \text{Unique Identifier of } o, \text{ Certificate Request Type} \\
  \text{Certificate Request, Offset, Template-Attribute} \\
  \( \text{Output} \) \quad \text{Unique Identifier of new Certificate object} \\
  \( \text{Side effect} \) \quad \text{Modify Link attribute of existing certificate and associated public key object} \\

\[
dercando(o, s, +\text{Re-Certify}) \leftarrow \text{typeof}(o, \text{PublicKey}) \&
dercando(o, \text{Link}, s, +\text{ModifyAttribute}).
\]
dercando(o, s, +Re-certify) ← typeof(o, Certificate) &
~done(o, s, +Re-certify, t) &
dercando(o.Link, s, +ModifyAttribute) &
publicKeyOf(o', o) &
dercando(o'.Link, s, +ModifyAttribute).

- **Locate**: A subject s can search for one or more managed objects depending on the attributes mentioned in the request. User may also specify maximum number of matched items to return using Maximum Items arguments. Using Storage Status Mask argument user may specify whether to include only on-line objects, only archived objects or both during search. Authorization for locate can be applied during or after the operation has been performed, applying policy on each object o that is returned by locate operation.

  Input Maximum Items, Storage Status Mask, Attributes
  Output Unique Identifiers of matched objects
  Sideffect Not Applicable

dercando(o, s, +Locate) ← dercando(o, s, +Get).

- **Check**: A subject s can use this operation to request the server to check for the use of a Managed Object o using server policy according to the values of the attributes specified in the request. The request should be part of a batched set placed after Locate, Create, Create Pair, Derive Key, Certify, Re-Certify or Re-Key followed by Get. User may specify usage limit count in the request to check the number of usage limits units to be protected against server policy. User may define the cryptographic usage mask in the request to specify the cryptographic usage of the object. User may also specify the lease time to ask the server to validate it against server policy.

  Input Unique Identifier, UsageLimitCount
  CryptographicUsageMask, LeaseTime
  Output Unique Identifier, UsageLimitCount
  CryptographicUsageMask, LeaseTime
  Sideffect Not Applicable

dercando(o, s, +Check) ← dercando(o.UsageLimit, s, +ReadAttribute) &
dercando(o.CryptographicUsageMask, s, +ReadAttribute) &
dercando(o.LeaseTime, s, +ReadAttribute).

- **Get Attributes**: A subject s can request one or more attributes of a Managed Object o using the get attribute operation. The user specifies the unique identifier of the managed object and attributes name in the request.

  Input Unique Identifier, Attributes Name
  Output Unique Identifier, Attributes
  Sideffect Not Applicable
3.2. KMIP ACCESS CONTROL POLICY DESIGN

\[\text{dercando}(o,[A_1], s, +\text{GetAttributes}) \leftarrow\]
\[\text{dercando}(o.A_1, s, +\text{ReadAttribute}).\]

\[\text{dercando}(o,[A_1, A_2, ..., A_n], s, +\text{GetAttributes}) \leftarrow\]
\[\text{dercando}(o.[A_1], s, +\text{GetAttributes}) \&\]
\[\text{dercando}(o.[A_2, ..., A_n], s, +\text{GetAttributes}).\]

- **Get Attribute List**: A subject \(s\) can request a list of the attribute names associated with a Managed Object \(o\) using get attribute list operation.

<table>
<thead>
<tr>
<th>Input</th>
<th>Unique Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Unique Identifier, Attributes Name</td>
</tr>
<tr>
<td>Sideffect</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

\[\text{dercando}(o, s, +\text{GetAttributeList}) \leftarrow \text{dercando}(o, s, +\text{Get}).\]

- **Add Attribute**: An owner of the requested Managed Object \(o\) shall be allowed to add a new attribute instance to it and set its value using add attribute operation. For multi instance attribute, existing attribute shall not be modified. Read only attributes shall not be added using the add attribute operation.

<table>
<thead>
<tr>
<th>Input</th>
<th>Unique Identifier, Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Unique Identifier, Added Attribute</td>
</tr>
<tr>
<td>Sideffect</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

\[\text{dercando}(o.\text{Attribute}, s, +\text{AddAttribute}) \leftarrow\]
\[\text{dercando}(o.\text{Attribute}, s, +\text{ModifyAttribute}).\]

- **Delete Attribute**: An owner of the requested managed Object \(o\) shall be allowed to delete an attribute associated with it using delete attribute operation. Attributes that are always required to have a value shall not be deleted. The request should contain the attribute name. In case of attribute having multiple indexes, attribute index has to be specified in the request.

<table>
<thead>
<tr>
<th>Input</th>
<th>Unique Identifier, Attribute Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Unique Identifier, Deleted Attribute</td>
</tr>
<tr>
<td>Sideffect</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

\[\text{dercando}(o.\text{Attribute}, s, +\text{DeleteAttribute}) \leftarrow\]
\[\text{dercando}(o.\text{Attribute}, s, +\text{ModifyAttribute}).\]

- **Get Usage Allocation**: A subject \(s\) can request the KMIP server to obtain an allocation from the current usage limits value so that he can use the managed cryptographic object \(o\) for applying cryptographic protection. The request should be received after the activation

<table>
<thead>
<tr>
<th>Input</th>
<th>Unique Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Unique Identifier</td>
</tr>
<tr>
<td>Sideffect</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

\[\text{dercando}(o.\text{Usage}, s, +\text{GetUsage}).\]
date and before the Protect stop date. The request specifies the number of units that need to be protected. The allocation amount requested should be less than the usage limit value of the managed object.

**Input** Unique Identifier, Usage Limits Count  
**Output** Unique Identifier  
**Sideeffect** Not Applicable

\[
\text{dercando}(o, s, +\text{GetUsageAllocation}) \leftarrow \neg \text{typeof}(o, \text{OpaqueObject}) \& \\
\neg \text{typeof}(o, \text{SplitKey}) \& \\
\neg \text{typeof}(o, \text{SecretData}) \& \\
\text{dercando}(o.\text{UsageLimit}, s, +\text{ReadAttribute}).
\]

- **Obtain Lease:** A subject \( s \) can request to obtain a new lease time for a specified Managed Object \( o \) using obtain lease operation. The lease time is an interval value that determines when the client’s internal cache of information about the object expires and needs to be renewed. On expiration of lease, user shall not be allowed to use the corresponding secret object until a new lease is obtained.

**Input** Unique Identifier  
**Output** Unique Identifier, Lease Time, Last Change Date  
**Sideeffect** Not Applicable

\[
\text{dercando}(o, s, +\text{ObtainLease}) \leftarrow \text{dercando}(o.\text{LeaseTime}, s, +\text{ReadAttribute}) \& \\
\text{typeof}(o, \text{ManagedCryptographicObject}).
\]

### 3.3 KMIP Strict Access Control Policy Design

Cryptographic operations such as key wrapping, key derivation create dependency relations among keys. The dependency relation here means the bitwise difference between keys can be retrieved using class of cryptographic functions. This relation creates a dependency among the security requirements where access to a key depends upon the security of all keys related to it. For example, using derive operation a hierarchical dependency as shown in Figure 3.2 is created among the keys. A base key is used to derive other keys using key derivation scheme such as one way hash functions or pseudo random functions. A user having access to a base key should have access to all the derived keys else he can exploit the dependency relation between keys using a brute force attacks [5] to retrieve the bitwise difference information.

There are known API attacks [5, 6] such as key conjuring using the Import/Export functionality of KMS system. The key conjuring attack leads to unauthorized generation of keys in the system. These attacks arise due to vulnerabilities in the cryptographic protocols and mechanism used in performing the operations. These vulnerabilities can be handled using an efficient access control mechanism which considers the relations among keys, the history of accesses and key usage type to take access decision. The access control mechanism has to be stateful considering all the access states of the key from creation and the integrity relations with the dependent keys. For example, to read a base key which is used to derive other keys, the access control policy must check the integrity constraint whether the subjects who read this key before can get all the dependent keys of the base key. A policy that implements the integrity constraints to guarantee that a user can not exploit the API to violate the access control policies is termed as strict access control.
3.3. KMIP STRICT ACCESS CONTROL POLICY DESIGN

3.3.1 ASL Primitive Extension for Strict Access Control

- **dependent(o’, o)**: A binary predicate that captures dependency relation between two managed secret objects of the same type. In KMIP, a user can derive symmetric keys from a base symmetric key creating hierarchical dependency as shown in Figure 3.2. The security of a descendant managed secret object is affected by the security policies of all the ancestor managed secret objects. The dependent primitive represents the fact that the managed secret object o’ is cryptographically related with the managed secret object o. A dependent rule is specified as below:

\[
dependent(o', o) \leftarrow .
\]

3.3.2 KMIP Services and Strict Access Control Policies

- **Derive Key**: An owner of the requested managed cryptographic object o of type symmetric key or secret data shall be allowed to perform the derive operation. The managed cryptographic object o must have the derive key bit set in the cryptographic usage mask attribute. The cryptographic usage mask is an attribute associated with a managed object representing the key usage. It is a subset of \{SignKeyBit, VerifyKeyBit, EncryptKeyBit, DecryptKeyBit, DeriveKeyBit, WrapKeyBit, UnwrapKeyBit\} set.

\[
\text{Input} \quad \text{Unique Identifier of } o, \text{ Object Type} \\
\text{Output} \quad \text{Unique Identifier of newly derived object.} \\
\text{Side effect} \quad \text{Modify Link attribute of } o
\]

\[
dercando(o, s, +\text{DeriveKey}) \leftarrow \text{typeof}(o, \text{SymmetricKey}) \& \\
\quad \text{owner}(o, s) \& \\
\quad \text{dercando}(o.\text{Link}, s, +\text{ModifyAttribute}).
\]

\[
\text{error()} \leftarrow \text{dercando}(o, s, +\text{DeriveKey}) \& \\
\quad \{\text{DeriveKeyBit} \notin o.\text{CryptoUsageMask}\}.
\]
derive(o, s, +DeriveKey) ← type(o, SecretData) & owner(o, s,) &
    derive(o, Link, s, +ModifyAttribute).

error() ← derive(o, s, +DeriveKey) &
    {DeriveKeyBit} ≠ o.CryptoUsageMask.

