THE CONDITION-DRIVEN AUTHORIZATION MODEL FOR DISTRIBUTED SYSTEM SERVICES

by

Tatyana Ryutov

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL,
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2002
Acknowledgements

First and foremost, I wish to express my deepest gratitude to my advisor, Dr. Clifford Neuman. This dissertation would not have been possible without his continuous support, encouragement and expert guidance. I would like to thank Dr. Carl Kesselman for his help, advice, and suggestions throughout my work. I am also grateful to the other members of my dissertation committee, Dr. John Heideman and Professor Elis Horowitz, for their valuable insights and comments toward the improvement of this work. I would also like to express my thanks and appreciation to Laura Perlman who greatly improved the prototype implementation.

I would like to thank Professor Leonard Adleman whose valuable comments suggested a new approach in this work.

I have enjoyed my personal and professional association with the members of the Global Operating Systems Technology group. In particular, I wish to thank Professor Gene Tsudik for his support to pursue my goals.

I am very grateful to my husband, Alex Ryutov, who not only proof-read multiple versions of this dissertation, but also provided many suggestions to help me improve my presentation and clarify my arguments.

My sincere gratitude goes to my parents, Lyudmila and Viktor Gakharia. Partly due to my parents’ influence (both are computer scientists) I decided very early in life to pursue a research career in computer science.
Thanks go to the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation for their financial support\(^1\).

\(^1\)Grants (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DE-FCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); (4) DE-FC03-99ER25397, Diplomat project; and (5) HE1254-97, XAUTH Project.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy, the U.S. Government or the Xerox Corporation.
# Contents

Acknowledgements .......................................................... ii

Abstract ........................................................................... x

1 Introduction ..................................................................... 1
   1.1 Thesis Statement, Goals and Contributions .......................... 3

2 An Introduction to Authorization and Access Control ............... 9
   2.1 Definitions .................................................................... 9
   2.2 Security Policy Representation ....................................... 12
   2.3 Security Models .......................................................... 14
      2.3.1 Discretionary and Mandatory Access Control .............. 14
      2.3.2 MAC Confidentiality: Bell-LaPadula Model (BLP) .......... 15
      2.3.3 MAC Integrity: Biba Model ...................................... 16
      2.3.4 Integrity: Clark-Wilson Model .................................. 17
      2.3.5 Change of access rights: Chinese Wall Model .............. 18
      2.3.6 Role Based Access Control Model ............................ 19
      2.3.7 The Principal of Least Privilege ............................... 20
   2.4 Other Security Services ............................................... 20
      2.4.1 Intrusion and Misuse Detection and Response .............. 22
<table>
<thead>
<tr>
<th>2.5 Summary</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Conceptual Model</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Authentication</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Delegation of Access Rights</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Security Policy Domains</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Policy Composition</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Authorization Policy</td>
<td>30</td>
</tr>
<tr>
<td>3.6 Conditions</td>
<td>31</td>
</tr>
<tr>
<td>3.7 Evaluation of Conditions</td>
<td>35</td>
</tr>
<tr>
<td>3.8 System Variables</td>
<td>35</td>
</tr>
<tr>
<td>3.9 System State Representation</td>
<td>37</td>
</tr>
<tr>
<td>3.10 Taxonomy of Conditions</td>
<td>39</td>
</tr>
<tr>
<td>3.10.1 Read, Write and Opaque Conditions</td>
<td>39</td>
</tr>
<tr>
<td>3.10.2 Pre-, Mid-, Post- and Request-result Conditions</td>
<td>42</td>
</tr>
<tr>
<td>3.11 The Three-Phase Policy Enforcement</td>
<td>45</td>
</tr>
<tr>
<td>3.11.1 Access Control Phase</td>
<td>47</td>
</tr>
<tr>
<td>3.11.2 Execution Control Phase</td>
<td>47</td>
</tr>
<tr>
<td>3.11.3 Post-execution Actions Phase</td>
<td>48</td>
</tr>
<tr>
<td>3.12 Summary</td>
<td>48</td>
</tr>
<tr>
<td>4 A Formal Presentation</td>
<td>49</td>
</tr>
<tr>
<td>4.1 Basic Model</td>
<td>50</td>
</tr>
<tr>
<td>4.1.1 Basic Functions for Authorization</td>
<td>53</td>
</tr>
<tr>
<td>4.1.2 ACL, Capability and Commpact</td>
<td>56</td>
</tr>
<tr>
<td>4.2 Extended Model</td>
<td>58</td>
</tr>
<tr>
<td>4.2.1 Algorithm 1</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2 Summary</td>
<td>66</td>
</tr>
</tbody>
</table>
6.5 Generic Authorization and Access-control API (GAA-API) .......................... 113
   6.5.1 Callback Functions ............................................................................. 113
   6.5.2 GAA-API Concepts and Typical Usage ............................................. 114
   6.5.3 GAA-API Security Context ............................................................... 117
       6.5.3.1 Creation of the GAA-API Security Context .............................. 118
   6.5.4 Main GAA-API Functions ................................................................. 119
       6.5.4.1 The gaa_get_object_policy_info Function .............................. 119
       6.5.4.2 The gaa_check_authorization Function .................................. 120
       6.5.4.3 The gaa_execution_control Function ...................................... 122
       6.5.4.4 The gaa_post_execution_actions Function ............................... 123
6.6 Security Concerns .................................................................................. 123
6.7 History of the GAA-API Implementation ............................................... 124
6.8 Integrating the GAA-API into Applications ........................................... 124
   6.8.1 Integration with Alternative Authentication Mechanisms .................. 124
   6.8.2 Implementation of the Delegation of Access Rights ......................... 125
   6.8.3 Outside Integration ............................................................................. 129
   6.8.4 Current Work on the Application Integration ................................... 130
6.9 Intrusion and Misuse Attack-Response Examples ..................................... 132
   6.9.1 Network Lockdown .......................................................................... 133
   6.9.2 NIDS fine-tuning .............................................................................. 134
   6.9.3 Application-level Intrusion Detection .............................................. 135
6.10 Prototype Evaluation ............................................................................. 138
6.11 Summary .................................................................................................. 140