- **Read**: An owner of the requested managed object o can read the object and its attribute. The subject s must also be allowed to read all the dependents of the requested managed object o.

  Input: Unique Identifier of o
  Output: Managed Object
  Side Effect: Not Applicable

  derive(o, s, +Read) ← derive(o, s, +Get).
  error() ← dependent(o', o) &
    do(o, s, +Get) &
    do(o', s, -Read).

- **Unwrap**: An owner of the specified unwrapping key w of type symmetric key shall be allowed to use it to perform unwrap operation. The unwrapping key w must not have sign key bit, verify key bit, encrypt key bit, decrypt key bit, derive key bit set in the cryptographic usage mask attribute of the key. The unwrapping key must not have been read by any subject in the system.

  Input: Unique Identifier of unwrapping key
  Output: Not Applicable
  Side Effect: Not Applicable

  cando(w, s, +Unwrap) ← type(w, SymmetricKey) &
    owner(w, s).
  error() ← derive(w, s, +Unwrap) &
    done(w, s', Ø, +Get, t).
  error() ← derive(w, s, +Unwrap) &
    {UnwrapKeyBit} ≠ w.CryptoUsageMask.

- **Import**: A subject s can import a wrapped object if he is allowed to perform the unwrapping operation.

  derive(o, w, s, +Import) ← derive(w, s, +Unwrap).
3.4. COMPLETENESS AND CONSISTENCY OF KMIP POLICIES

- **Export**: An authorization request must satisfy the following conditions to perform the export operation:

1. subject $s$ must be the owner of the requested managed object $o$.
2. subject must be the owner of the specified wrapping key $w$ of type symmetric key.
3. wrapping key $w$ must not belong to dependent objects of managed object $o$.
4. wrapping key must only be used for performing wrap operation.
5. all the subjects who have read the wrapping key must have read access to all the dependent objects of the requested managed object $o$.

**Input**: Unique Identifier of wrapping key, Unique Identifier of managed object

**Output**: Wrapped managed object

**Side effect**: Not Applicable

\[
cando((o, w), s, +Export) \leftarrow \text{owner}(o, s) \\& \\text{owner}(w, s) \\& \\
\text{type}(w, \text{SymmetricKey}) \\& \text{dependent}(w, o) \\& \\
\{\text{WrapKeyBit}\} = w.\text{CryptoUsageMask}.
\]

\[
error() \leftarrow \text{done}([o, w], s', \emptyset, +Export, t) \\& \\text{dependent}(o', o) \\& \\
\text{do}(o', s', -Get).
\]

### 3.4 Completeness and Consistency of KMIP Policies

An authorization specification \([10]\) in ASL consists of a set of authorization (\textit{cando}), derivation (\textit{dercando}), resolution (\textit{do}), and access decision (\textit{grant}) rules. The access decision for an access request is based on the evaluation of the above rules. The authorization specification of an operation shall be complete and consistent so that the policies are correct and provide consistent evaluation result for different combination of entities in an access request.

**Definition 3.4.1.** An authorization specification is said to be **complete**, if for every access request there exist an access decision.

To make KMIP policy complete, we include the following ASL rules:

\[
grant(o, s, R, \pm a) \leftarrow do(o, s, \pm a) \& R = \emptyset. \tag{3.1}
\]

\[
do(o, s, -a) \leftarrow \neg \text{dercando}(o, s, +a). \tag{3.2}
\]

The first rule specifies an access decision for an operation to be either permit represented by positive or deny represented by negative sign. A positive evaluation result in \textit{do} statement leads to permit access decision in grant statement and a negative result leads to deny access decision.

The second rule specifies that if there does not exist a positive authorization then the policy will return deny decision for an access request.

**Definition 3.4.2.** An authorization specification is said to be **consistent**, if there is no access request for which the access decision returns a permit as well as a deny.
To make the KMIP policy consistent, we specify the following ASL rules for conflict resolution to be included in every authorization specification:

\[
\begin{align*}
do(o, s, +a) &\leftarrow \text{dercando}(o, s, +a) \& \neg \text{dercando}(o, s, -a). & (3.3) \\
do(o, s, -a) &\leftarrow \text{dercando}(o, s, -a). & (3.4)
\end{align*}
\]

The third rule specifies that the conflict resolution mechanism will return a positive authorization if and only if for the access request only a positive authorization can be derived; i.e., the evaluation does not result in a negative authorization.

The fourth rule specifies that the conflict resolution mechanism will return a negative authorization, if for the access request a negative authorization can be derived from the policy.

These four rules are sufficient to make the KMIP access control policy complete and consistent. Below we present a proof for the previous statement.

To prove that the KMIP access control policy will be complete and consistent using the above rules, we consider four cases representing all possible access decision for an operation according to its authorization specification:

1. **No Authorization Specification:** This case represents the situation when the KMIP policy does not contain authorization for an operation requested in the access request. In this case, rule (3.2) will turn the result in to a negative decision.

   \[\vdash \neg \text{dercando}(o, s, -a) \& \neg \text{dercando}(o, s, +a)\]

   Using rule (3.2) above,
   \[\Rightarrow \text{do}(o, s, -a)\]

   Using rule (3.1) above,
   \[\Rightarrow \text{grant}(o, s, -a).\]

   Therefore, the KMIP policy will always return a deny decision when policy does not contain authorization for a requested operation. There will always be an access decision for every request proving the completeness of the KMIP policy.

2. **Positive Authorization Specification:** This case represents the situation when the KMIP policy only contains a positive authorization for a requested operation.

   \[\vdash \text{dercando}(o, s, +a) \& \neg \text{dercando}(o, s, -a)\]

   Using rule (3.3) above,
   \[\Rightarrow \text{do}(o, s, +a)\]

   Using rule (3.1) above,
   \[\Rightarrow \text{grant}(o, s, +a).\]

   Therefore, KMIP policy will always return a permit decision when only a positive authorization exists for the requested operation.

3. **Negative Authorization Specification:** This case represents the situation when the KMIP policy only contains a negative authorization for a requested operation.

   \[\vdash \text{dercando}(o, s, -a) \& \neg \text{dercando}(o, s, +a)\]

   Using rule (3.4) above,
   \[\Rightarrow \text{do}(o, s, -a)\]

   Using rule (3.1) above,
   \[\Rightarrow \text{grant}(o, s, -a).\]
Therefore, KMIP policy will always return a *deny* decision when only a negative authorization exist for the requested operation.

4. **Both Negative and Positive Authorization Specification:** This case represents the situation when the KMIP policy contains both positive and negative authorization for a requested operation.

\[
\begin{align*}
\triangleright & \text{dercando}(o, s, -a) \& \text{dercando}(o, s, +a) \\
& \text{Using rule (3.4) above,} \\
& \Rightarrow \text{do}(o, s, -a) \\
& \text{Using rule (3.1) above,} \\
& \Rightarrow \text{grant}(o, s, -a).
\end{align*}
\]

Therefore, KMIP policy will always return a *deny* decision when both positive and negative authorization exist for the requested operation proving the *consistency* of the KMIP policy.

From the above discussion, the access decision for an access request can be modeled as function calls from resolution to derived rules to facts. For example, the access decision evaluation specification for get attribute list operation for secret objects specified in Section 3.2.3 can be completely represented as:

\[
\begin{align*}
\text{grant}(o, s, R, +\text{getAttributeList}) & \leftarrow \text{do}(o, s, +\text{getAttributeList}) \& R = \emptyset. \\
\text{do}(o, s, +\text{getAttributeList}) & \leftarrow \text{dercando}(o, s, +\text{getAttributeList}). \\
\text{dercando}(o, s, +\text{getAttributeList}) & \leftarrow \text{dercando}(o.\text{Attribute}, s, +\text{get}). \\
\text{dercando}(o.\text{Attribute}, s, +\text{get}) & \leftarrow \text{cando}(o.\text{Attribute}, s, +\text{get}). \\
\text{cando}(o.\text{Attribute}, s, +\text{get}) & \leftarrow \text{owner}(o, s).
\end{align*}
\]

1. The *grant* rule specifies that a request to get attribute list of a managed object submitted by a subject having no role will be granted if subject has a positive authorization for the get attribute list operation.

2. The *do* rule states that a subject can exercise get attribute list operation on a managed object if subject has a positive authorization over it. Here we enforce the closed policies on managed objects to satisfy consistency of *do* statement.

3. The *dercando* rule for get attribute list derives a positive authorization for subject *s* to get a attribute list of a managed object *o*, if *s* can derive positive authorization for getting that object represented by *get* operation in Section 3.2.3.

4. The *dercando* rule for getting managed object implies derivation of positive authorization for get operation based on the presence of positive authorization for get operation provided by the system administrator.