7 Related Work .............................................................................................. 142
   7.1 Relationship to Existing Models and Policy Languages ....................... 142
   7.2 Relationship to Existing Systems ......................................................... 147
8 Conclusion

8.1 Contributions ......................................................... 154
8.2 Future Directions ..................................................... 155
8.3 Concluding Remarks ................................................. 159
List of Figures

2.1 Possible realizations of the Lampson matrix. .......................... 12

3.1 Condition evaluation...................................................... 38
3.2 The policy enforcement process.......................................... 46

4.1 Condition evaluation tree.................................................. 61
4.2 Three-level hierarchy with assigned priorities.......................... 71

5.1 Chinese Wall policy example............................................ 95
Abstract

Organizations of all kinds, from publishing to health care, are turning to on-line services. As they become dependent on this new technology, many are exposed to the downside of computer networking: theft, fraud and denial of service risks. These risks must be managed by enforcing security policies.

Traditional authorization mechanisms are based on the premise that once a user is authorized to perform some operation, the access is granted unconditionally. This practice will not detect the abuse of user privileges. In a distributed multi-user environment, the security policy must not only specify legitimate user privileges but also aid in the detection of the abuse of the privileges and adapt to perceived system threat conditions.

This dissertation presents a new model that provides clear and precise semantics for authorization. The semantics is independent from underlying security mechanisms and is separate from implementation. The model has three principal features: support for adaptive policies, support for policy composition and three-phase policy enforcement.

This dissertation differs from other work in the area by concentrating on policies which can have side-effects, an aspect of the policy specification and evaluation problem that has been largely ignored until now.

The major contributions of this research is development of a Conceptual Model, which allows the specification of complex authorization policies and provides a generic policy evaluation environment. A formal presentation of the Conceptual Model based on progression of models from simple to higher complexity is described. The demonstration of
the translation of security policies across existing authorization models to the canonical representation described by the Conceptual Model is presented.

Finally, the dissertation describes the implementation of the Conceptual Model - the Generic Authorization and Access-control Application Programming Interface (GAA-API), which demonstrates the usefulness of the model. The API provides a generic framework by which applications facilitate access control decisions, policy enforcement and application-level intrusion detection and response capabilities. The design, implementation, and use of the GAA-API are described. The integration of the API with several applications is discussed.
Chapter 1

Introduction

This dissertation describes a new authorization system that allows the specification of adaptive fine-grained authorization policies enhanced with audit and notification capabilities. The system provides a uniform control interface and a generic policy evaluation environment.

As more and more enterprises make their critical information available on the Internet, whether only to employees or to end-customers, they are exposed to significant risks such as theft, fraud, and denial of service attacks. In general, the most significant consequences result from attacks within the system by otherwise legitimate users (or attackers posing as such users) performing unauthorized activities. Protective measures, such as audit analysis along with threshold control can be used to examine user actions.

In addition to having a means to detect attacks (the role of an intrusion detection system) it is essential to have well defined policies that indicate what to do under perceived or suspected attack conditions, or for that matter under suspicion of attack conditions so that data can be gathered to characterize the perceived attack.

Current access control systems are based on the premise that once a user is authorized to perform some operation, the access is granted unconditionally. This practice is not likely to detect the abuse of user privileges. To provide additional level of security checks, close monitoring of authorized actions may be necessary. The policies themselves must
automatically adapt to meet the changing security requirements in the event of possible intrusion while allowing users to operate in the changing environment. Policies can be applied to controlling execution of the requested actions.

The points of the policy enforcement may include three time phases:

1. Before requested operation starts; to decide whether this operation is authorized.

2. During the execution of the authorized operation; to detect malicious behavior in real-time (e.g., a user process consumes excessive system resources).

3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

In a distributed environment computer systems are managed by separate administrative authorities. The lack of a central authority responsible for policy specification and enforcement results in coexistence of different (possibly contradictory) security policies. The situation may be further complicated when organizations have different policy formalism and implementation platforms.

To enforce policies that several authorities intend to have in place, we must be able to reason about the composition of the policies.

Traditional authorization systems are designed to enforce a single security policy. These systems are not sufficient to address the problems of specification, enforcement and composition of security policies that arise in distributed systems.

In this dissertation, we propose a solution to these issues that allows us to cope with the growing complexity of the policy specification and enforcement in distributed, heterogeneous environments.
1.1 Thesis Statement, Goals and Contributions

To protect sensitive and critical system resources in heterogeneous, administratively decentralized distributed environment, a system must be capable of supporting advanced security policies:

1. The security policies must address the sophisticated access control requirements posed by the Internet-based services. Therefore, the policy model should be able to specify and enforce complex and fine-grained access control policies in a uniform and structured way.

2. The policies must be adaptive \(^\text{1}\) to accommodate changes in the security requirements and assist in detecting and responding to intrusion and misuse. To do so, the policies should indicate not only what activities are authorized, but also provide the means to detect abuse of user privileges. In particular, the policy should specify when audit records should be generated and allow for immediate notification.

3. Policy enforcement can be required at various time stages of the requested action. Thus, the policies should indicate when the policy has to be enforced.

4. The policies should have a set of qualifiers that capture policy evaluation properties (such as priority and composition mode) to support policy composition in a controlled and secure manner.

The goal of this work is to design an authorization system that supports the advanced security policies.

Supporting such advanced policies is a complex matter and research into facilitating the policy specification and enforcement is thus an important area. To cope with the

\(^\text{1}\)The term “adaptive” in this dissertation is used to indicate that the security policy to be enforced depends on the current state of the system, e.g., system load, system treat level or time of day (more restrictive organizational policy may be enforced during after hours).
The growing complexity of policy specification it is useful to design a conceptual model that
gives a structured way to think about policies. A model enables one to better understand
the domain of study, visualize the main elements and their behavior at some chosen level
of detail and use a short hand notation for precise description and decreased ambiguity.
Furthermore, the conceptual integrity of a system derives from a coherent high-level view
of the system organization and functionality. Thus, one of the main objectives of this work
is to construct a conceptual model for policy representation and evaluation. We describe
our model formally by using a progression of formal models starting from a simple model
and moving to a more complex model that represents the implemented system. For doing
so, we use a methodology based on concepts of sets and functions.

Here we outline major steps of the policy developing process [13]:

1. Informal policy specification.

   Informal policy specification results from analysis of the policy requirements (written
   rules) expressed in real-world terms. This includes defining the range of threats that
   an organization chooses to guard against and how these threats are dealt with when
   manifested.

2. Formal policy specification.

   Formal policy specification is a transformation of the informal specification into
   a precise abstract formulation. It should be possible to do some checking of the
   consistency between the formal and informal specifications.

3. Policy implementation.

   Policy implementation results in an executable system. At this step we must ensure
   correct and cost-effective implementation of the formal policy model.
4. Policy correction.

Policy correction and improvements to the implementation may be needed by observing policy processing and execution. This may cause changes in the real-world policy.

This work concentrates on the second third steps of the process.

There has been extensive research into authorization and a number of formal models have been developed. Some of these contributions focus on addressing authorization requirements for specific policy domains, e.g., database systems [12], collaborative environments [71] or separation of duty [2]. Others are concerned with a particular access control mechanism, such as ACL [1] or role-based access control [67]. Traditionally, authorization systems are designed to enforce a single security policy, for example some form of mandatory access control [10] used in defense applications. This approach considerably decreases the set of security requirements that can be supported. Supporting just one particular policy constraints the kinds of policy users can choose to enforce.

In addition, existing authorization systems are designed to address authorization requirements for specific policy domains, often implemented in a proprietary way (e.g., az-nAPI [6]). In networked environments, such heterogeneity leads to interoperability problems, where each system has a different control interface and application developers lack a uniform integration model.