5. The *cando* rule represent the fact that a subject *s* has a positive authorization to get the requested managed object *o*, if *s* is the owner of the *o*. 
Chapter 4

KMIP XACML Policy Implementation

4.1 Translating ASL Primitives Into XACML Policy Components

For translation, we need to consider the action hierarchy in addition to resource hierarchy, which arises due to implied relations among various actions and their side effects. Figure 4.1 represents the action hierarchy implied from the various ASL rules. The first column represents the various set of resources, the second column contains the action that can be directly applied on these resources and the third column contains the action that are derived from direct actions.

The various primitives defined in ASL above can be translated in to XACML components as below:

4.1.1 ASL Constant Symbols

The ASL constant symbols represent the subset of the set of objects o, subjects s and actions action of the KMIP system. In XACML, they are translated as attribute value pair (AVP) consisting of an attribute identifier and an attribute value representing a resource, a subject and an action inside an access request.

4.1.2 ASL Predicate Symbols

typedef(o', o) :

The typedef primitive represents the grouping of objects. It is translated in to a policy or a PolicySet containing authorization for the similar type of elements. The policies are indexed by the object type represented by typedef in their target. Depending on the position of the object type in the grouping hierarchy, a policy can be combined with other policies in an ancestor PolicySet. The ancestor PolicySet shall contain a disjunctive set of resources representing all its descendants in its target. In KMIP resources are grouped in to a hierarchy represented by typedef in the ASL policies.

typedef(o', o) ::= PolicySet( Target( Resources ) )
| Policy( Target( Resources ) )
Resources = (Disjunctive (R1, R2, ⋅⋅⋅ , Rn))
Ri = (ResourceMatch string-equal(AV, AD))
AV = (AttributeValue 'objectType' string)
AD = (ResourceAttributeDesignator resource-type string)
In XACML, \textit{owner} is translated as a condition inside a rule. The condition checks whether the value of owner attribute associated with resource is equal to the subject identifier in the
request. $o$ is translated as resource-owner attribute and $s$ is translated as subject-id attribute.

$$\text{owner}(o, s) ::= \text{Condition}(\text{Expression})$$

$$\text{Expression} = (\text{Apply string-equal}(\text{Expression}1, \text{Expression}2))$$

$$\text{Expression}1 = (\text{Apply string-one-and-only}(\text{AD}))$$

$$\text{AD} = (\text{ResourceAttributeDesignator resource-owner string})$$

$$\text{Expression}2 = (\text{Apply string-one-and-only}(\text{AD}))$$

$$\text{AD} = (\text{SubjectAttributeDesignator subject-id string})$$

$$\text{dependent}(o', o)$$:

The dependent primitive is translated as a part of the XACML data flow entities. PIP is used to retrieve the dependent resources of the requested resource. The access request contains the requested resource $o$ identifier. Using this identifier we retrieve the link attribute of the requested resource using PIP. The link attribute represents the directly dependent resources. PIP recursively retrieve the link attribute of already extracted dependent resources, creating a tree like structure with resource contained in initial access request as the root. This way we can retrieve all the dependent resources of the requested object. This set of dependent resources is used by the PDP in evaluating the access decision using set based match function in a condition inside the rule.

$$\text{done}(o, s, R, \text{action}, t)$$:

In KMIP, the primitive represents the history of accesses for the requested resource. It represents a subject $s$ having role $R$ who has performed the $\text{action}$ on an object $o$ at a time before the current access request. In XACML, $\text{done}$ is translated as set of subjects who have accessed the resource before as a part of the current access request. The set of subjects representing access history is retrieved using the Policy Information Point. PIP sends a request to KMIP server to retrieve the access history of the resource sent in access request. The retrieved set of subjects is used by PDP to evaluate the access decision using a set based match function in a condition inside the rule.

4.1.3 ASL Rules

$$\text{cando}(o, s, \pm\text{action})$$:

In XACML a $\text{cando}$ statement is translated as a rule inside a policy. The tuple $(o, s, \text{action})$ translates in to elements of target inside the rule. In KMIP target, $o$ represents resource-id, $s$ represents subject-id and action represents action-id attributes of an access request. A positive action represents permit effect and a negative action represents deny effect in a rule. The disjunctive set of literals on the right hand side of $\text{cando}$ rules are translated as a disjunctive set of
4.1. TRANSLATING ASL PRIMITIVES INTO XACML POLICY COMPONENTS

Apply functions inside the condition of a rule.

cando(o, s, ±action) ::= Rule(Effect Target Condition)

Effect = if ±action then Deny else Permit
Target = (Subject Resource Action)
Subject = (anySubject)
Action = (ActionMatch string-equal(AV, AD))
   AV = (AttributeValue GET string)
   AD = (ActionAttributeDesignator action-id string)
Resource = (ResourceMatch string-equal(AV, AD))
   AV = (AttributeValue SecretObject string)
   AD = (ResourceAttributeDesignator resource-id string)

dercando(o, s, ±action)

To translate dercando rule in XACML we have to design the solution for following problems:

1. An access request in XACML version 2.0 can contain only one action.

   In KMIP, actions can be represented in a hierarchy as shown in Figure 4.1 where a child action can only be performed if the subject is allowed to perform all the ancestor action. For example re-key operation involve calls to following operation during its execution:

   • modify attribute operation to modify the link attribute.
   • read attribute operation to read the initial date, activation date, process start date.

   This condition can be easily modeled in ASL using dercando rules. A dercando rule is similar to a set of function calls as shown in ASL policy statements for KMIP in Section 3.2.2. The dercando rule in ASL involves evaluating a set of conjunctive literals and combining their results to decide whether the rule implies positive or negative authorization.

   Implementing the action hierarchy in XACML is not possible [11, Section 4.2] because of its limitation in expressiveness. We looked in to some of the solutions, each having limitations either in terms of implementation or involve increasing the coupling between polices and authorization mechanisms.

   Our solution focuses on using the current XACML policy model, without involving writing additional authorization code. The solution involves writing secondary policies for the direct actions in the action hierarchy, which are indexed by their policy identifier instead of target. The secondary policies do not contain the action represented by the policies in their target. For example, secondary policies for read attribute action do not contain this action in the authorization rules but only the resources and corresponding conditions. The action is specified as the policy identifier, which is used to reference the policy in the policy set of the derived action. This assumption is necessary since an XACML access request can only contain one action, which will be the derived action in case of an access request for performing the derived operation.

   Also, as shown in above re-key example, multiple resources are involved in performing the operation. The policy evaluation performed by XACML policy decision point is defined in terms of a single requested resource, with access decision contained in single
Result element of the response context. To include multiple resources in a single access request we have to use the XACML multiple resource hierarchy profile [3]. The policy decision engine evaluates the access decision for each requested resource in the access request. The final access decision is the combination of all the access decisions for each individual resource in the request. For example, the access request for re−key will be a tuple of the form [Joe, {Link, InitialDate, ActivationDate, ProcessStartDate}, SymmetricKey, Joe, re−key] representing [subject, [resources], resource−type, resource−owner, action]. Using multiple resource profile PDP will transform the request as shown below

\[
\text{[Joe, \{Link, SymmetricKey, Joe\}, re−key]} \\
& \text{[Joe, \{InitialDate, SymmetricKey, Joe\}, re−key]} \\
& \text{[Joe, \{ActivationDate, SymmetricKey, Joe\}, re−key]} \\
& \text{[Joe, \{ProcessStartDate, SymmetricKey, Joe\}, re−key]}
\]

Each access request above will be evaluated separately by the PDP and the final result for re−key action will be the conjunctive combination of the individual results.

Figure 4.2: KMIP XACML Policy Implementation

2. Combining secondary policies in to a PolicySet.

The policies or PolicySets can be combined inside a single PolicySet by referencing the policies or PolicySet using their policy identifier value. The combination process follows the policy combining algorithm specified in the root PolicySet. XACML standard specifies four policy combining algorithms: first-applicable, only-one applicable, permit-overrides and deny-overrides. The combination process must follow the completeness and consistency rules discussed in Section 3.4. The combination process must take the following issues into consideration

- Referenced policies are not disjoint having rules containing similar target. Hence, a request can match multiple policies.
• Each algorithm is sequential, processing policies or PolicySets in the order they are referenced. The policies are evaluated till the point where the condition of the algorithm is not satisfied. For example deny-override algorithm will process the referenced policies until it finds a policy which returns a deny. No parameter can be passed to the algorithms specifying a different policy order or a range of policies or PolicySets that need to be evaluated for a particular request.

• Semantics of combination algorithm. For example, deny-override algorithm returns permit if all the referenced policies return permit; deny if any of the referenced policies return deny and indeterminate in all other cases. Therefore, all the referenced policies must return a decision so that the root PolicySet is complete and consistent.

Let us consider the example of an access request for re−key actions discussed above, which requires to check whether the subject is allowed to perform modify attribute and read attribute actions on the subset of attributes of the requested resource.

The root policy in KMIP is the Managed Object PolicySet as shown in Figure 4.4. The PolicySet references the Secret Object, Public Object, Template Object policy sets and combine them using the first-applicable policy combination algorithm. To make the Managed Object policy complete and consistent we add a dummy policy containing a deny rule, which is applicable to every request, as the last policy. The dummy policy behaves like a catch in terms of programming semantics.

The Secret Object, Public Object and Template Object PolicySets are indexed by the resource type attribute value. For example, Secret Object PolicySet target contains Symmetric Key, Private Key, Split Key, Secret Data and Opaque Object. These PolicySets references the PolicySets for direct and derived actions, which are indexed by their action attribute value. The referenced direct and derived action PolicySets are combined using first-applicable rule.

The re−key request will first match the Secret Object PolicySet using the resource type attribute. Since, re−key represents a derived action it will match the derived action policy referenced inside the Secret Object PolicySet.

The PolicySet of re−key derived action references secondary policies of read attribute and modify attribute operation as discussed above. Both the policies contains a rule having subset of KMIP attribute name as resources in its target and a condition required to perform this action.

If the read attribute policy is referenced first then the combining algorithm will return the result based on this policy if it matches the condition. There is no way to tell the combining algorithm which referenced policy shall be used to evaluate a request.

To solve the above problems using XACML semantics, we model the secondary policies as closed policies. The policy contains permit rules containing authorization condition required to be satisfied for performing the operation represented by the policy. A wild card deny rule is added to the policy as the last rule. The policy specifies first-applicable as the rule combining algorithm so that if no permit rule matches, the policy returns a deny using the last wild card rule. It is similar to a function which returns a boolean. The function returns true if the function parameters satisfy the conditions specified inside the function. If no condition is satisfied, the function by default returns false. In the context of combined policies, this means policy will return deny for a request which does not belong to it.
One issue we need to look into is handling overlapping policies. For example, read attribute and modify attribute secondary policy can have similar set of matching attribute names in its rule target. The similarity of two policies will depend on the authorization condition specified in the rule. If the condition is different then a request will be always evaluated by the correct policy, but in case of similar condition the request will be evaluated by the policy referenced first. To solve this problem, re-key PolicySet contains two PolicySets. Each PolicySet contains the reference to the secondary policy. The PolicySet containing modify attribute policy will contain Link attribute as its target resource. Therefore, a request for Link attribute in re-key matches the correct secondary policy.

**Discussion** :

In this paragraph, we discuss two alternative strategies for implementing dercando rules with XACML policy model.

1. Customizing XACML syntax and extending XACML evaluation engine. In this approach, we can develop the syntax of a match function for XACML, which can reference a rule and return the result of evaluation of the referenced rule. For multiple actions we can reference multiple rules and combine their evaluation results through a logical function specified in XACML functions. Also, with each rule we need to specify a way for passing the request parameters as attributes which can be used by that rule. The advantage of this approach is we do not have to specify a lot of redundant policy or PolicySet. The disadvantage is that extending the framework is difficult and it will add more complexity to the system.

2. Extending the Java Authentication and Authorization Service (JAAS) model for XACML implementation [9]. Using JAAS we can implement the implied relations inside the Java code. The disadvantage of this approach is dependency of authorization mechanism and the authorization policies. It will not provide clear segregation between policies and mechanism. Hence, handling change in a policy will be difficult and the policies can not be distributed or shared across various mechanism.

**error()** :

An integrity constraint might involve multiple subjects, multiple resources and multiple actions. For example, integrity constraint for export operation involves verifying whether the set of subjects who have read the wrapped key before can read the set of all the dependent resources of the requested resource.

To model multiple actions in XACML we use the above approach of defining secondary policies for each action literal. To model integrity constraints involving set operations between set of subjects $S$ and resources $R$. Each resource in the set contains set of owners which is matched with set of subjects. We use the below algorithm to model the XACML access request using multiple resource profile:
For each resource $r$ in $R$
\[ \gamma = \text{get } r \text{ properties} \] (4.1)
\[ \varphi = \text{get } r \text{ owners} \] (4.2)
\[ \sigma = \text{create request involving } S, \{ \gamma, \varphi \} \text{ and action} \] (4.3)
\[ \Psi = \text{evaluate authorization decision for the } \sigma \] (4.4)
if $\Psi$ is deny return deny (4.5)
endFor (4.6)
return permit (4.7)

Another hypothetical approach for implementing error would be integrating it within the KMIP server. PIP queries the KMIP server to get the value of the integrity constraint for a particular request. If the KMIP server returns permit then the authorization request is evaluated further else the PDP returns deny.

do(o, s, ±action) :
In XACML do can be translated as the rule-combining or policy combining algorithm, which is used to combine the evaluation results of various rules or policies. For, KMIP we use derive OVERRIDE combining algorithm for combining results of policies in a PolicySet and first-applicable as rule combination algorithm in a policy.

grant(o, s, R, ±action) :
In XACML grant can be translated as the access decision returned by the Policy Decision Point after successfully evaluating the access request.

The final translation of ASL primitives to XACML components is shown in the Figure 4.3
4.2 KMIP XACML Policy Hierarchy

The XACML access control policies for KMIP is implemented by translating the ASL KMIP policies discussed above. A bottom up approach is used starting from implementing the secondary policies, combining them in to derived action Policysets. The derived action Policysets are grouped together with the direct action policies based on the `typeof` literal representing the resource type in their ASL policy. All the action policies having same resource type are combined into single Policyset. These individual Policysets are combined along the resource hierarchy, discussed in Section 3.1.2.

As shown in Figure 4.4, managed object in the resource hierarchy represents the root PolicySet for the system. The root PolicySet references the PolicySets for secret object, public object...
and template object, combining them using \textit{first-applicable} algorithm semantic. To make the root policy complete so that there is always a decision for every request, a wild card policy is added in the end. The wild card policy can match any request and returns a deny authorization. Therefore, for any access request if there does not exist a match in all the referenced PolicySets in the root policy, then the root PolicySet will return deny.

The referenced PolicySets \textit{secret object}, \textit{public object} and \textit{template object} are indexed by their child resource type as shown in Figure 3.1. For example, target of \textit{secret object} PolicySet contains disjunctive sets of \{Symmetric Key, Private Key, Split Key, Secret Object, Opaque Data\} as resources. An access request containing any one of these resources as a resource attribute will match the secret object PolicySet and use it to evaluate the access decision. The referenced PolicySets are further split in to direct action and derived action PolicySets to incorporate the action hierarchy.

The direct action policy is indexed by the actions which can be performed directly and contains the corresponding authorization rules for those action. A derived action PolicySet is indexed by the action which involves call to direct actions. The derived action PolicySet references the secondary policies to evaluate the access decision as discussed above. The secondary policies are modeled as open policies containing a permit rule specifying authorization context for performing the operation represented by the policy and a deny rule for all other requests that do not match the policy context.

The evaluation of an access request flows from the root policy to the referenced PolicySets, then to direct or derived action PolicySets and finally to secondary policies if the request contains a derived action. The access decision follows the path back from the leaf policy to the root PolicySet evaluated and combined using the policy or rule combination algorithm semantics of each policy in the path.

\section{4.3 KMIP XACML Data Flow Model}

The following steps illustrates how an authorization request can be handled by various XACML entities, separating server functionality, authorization polices and authorization mechanism to evaluate the decision using the polices.

1. XACML policies, PolicySets are loaded by the Policy Decision Point (PDP). The policies are provided by a Policy Administration Point(PAP) which stores and manages the polices. These policies, PolicySets have an associated life cycle in PDP. In the IBM Argus implementation of XACML evaluation engine when a request context matches a policy context that policy is brought in the inService state, else it is kept in an open state. The PDP matches the policy attribute with the request context to decide whether authorization should be granted.

2. KMIP sends an authorization request to the Policy Enforcement Point (PEP), which contains the subject, resource and action attributes. The subject attributes contains user credentials like subject id, resource attributes containing resource-id, resource type, resource owner and action attributes contain action id.

3. PEP before forwarding the request to the Context Handler (CH), can optionally check for the requested resource in the KMIP resources for its existence. This can be useful for actions like \textit{locate}, \textit{create}, \textit{register} where authorization can only be checked during the operation or after the completion of the operation.
4. PEP forwards the request to the CH, which transforms the request into an XACML request context.

5. The XACML request context is sent to PDP, which loads the corresponding policy to evaluate the access control decision.

6. PDP can also request CH to provide additional attributes related to resource, subject and environment which have to be matched with those in request context.

7. CH forwards the request to the Policy Information Point (PIP), which has required privilege to access KMIP resources.

8. PIP queries the KMIP Resources to get the desired attributes as requested by the CH.

9. PIP after collecting the requested attributes, sends them back to the CH.

10. CH after collecting all the attributed values requested by the PDP send them back as part of the response to the PDP.

11. PDP uses all the attributes available and apply match and combination algorithms to decide whether request should be permitted or denied. PDP creates a XACML response context from the evaluation result which is sent back to the CH.

12. CH transforms the XACML response context back to a response format which can be understood by the PEP.
Chapter 5

System Design and Implementation

5.1 System Architecture

Figure 5.1: KMIP XACML Data Flow Architecture

Figure 5.1 represents the architecture of the authorization mechanism that is integrated with the KMIP server whose architecture is discussed in Section 2.1. It represents the middleware infrastructure that is used for implementing the communication between various data flow entities. Below we discuss in detail the design choices for each component and their implementations.

Access Control Integration We consider two design strategies for integrating the access control mechanism with KMIP each having their advantages and disadvantages as discussed below.

1. Proxy Design It is based on the principal of reference monitor. The access control component is implemented as a proxy before KMIP server. A KMIP client request will be first processed by the access control proxy. The proxy will have to parse the request, authenticate the client by communicating with the KMIP server and retrieve attributes required for authorization.
The request if authorized will be forwarded to KMIP server, which again needs to authen-
ticate the client and perform the requested operation.

2. **Integrated Design** : It involves adding an additional layer for access control inside the
KMIP API stack. The layer is implemented as a plugin providing an interface that can be
easily integrated with KMIP API. In this design, KMIP server will authenticate the client
and will call the access control service for deciding whether the user has authorization to
perform the requested operation before dispatching the request.

<table>
<thead>
<tr>
<th>KMIP Parser</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>No end to end security. Re-</td>
<td>Single SSH connection be-</td>
</tr>
<tr>
<td>End to End Security</td>
<td>quire two SSH connections.</td>
<td>tween client and KMIP. Pro-</td>
</tr>
<tr>
<td>Client to Proxy, Proxy to Server</td>
<td></td>
<td>vides end to end security</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Complex</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Need to communicate with server to authenticate client, retrieve attributes</td>
<td>Need to communicate with server to retrieve attributes.</td>
</tr>
</tbody>
</table>

Table 5.1: Pros/Cons of Access Control Integration Designs

**Discussion**  Based on the above pros/cons we selected the integrated design for implementing
the authorization mechanism. The overhead in implementing integrated design is less compared
to proxy design. Because of its simple design it can be easily plugged inside the KMIP API
without requiring modification to the current KMIP implementation.

**Policy Decision Point (PDP)**  *IBM Argus Engine* is used for performing the evaluating of an
access decision using the XACML policies. IBM Argus Engine is implemented as a JAVA based
XACML evaluation plugin, which runs inside an *OSGi [2] * container. The *OSGi* framework
provides a dynamic development model where applications are composed of different reusable
components. These components communicate with each other through service objects provided
by the *OSGi* framework. The framework provides a dynamic environment where an application
can be build by combining different reusable components that had no a priori knowledge of each
other.

Policy Decision Point is implemented as a bundle that runs inside an *OSGi* framework. It
provides context handler service which act as interface between *IBM Argus Engine* and other
data flow entities. PDP wraps the resource, request, subject and action objects provided by the
engine by defining adapter classes for them. These adapter classes convert the KMIP access
request in to objects that are compatible with the engine implementation.

The context handler also provides an interface to communicate with *PIP* to retrieve the
attributes required to evaluate the access decision by the XACML policies.

**Policy Enforcement Point (PEP)**  Policy Enforcement Point act as an enforcer who needs to
communicate with the KMIP server and PDP to evaluate and enforce the access decision. It act
as a bridge between KMIP native system and PDP. KMIP server is a *J2EE* web application server
running inside a *Tomcat* container. PDP is a plugin bundle running inside an *OSGi* container.
PEP should act as a bridge between these two containers providing a way of communication.
Also, PEP should be able to interact with any obligation services like logging or sending a notification to administrator for every deny decision.

To incorporate the above design considerations, PEP is implemented as a web service. It defines an access request XML schema as shown in Figure 5.2, which can be used by any client to communicate with the PEP over HTTP protocol. We model PEP as a web bundle inside OSGi. For this purpose we embed Jetty web server inside the OSGi framework. The PEP is deployed inside the Jetty web server. This way PEP is able to access any component installed inside the OSGi framework and any application can access PEP from outside the OSGi framework over HTTP protocol.

![Figure 5.2: PEP XML Access Request Schema](image)

**Policy Information Point (PIP)** Policy Information Point retrieves the attributes of the KMIP managed objects needed by PDP to evaluate the access decision. It needs to interact with the key store to access the attributes of the managed objects. Since key store can only be accessed by a key management server, therefore, PIP should be able to query key management server with special privileges to retrieve the desired attributes.

This is incorporated by implementing PIP as a KMIP client having special privileges to perform only get attribute operation to retrieve the attributes required by PDP. The KMIP client proxy is ported to OSGi framework as PIP component. Special KMIP authentication credentials are associated with PIP. PIP can securely query the KMIP server using SSL or HTTPS transport as shown in Figure 5.1.

### 5.2 Sequence Diagram

Figure 5.3 represents the sequence diagram representing KMIP access control data flow entities, their activation and lifetime during a single access request and their interaction steps for evaluating the access decision with time.

Below we discuss the sequence diagram entities and their functionality represented in the diagram.

1. **AuthorizationHttpClient**: This entity represents the interface used by the KMIP server to create an authorization request and send it to PEP. It encapsulates the implementation details of building an XML based access request from KMIP objects compatible with PEP and the mechanism required for communicating with PEP. It provides following functionality createResource, addResourceAttributes, createSubject, createAction, createRequest, hasPermission.
Figure 5.3: Sequence Diagram

2. **PEPServer**: This entity receives the access requests from the KMIP server. The entity first validates the request content with its XML request schema. It then deserializes the various components in the access request into corresponding objects and forwards the request object to ContextHandlerService to get the access decision.

3. **ContextHandlerService**: This entity acts as a bridge between other access mechanism entities. It initializes the evaluation engine and loads the policies from the policy store.
It parses the PEP request object into corresponding XACML request objects. It provides functionality for communicating with the PIPService to retrieve the attributes.

4. PIPService: This entity provides the functionality for communicating with the KMIP Service to retrieve the resource attributes required for taking the access decision.

5.3 Performance Evaluation

In this section, we perform experiments to provide an insight into the performance of our system which can be used as a benchmark for comparing and analyzing it with different solutions. The data can be used in analyzing impact of integration of our solution on KMIP performance. The performance evaluation criteria of our KMIP access control system is based on the following factors:

1. **Load**: Load on the server is measured in terms of resource consumption in processing a request. Load is represented by the number of concurrent request being processed by the server. Each concurrent request is allocated a separate thread to which various server resource are allocated. In case of shared resources thread have to wait in a queue for using those resources. As, the number of concurrent request increases the load on server increases.

2. **Throughput**: Throughput measures the number of requests processed in a second by the server. It gives an idea regarding the processing speed of the server under the varying load.

3. **Latency**: Latency represents the time taken in between receiving the response for two consecutive requests. In case of concurrent requests, it represents the time taken between two requests in consecutive set of requests. Latency gives us a measure of the delay in processing a request. It involves total time to connect to the server, processing the request by the server and waiting for receiving the response back. By increasing the concurrency the latency increases because server has to process the entire set of concurrent request before processing the next consecutive request.

The performance evaluation results of a web based application is affected by various factors such as network conditions, server performance etc. To compare our solution with another we need to benchmark our solution with the data that can act as a baseline taking into account the above effects. We consider the following components for performing the evaluation:

1. **Web Server**: We measure the values of above performance criteria’s for the Jetty web server used in our implementation. The performance data act as a baseline for comparing the performance of other components. We deployed a simple HTML page on the server and measured the performance values for different criteria’s. This data represents the maximum throughput and minimum latency that can be achieved.

2. **Policy Decision Point (PDP)**: To study the performance of PDP we measure the values of throughput and latency under varying load conditions when only PDP is plugged inside the system. We also measure the response time for different operations involving different number of policies and rules. This data provides benchmark for the performance of the IBM Argus engine, which can be used to compare it with different XACML evaluation engines such as Sun XACML, XACML Light and XACML enterprise [10].
3. **Policy Information Point (PIP):** We measure the impact of PIP round trip time on the overall performance of the entire system. The data is collected after plugging the PIP inside the system with PDP. We compare the PDP+PIP data with the data collected only for PDP to evaluate the performance impact.

5.3.1 **Test Bed**

The test bed involves following two machines with described configurations:

- A virtual machine running Red Hat Enterprise Linux workstation with 64 bit Kernel version 2.6. The virtual machine runs on Intel(R) Xenon(TM) 1 CPU 3.00GHz processor with 2 GiB of RAM and 18 GiB of hard disk. It use an IBM JVM for running JAVA code. We deploy our access control system on the virtual machine, running inside OSGi container. The Jetty [7] web server is embedded inside the OSGi container to handle the request over HTTP. We created a JUNIT plugin test module containing AuthorizationHTTPClient to dispatch request to the PEP server over HTTP channel. The module contains test suite having test case for each KMIP operation and is used for verifying the policies in different scenarios for each operation.

- A local machine running Windows XP with 32 bit Kernel. The local machine runs on Intel(R) Core(TM) 2 CPU 2.33GHz processor with 3GiB of RAM and 100GiB of hard drive. It use an IBM JVM for running JAVA code. We deploy the KMIP prototype built at IBM Research on this machine. The KMIP server implementation is a J2EE based application. The KMIP application runs inside a Tomcat web container. We also run our performance benchmarking framework Apache Benchmark (AB) on this machine. AB simulates sending request to the access control system from the KMIP server and receiving the response back. AB provides a good measure of the performance of the various components for a single type of request. In a real world environment users send different kind of requests and receive different response, which is the limitation of AB.

5.3.2 **Experimental Results**

We use the following apache benchmark query to perform the experiments

```
ab -n 20000 -c 100 -p Request.xml -T text/xml http://PEPService/PEPServer
```

The \(n\) switch in the query represents the total number of requests to be forwarded to server, \(c\) represents the concurrency level, \(p\) represents the data to be posted with the request and \(T\) represents the content type of data sent with the request.

The above query tells apache benchmark (ab) to request the PEPService 20000 times and to make these requests 100 at a time. For performing the experiments we considered two set of concurrent requests. The first set involves variation of concurrency by a factor of 10 between 0 and 100. This set gives information regarding variation of latency and throughput with a small change in load. The second set involves variation of concurrency by a factor of 100 from 0 till 1000. To minimize the effect of factors affecting the performance results we take an average over 4 readings for each entry in the two sets.

1. **Latency**

Figure 5.4 represents the latency for the components discussed above under varying load conditions. The request involves access request for performing a direct operation involving a single resource. The latency increases with increase in concurrency for each component because server has to process more requests before moving to the next consecutive
request from a user. The latency is maximum for each set of concurrent requests when the PIP is plugged inside the system with the PDP.

As the concurrency increases, the difference between latency for PDP and PDP+PIP decreases. The reason for this behavior can be explained by the mechanism of handling requests by a web server. A web server is associated with a set of acceptor threads and a thread pool whose maximum size can be configured. The acceptor threads take a request from the incoming request buffer and create a thread for each request and add it to the thread pool for processing. In case of concurrent threads each thread is started simultaneously for processing. As, concurrency increases the time taken by each thread to process the request increases, which is able to compensate for the round trip time taken by PIP to get the attributes.

Therefore, as concurrency increases the impact of PIP to retrieve attributes for an access request involving one single resource decreases.

2. Throughput Figure 5.5 represents the requests per second that are processed by the access control system with different components under varying load conditions. The access request involves performing a direct operation involving a single resource. Throughput increases in the interval between 0 and 100 simultaneous request, then decreases slightly and finally reach a state where increase in concurrency only have a minor impact on the throughput. The reason for this behavior can be explained using the latency graph above. The time to process two consecutive request inside same batch of concurrent requests varies only slightly by a factor of 1-2 millisecond with increase in concurrency. Therefore, increase in concurrency does not have much impact on the throughput.

Another interesting result is the convergence of throughput curves for PDP and PDP+PIP to approximately similar values with increase in concurrency. Here, PIP sends only one request to KMIP for each access request. From latency graph above we can analyze that difference in latency for PDP and PDP+PIP decreases with increase in concurrency. Therefore, server is able to process approximately equal number of requests per second with increasing concurrency and hence similar throughput.

3. Performance Impact of Derived Operation
Figure 5.5: Throughput Vs Load

Figure 5.6 represents the performance result of PDP and PDP+PIP for derived operation ReKey involving five resources within a single access request. Here, PIP send five requests to KMIP for each access request. As can be seen from the latency vs load curve on the left and throughput vs load curve on the right, PIP has a major impact on the performance. Latency almost increase by three times and throughput decreases by almost the same amount.

Figure 5.6: Indirect Operation Performance

4. XACML Engine Performance Evaluation

On comparing the results of latency and throughput in Figures 5.4, 5.5 and 5.6 for PDP only, we see a minor increase in latency and decrease in throughput.

To look in to the reason for the variations in performance result for PDP between direct and indirect operations, we calculated the number of policies, rules, subjects, resources and actions involved in each operation, as shown in Table 5.2. Rekey operation involves maximum number of policies, rules and resources that explains the difference in the PDP evaluation time.
5.3. PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th>Operation</th>
<th>Policy</th>
<th>Rule</th>
<th>Subject</th>
<th>Resource</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetOperation</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ReKey</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>AddAttribute</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Operation Components

We measured the time taken by PDP for evaluating the access decision for a single request for different operations specified in table. Figure 5.7 contains graph measuring the performance of IBM Argus Engine in terms of time taken to evaluate different number of policies and rules.

![Figure 5.7: XACML Engine Performance](image)

5.3.3 Discussion

As discussed above, PIP becomes a performance bottleneck when number of resources inside a single access request increases. One strategy that can be employed to optimize the performance of PIP is to cache the KMIP responses at PIP. With caching PIP do not have to contact the KMIP server for similar resources. The cache will be valid for one single access request involving multiple resources. Caching will decrease the PIP total round trip time for operations involving multiple resources and having attributes as resources such as Rekey.

We also looked into the memory usage pattern of the virtual machine with increase in concurrency to look for possible memory leaks. There was not much variations in the memory usage and JVM was able to garbage collect the objects after finishing the processing for different set of concurrent requests.
5.4 Related Work

In industry, there are products for key management system from various vendor such as IBM, HP, Sun. Since these product are proprietary solution, no information is available regarding the implementation of their access control component. In academia, to our knowledge, little work has been done in the area of designing access control for key management system. Samarati et al. [14] proposed an access control model for a public key management system which discusses the various nuances of key management system related to access control. They propose a powerful but flexible access control model using a policy based on principal, ownership, and authority relationships on keys. The model categorizes the data elements of public key management system in terms of subjects, objects and operations, which are part of policies for the access control model. The set of operations, which are considered in the model include, register, lookup, revoke, escrow, decrypt, sign and recover. The objects in the model consist of public and private keys. The subjects are defined as individual users, users as a member of a group or users having roles. The model provides an expressive logic-based authorization language to express the various conditions on a subject and keys. An owner can specify while delegating authority, whether it applies to a specify key or to the key registered for a principal using the above language. The owner can also specify whether the delegated authority holds upon expiration or revocation of key. The model allows for joint authority, delegation of authority and assignment of specific roles to a delegated authority. It considers public-private key pair as an individual object, which are not related to any other key or key pair in the system. It also considers operations to be independent of each other.

5.4.1 Comparison With Related Work

The related work discussed above focuses on public key management systems involving public-private key pairs providing services including Register, Lookup, Escrow, Encrypt, Revoke, Recover and Sign operation. Our work focuses on key management systems such as KMIP with wider scope involving symmetric keys, public-private key pairs, split keys, opaque object, secret data and certificates. These key management systems are used by big enterprises providing wide range of cryptographic services discussed in Section 3.1.3.

The related work considers each subject in a public key management system to be a singleton pair [user, role] where role can be null. It also specifies that a public key management system do not handle role or group management, which need to be performed by the enterprise only. Every key is associated with a set of subjects and an owner can specify authorizations either based on key identifier or subjects associated with the key. Our approach considers a subject to be a unique authenticated KMIP user, which can contain information regarding its role or group membership as a part of its credentials. Each key is associated with a set of subjects containing at least one subject who is the owner. We do not consider any subject hierarchy to prevent any unauthorized access leakage such as a subject getting an access to a resource because of its group membership or a role.

The related work does not consider any relation between resources since they do not take in to account the operation semantics and API attacks that can be performed. We looked in to the dependency relation that is created among the resources because of cryptographic nature of operations. We described a strict form of access control policies taking in to account dependency relation and integrity constraints to prevent any attacks that can be performed by exploiting this information. We focused our work on understanding the semantics of various key management operation from a cryptography perspective and their side effects. Based on the knowledge, we proposed an action hierarchy among the operations, which should be taken in to account while
evaluating the access decision.

Both our work and related work uses different logic-based languages to design the access control policies. The related work focuses only on relations between subject and keys such as authority, joint authority, delegation and ownership. They provide their own logic-based language to model these relations. Our work takes into account complex relations such as derivation of authorizations along action hierarchy, integrity constraints. We model separate policies for each operation and combine these policies together using our translation mechanism to create the complete policy for the entire system. We use ASL to model the implied relation since it provides expressiveness and flexibility to represent them. We provide the rules for making our key management system policy complete and consistency. We look into different scenarios to verify that our policy will be complete and consistent.

In addition, we provide an implementation of these policies in XACML by providing a translation from ASL primitives to XACML components. Using the translation we implemented policies for different operations, building a policy hierarchy by combining these policies together in XACML. We developed a service oriented authorization mechanism, which is independent from policies and server functionality. We discussed the performance and provide a benchmark that can be used to compare it with other mechanisms.
Chapter 6

Conclusion and Future Work

In this thesis, we presented a novel approach for solving the problem of designing and implementing access control policies for key management systems. In Chapter 1, we looked into the nuances of key management systems from an access control perspective. From the discussion we concluded that key management systems require an efficient access control mechanism not only to prevent unauthorized access but also to verify the integrity constraints of the involved entities. The integrity constraints are required to prevent leakage of information using known API attacks. We discussed the notion of strict access control which takes into account the history of accesses. We considered the related work in this area, their design considerations and assumptions in building access control policies for a key management system. This motivated us into defining our goals and provided us with an insight into design and implementation considerations. After understanding the complexity of XACML, we proposed to design the policies using ASL and translating them into XACML.

In Chapter 2, we introduced the various components involved in the thesis. We discussed in detail various aspects of KMIP and its architecture. We considered a logic-based authorization language ASL, discussed its syntax, semantics and reasons for choosing this language to design our policies. We discussed in detail the three components of XACML and provided a simple representation for its policy elements.

In Chapter 3, we designed our access control model using the KMIP specification. We defined the various elements of our model including Subject, Resource and Action. We discussed the assumptions regarding authority and joint authority in our model. We extended ASL by adding primitive symbols representing relations between elements. We discussed the semantics of various KMIP services including its side effects and designed our policies using ASL for each operation. We provided formulation of the policies taking into account various aspects of access control such as derivation, integrity and access constraints. Since, each operation is represented by a different policy we defined rules for combining these policies together. These rules are proved to make our policies complete and consistent in different scenarios.

In Chapter 4, we provided a translation of ASL primitives into XACML components based on their semantics. While performing the translation we discussed the possible solutions for implementing policies containing multiple actions. We proposed a solution by implementing secondary policies to represent direct actions which can be referenced by derived actions policies that are dependent on direct actions. Using this solution we were able to incorporate all authorizations inside the policies instead of code providing a segregation between access control and server functionality.

In Chapter 5, we discussed the implementation details for building the authorization mechanism involving the access control data flow entities. We looked into design considerations based on entities communication and integration with evaluation engine and the current KMIP imple-
mentation. We used a service oriented architecture to design our system to take the advantage of easy integration with any platform and language. We developed a test bed to test our policies. We did a performance evaluation of our design in terms of latency and throughput for different components. This data provides a benchmark criteria of our implementation, which can be used for comparing it with other solutions. We also discussed the various optimization techniques that can be used to increase the performance of our solution.

Due to implementation restriction of KMIP, we did not implement strict access control policies into XACML policies. We provided a theoretical study of translation algorithms for modeling the request and the corresponding policies for strict access control.

**Future Work**

Delegation of responsibility is an interesting research area in access control. Designing policies based on delegation is a challenging task. In the context of key management systems, delegation means giving right of ownership on a key to another user. Therefore, policies involving delegation should take in to account the relations between different entities and incorporate an auditing mechanism to prevent any fraud using access leakage by exploiting any integrity constraint. Currently, KMIP and XACML version 2.0 do not support delegation of permissions. Therefore, as a future work we propose building access control policies based on delegation and implementing a delegation service inside KMIP.

To abstract the complexity of XACML from policy author, we propose building a translation tool based on the approach we used in this thesis. The tool will involve implementing a translation algorithm with logic-based language authorization rules as input and corresponding XACML policies as output. A policy author can easily design his authorization policies using an abstract highly expressible logic-based language like ASL. The set of symbols of a logic-based language can be easily represented graphically using flow diagrams. The tool shall be able to automatically generate XACML from the logic-based language representation using translation algorithms.
Appendix A

KMIP Default Operation Policy

This appendix contains the default operation policy in a tabular form for each class of managed object as specified in KMIP spec [8, page 40]. For each operation we specify a set of policy statements by which we can decide whether to allow or deny the access to a user.

A.1 Operations Outside of Operation Policy

The set operations given below will not be considered as part of the default operation policy of KMIP. We do not consider Certify and Re-Certify operation since it requires an implementation of certification authority, which is currently out of the scope of KMIP implementation.

- Create
- Create Key Pair
- Register
- Certify
- Re-Certify
- Validate
- Query
- Cancel
- Poll

Default Operation Policy for Secret Objects

The policy statements in Table [A.1] applies to children of Secret Object as shown in Figure [3.1]. These policies also applies to opaque object which are secret objects but not managed cryptographic objects. In the policy statements, a secret object is represented by $o$, user is represented by $s$, $R$ represents role-sets which in our case will be any role. The policy statements below describes how authorization rules for the various operations on secret objects either can be specified as facts or logically derived from the specified facts for other operations.
## Table A.1: Operation on Secret Managed Objects

<table>
<thead>
<tr>
<th>Operation</th>
<th>Secret Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>( \text{cando}(o, s, +\text{get}) \leftarrow \text{owner}(o, s) ).</td>
<td></td>
</tr>
<tr>
<td>Get Attributes</td>
<td>( \text{dercando}(o, [A_1, A_2, \ldots, A_n], s, +\text{getAttributes}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, A_1, s, +\text{readAttribute}) &amp; ) &lt;br&gt;&lt;br&gt;( \ldots \ldots ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, A_n, s, +\text{readAttribute}) ).</td>
<td></td>
</tr>
<tr>
<td>Get Attribute List</td>
<td>( \text{dercando}(o, s, +\text{getAttributeList}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, s, +\text{get}) ).</td>
<td></td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>( \text{cando}(o, \text{Attribute}, s, +\text{modifyAttribute}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{owner}(o, s) ).</td>
<td>State of object may change.</td>
</tr>
<tr>
<td>Add Attribute</td>
<td>( \text{dercando}(o, \text{Attribute}, s, +\text{addAttribute}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{Attribute}, s, +\text{modifyAttribute}) ).</td>
<td></td>
</tr>
<tr>
<td>Delete Attribute</td>
<td>( \text{dercando}(o, \text{Attribute}, s, +\text{deleteAttribute}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{Attribute}, s, +\text{modifyAttribute}) ).</td>
<td></td>
</tr>
<tr>
<td>Re-Key</td>
<td>( \text{dercando}(o, s, +\text{re-Key}) \leftarrow ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{Link}, s, +\text{modifyAttribute}) &amp; ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{InitialDate}, s, +\text{readAttribute}) &amp; ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{ActivationDate}, s, +\text{readAttribute}) &amp; ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{ProcessStartDate}, s, +\text{readAttribute}) &amp; ) &lt;br&gt;&lt;br&gt;( \text{dercando}(o, \text{DeactivationDate}, s, +\text{readAttribute}) ).</td>
<td>Modify Link attribute of existing symmetric key.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Operation</th>
<th>Secret Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
</table>
| Locate                 | `dercando(o, s, +locate) ←
                          dercando(o, s, +get).`                                                                                                                                                                                                                  |                                                                                                                                                                     |
| Check                  | `dercando(o, s, +check) ←
                          dercando(o.UsageLimit, s, +readAttribute) &
                          dercando(o.CryptoUsageMask, s, +readAttribute) &
                          dercando(o.LeaseTime, s, +readAttribute).`                                                                                                                                                                     |
| Obtain Lease           | `dercando(o, s, +obtainLease) ←
                          dercando(o.LeaseTime, s, +readAttribute) &
                          ¬typeof(o, OpaqueObject).`                                                                                                                                                    |                                                                                                                                                                     |
| Get Usage Allocation   | `dercando(o, s, +getUsageAllocation) ←
                          dercando(o.UsageLimit, s, +readAttribute) &
                          ¬typeof(o, SplitKey) &
                          ¬typeof(o, SecretData) &
                          ¬typeof(o, OpaqueObject).`                                                                                                                                                    |                                                                                                                                                                     |
| Activate               | `cando(o, s, +activate) ← owner(o, s) &
                          ¬typeof(o, OpaqueObject).`                                                                                                                                                    | Modify Object State and Activation Date.                                                                                                                             |
| Revoke                 | `cando(o, s, +revoke) ← owner(o, s)`.                                                                                                                                                                                                        | Modify Object State, Compromise Date and Deactivation Date.                                                                                                          |
| Destroy                | `cando(o, s, +destroy) ← owner(o, s)`.                                                                                                                                                                                                       | Expired or Re-voked related objects can’t be recovered.                                                                                                               |
| Archive                | `cando(o, s, +archive) ← owner(o, s)`.                                                                                                                                                                                                       |                                                                                                                                                                     |
A.1. OPERATIONS OUTSIDE OF OPERATION POLICY

Table A.1 – continued from previous page

<table>
<thead>
<tr>
<th>Operation</th>
<th>Secret Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recover</td>
<td>( cando(o, s, +\text{recover}) \leftarrow \text{owner}(o, s) ).</td>
<td>Modify Object State.</td>
</tr>
</tbody>
</table>

**Default Operation Policy for Public Objects**

Table A.2 represents the policy statement for the KMIP Managed Public object which consist of Certificate and Public Key object as shown in Figure 3.1. In the policy statements below, public object is represented by \( o \), an authenticated KMIP user is represented by \( s \) and \( R \) represent role-sets which in our case will be any role.

Table A.2: Operation on Public Managed Objects

<table>
<thead>
<tr>
<th>Operation</th>
<th>Public Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>( cando(o, s, +\text{get}) ).</td>
<td></td>
</tr>
<tr>
<td>Get Attributes</td>
<td>( dercando(o,[A_1,A_2,\ldots,A_n], s, +\text{getAttributeList}) \leftarrow ) ( dercando(o,[A_1,A_2,\ldots,A_n], s, +\text{getAttributes}) \leftarrow ) ( dercando(o, s, +\text{getAttributeList}) \leftarrow ) ( dercando(o, s, +\text{get}) ).</td>
<td></td>
</tr>
<tr>
<td>Get Attribute List</td>
<td>( dercando(o, s, +\text{getAttributeList}) \leftarrow ) ( dercando(o, s, +\text{get}) ).</td>
<td></td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>( cando(o,\text{Attribute}, s, +\text{modifyAttribute}) \leftarrow \text{owner}(o, s). )</td>
<td>State of object may change.</td>
</tr>
<tr>
<td>Add Attribute</td>
<td>( dercando(o,\text{Attribute}, s, +\text{addAttribute}) \leftarrow ) ( dercando(o,\text{Attribute}, s, +\text{modifyAttribute}) ).</td>
<td></td>
</tr>
<tr>
<td>Delete Attribute</td>
<td>( dercando(o,\text{Attribute}, s, +\text{deleteAttribute}) \leftarrow ) ( dercando(o,\text{Attribute}, s, +\text{modifyAttribute}) ).</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
## Table A.2 – continued from previous page

<table>
<thead>
<tr>
<th>Operation</th>
<th>Public Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locate</td>
<td>dercando(o, s, +locate) ← dercando(o, s, +get).</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>dercando(o, s, +check) ← dercando(o.UsageLimit, s, +readAttribute) &amp; dercando(o.CryptoUsageMask, s, +readAttribute) &amp; dercando(o.LeaseTime, s, +readAttribute).</td>
<td></td>
</tr>
<tr>
<td>Obtain Lease</td>
<td>dercando(o, s, R, +obtainLease) ← dercando(o.LeaseTime, s, +readAttribute).</td>
<td></td>
</tr>
<tr>
<td>Activate</td>
<td>cando(o, s, +activate) ← owner(o, s).</td>
<td>Modify Object State and Activation Date.</td>
</tr>
<tr>
<td>Revoke</td>
<td>cando(o, s, +revoke) ← owner(o, s).</td>
<td>Modify Object State, Compromise Date and Deactivation Date.</td>
</tr>
<tr>
<td>Destroy</td>
<td>cando(o, s, +destroy) ← owner(o, s).</td>
<td>Expired or Revoked related objects can’t be recovered.</td>
</tr>
<tr>
<td>Archive</td>
<td>cando(o, s, +archive) ← owner(o, s).</td>
<td></td>
</tr>
<tr>
<td>Recover</td>
<td>cando(o, s, +recover) ← owner(o, s).</td>
<td>Modify Object State.</td>
</tr>
</tbody>
</table>

### Default Operation Policy for Template Object

Table A.3 contains the authorization decision policy statements for Private Template objects. Private template objects are created by client when they register a new template object with the KMIP server. In the policy statement below, $o$ represents the private template object, $s$ represents the authenticated KMIP user, $R$ represents role-sets which in our case is any role.
Table A.3: Default Operation Policy for Private Template Objects

<table>
<thead>
<tr>
<th>Operation</th>
<th>Private Template Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>( \textit{cando}(o, s, +\text{get}) \leftarrow \textit{owner}(o, s). )</td>
<td></td>
</tr>
<tr>
<td>Get Attributes</td>
<td>( \textit{dercando}(o, [A_1, A_2, \ldots, A_n], s, +\text{getAttribute}) \leftarrow ) ( \textit{dercando}(o, [A_1, A_2, \ldots, A_n], s, +\text{readAttribute}). )</td>
<td></td>
</tr>
<tr>
<td>Get Attribute List</td>
<td>( \textit{dercando}(o, s, +\text{getAttributeList}) \leftarrow ) ( \textit{dercando}(o, s, +\text{get}). )</td>
<td></td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>( \textit{cando}(o, \text{Attribute}, s, +\text{modifyAttribute}) \leftarrow ) ( \textit{owner}(o, s). )</td>
<td></td>
</tr>
<tr>
<td>Add Attribute</td>
<td>( \textit{dercando}(o, \text{Attribute}, s, +\text{addAttribute}) \leftarrow ) ( \textit{dercando}(o, \text{Attribute}, s, +\text{modifyAttribute}). )</td>
<td></td>
</tr>
<tr>
<td>Delete Attribute</td>
<td>( \textit{dercando}(o, \text{Attribute}, s, +\text{deleteAttribute}) \leftarrow ) ( \textit{dercando}(o, \text{Attribute}, s, +\text{modifyAttribute}). )</td>
<td></td>
</tr>
<tr>
<td>Locate</td>
<td>( \textit{dercando}(o, s, +\text{locate}) \leftarrow ) ( \textit{dercando}(o, s, +\text{get}). )</td>
<td></td>
</tr>
<tr>
<td>Destroy</td>
<td>( \textit{cando}(o, s, +\text{destroy}) \leftarrow \textit{owner}(o, s). )</td>
<td></td>
</tr>
</tbody>
</table>

Table A.4 contains the authorization decision policy statements for the Public Template objects. Public Template objects are default template objects which are provided by the KMIP server to various users. In the policy statements below, \( o \) represents a public template object, \( s \) represents an authenticated KMIP user and \( R \) represents role-sets which in our case can be any role.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Public Template Object Authorization Policy Statement</th>
<th>Sideeffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>( cando(o, s, +\text{get}). )</td>
<td></td>
</tr>
<tr>
<td>Get Attributes</td>
<td>( \text{dercando}(o, [A_1, A_2, \ldots, A_n], s, +\text{getAttributes}) \leftarrow ) ( \text{dercando}(o, [A_1, A_2, \ldots, A_n], s, +\text{readAttribute}). )</td>
<td></td>
</tr>
<tr>
<td>Get Attribute List</td>
<td>( \text{dercando}(o, s, +\text{getAttributeList}) \leftarrow ) ( \text{dercando}(o, s, +\text{get}). )</td>
<td></td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>( \text{do}(\text{Attribute}, s, -\text{modify}) \leftarrow ) ( \neg \text{dercando}(\text{Attribute}, s, +\text{modify}) ) &amp; ( \neg \text{dercando}(\text{Attribute}, s, -\text{modify}). )</td>
<td></td>
</tr>
<tr>
<td>Add Attribute</td>
<td>( \text{do}(\text{Attribute}, s, -\text{addAttribute}) \leftarrow ) ( \neg \text{dercando}(\text{Attribute}, s, +\text{addAttribute}) ) &amp; ( \neg \text{dercando}(\text{Attribute}, s, -\text{addAttribute}). )</td>
<td></td>
</tr>
<tr>
<td>Delete Attribute</td>
<td>( \text{do}(\text{Attribute}, s, -\text{delete}) \leftarrow ) ( \neg \text{dercando}(\text{Attribute}, s, +\text{delete}) ) &amp; ( \neg \text{dercando}(\text{Attribute}, s, -\text{delete}). )</td>
<td></td>
</tr>
<tr>
<td>Locate</td>
<td>( \text{dercando}(o, s, R, +\text{locate}) \leftarrow ) ( \text{dercando}(o, s, +\text{get}). )</td>
<td></td>
</tr>
<tr>
<td>Destroy</td>
<td>( \text{do}(o, s, -\text{destroy}) \leftarrow ) ( \neg \text{dercando}(o, s, +\text{destroy}) ) &amp; ( \neg \text{dercando}(o, s, -\text{destroy}). )</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

KMIP XACML Policies

Root PolicySet

<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-applicable"
PolicySetId="ManagedObjectPolicy" Version="1.0"
  <Target>
  </Target>
  <PolicySetIdReference>SecretObjectPolicySet</PolicySetIdReference>
  <PolicySetIdReference>PublicObjectPolicySet</PolicySetIdReference>
  <PolicySetIdReference>TemplateObjectPolicySet</PolicySetIdReference>
  <Policy PolicyId="NoContextMatch"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable">
    <Rule Effect="Deny" RuleId="NoContextMatchDenyRule"/>
  </Policy>
</PolicySet>

SecretObject PolicySet

<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os" Version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" PolicySetId="SecretObjectPolicySet"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-applicable"
  <Target>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">SYMMETRIC_KEY</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-type" MustBePresent="true" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PRIVATE_KEY</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-type" MustBePresent="true" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">SPLIT_KEY</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-type" MustBePresent="true" />
        </ResourceMatch>
      </Resource>
    </Resources>
  </Target>
</PolicySet>
APPENDIX B. KMIP XACML POLICIES

...
<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">RECOVER</AttributeValue>
    <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
  </ActionMatch>
</Action>

<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ARCHIVE</AttributeValue>
    <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
  </ActionMatch>
</Action>

<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">DESTROY</AttributeValue>
    <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
  </ActionMatch>
</Action>

<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">REVOKE</AttributeValue>
    <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
  </ActionMatch>
</Action>
</Actions>
</Target>

<Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
        DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
    </Apply>
  </Apply>
</Condition>
</Target>

<Rule Effect="Permit" RuleId="SecretObjActivateOperation">
  <Description>Rule specifying authorization for KMIP secret object operation</Description>
</Rule>

<Description>Rule specifying authorization for KMIP secret object operation</Description>

<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">SECRET_DATA</AttributeValue>
  <ResourceAttributeDesignator AttributeId="resource-type" MustBePresent="true" />
</ResourceMatch>

</Resource>
</Resources>

<Actions>
  <Action>
    <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ACTIVATE</AttributeValue>
      <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" MustBePresent="true" />
    </ActionMatch>
  </Action>
</Actions>
</Target>

<Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id" MustBePresent="true" />
    </Apply>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <ResourceAttributeDesignator AttributeId="resource-owner" MustBePresent="true" />
    </Apply>
  </Apply>
</Condition>
</Rule>

<Rule Effect="Permit" RuleId="SecretObjModifyAttributes">
  <Target>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">NAME</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-attrId" MustBePresent="false" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">CRYPTOGRAPHICPARAMETERS</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-attrId" MustBePresent="false" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">USAGE LIMITS</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-attrId" MustBePresent="false" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ACTIVATION DATE</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-attrId" MustBePresent="false" />
        </ResourceMatch>
      </Resource>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PROCESS START DATE</AttributeValue>
          <ResourceAttributeDesignator AttributeId="resource-attrId" MustBePresent="false" />
        </ResourceMatch>
      </Resource>
    </Resources>
  </Target>
</Rule>
<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PROTECT STOP DATE</AttributeValue>
    <ResourceAttributeDesignator AttributeId="resource-attrId" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
  </ResourceMatch>
</Resource>

<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">DEACTIVATION DATE</AttributeValue>
    <ResourceAttributeDesignator AttributeId="resource-attrId" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
  </ResourceMatch>
</Resource>

<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">OBJECT GROUP</AttributeValue>
    <ResourceAttributeDesignator AttributeId="resource-attrId" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
  </ResourceMatch>
</Resource>

<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">LINK</AttributeValue>
    <ResourceAttributeDesignator AttributeId="resource-attrId" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
  </ResourceMatch>
</Resource>

<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">CONTACT INFORMATION</AttributeValue>
    <ResourceAttributeDesignator AttributeId="resource-attrId" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
  </ResourceMatch>
</Resource>

</Resources>

<Actions>
  <Action>
    <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">MODIFY_ATTRIBUTE</AttributeValue>
      <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false"/>
    </ActionMatch>
  </Action>
</Actions>

</Target>

<Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"/>
    </Apply>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:attribute:attribute-id" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"/>
    </Apply>
  </Apply>
</Condition>

</Rule>
</Policy>

<PolicySetIdReference>SecretObjectDerivedPolicy</PolicySetIdReference>
<PolicyPolicyId="NoSecretObjContextMatch"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable">
  <Rule Effect="Deny" RuleId="NoSecretObjContextMatchDenyRule"/>
</Policy>
</PolicySet>
Bibliography