What is still missing, is a unified view of authorization in a distributed, multi-policy environment. Such an environment is composed of connected independent computer systems managed by separate administrative authorities.
When resources are disjointly owned and administered by autonomous security domains, the following issues must be addressed:

- **Policy translation.**

  In a multi-policy environment the policy integration should incorporate diverse authorization models, which can coexist in distributed systems. Administrators of each domain might express security policies by means of different formalisms and use diverse authorization mechanisms. Common interpretation of a policy model is essential for joint administration of shared resources. Generalizing the way that applications define their authorization requirements provides the means for integration of local and distributed security policies and translation of security policies across multiple authorization models. The model should be easy to map to a number of different languages and to be extensible to allow domain-specific interpretation of policies.

- **Policy composition.**

  Policy translation alone does not solve the multi-policy security issues. Given a set of translated policies, we must be able to reason about the composition of the policies. Therefore, the model should provide mechanisms to compose policies meaningfully and to resolve conflicts among policies.

  The contributions of this dissertation include the development of a new model that allows the specification of complex authorization policies and provides a generic policy evaluation environment. In particular, our model allows us to represent existing access control mechanisms (e.g., ACL, capability) and models such as role-based access control, Chinese Wall, Clark-Wilson model, and lattice-based policies a uniform and consistent manner.

  We demonstrate the translation of security policies across multiple authorization models to the canonical representation described by the conceptual model.
Furthermore, the model provides a general basis for identifying and resolving issues, not well understood before, such as side effects of the policy evaluation on the system state and related policies. By separating generic from domain specific elements, we ensure that the model is extensible to arbitrary (authorization policy) domains.

It has been stated by [66], [77], [12] that authorization is an independent security concept that should be separated from implementation mechanisms. We take this statement further and consider authorization policy as the only first class citizen. We argue that our condition-driven approach facilitates integration of other security services, such as authentication, audit, notification and intrusion detection with applications. Selection of these other services is based on policies.

This work describes a generic authorization model that can be usefully applied to intrusion and misuse detection. The model allows the specification of fine-grained audit policies in addition to authorization policies.

Historically, support for certain security mechanisms has a significant negative effect on system performance. By using adaptive security policies that control the application of certain security methods can limit this performance degradation to only those situation where the slow methods are actually required. For example, the problem of over-auditing is avoided with the fine-grained auditing policies that control what is audited and when, based on who is accessing the system and how it is being accessed. Thus, this approach shows a practical way to tune the performance of the security services used in a system.

A further contribution of this research is the development of an actual system based on the described model - the Generic Authorization and Access-control API (GAA-API), a common access control API, which allows applications to request the authorization policy information for a particular resource and to evaluate this policy against credentials carried in the security context for the current connections. Applications invoke the GAA-API functions to determine if a requested operation was authorized or if additional checks are necessary and to perform controlled execution of the authorized operation.
The policy language that we implemented is called Extended Access Control List (EACL). The EACL is simple language designed to describe user-level authorization policy.

The EACL language is a general way of expressing many kinds of policies in a form that can be used to automatically test conditions. The GAA-API is not concerned about the meaning the conditions have to the application, it only formally evaluates the specified conditions to determine whether a given set of conditions satisfy a policy or not.

The GAA-API provides general-purpose execution environment in which EACLs are evaluated. The API is designed for extensibility: additional modules can be registered with the GAA-API to support application-specific policy semantics. EACL conditions can include rather complex programming, for example, in present implementation the KeyNote system [15] is used to evaluate one type of conditions.

EACL is one of possible policy languages supported by the GAA-API. The underlying architecture of the GAA-API allows different policy languages to co-exist in one execution environment.

Our model is applicable for a wide range of systems and applications. The GAA-API has been integrated with the several applications: Prospero Resource Manager [58], Grid Security Infrastructure [32], KeyNote [15], SSH deamon [79] and Apache server [4]. The integration of the GAA-API with FreeBSD/WAN IPsec [49] is almost completed.
Chapter 2

An Introduction to Authorization and Access Control

In this chapter, we briefly summarize the basic notions of access control that are used in the remainder of this dissertation.

2.1 Definitions

- **Object**
  
is a target of requests and it has to be protected, e.g., critical programs, files, hosts and print jobs.

- **Access Right (operation, permission, action, access mode)**
  
is a particular type of access to a protected object, e.g., read, write or execute. Specific system events, such as restarting or shutting down the system, system log-in and log-off can be modeled as access rights associated with the system, where the system is the protected object.

- **Subject**
  
is an entity (e.g., individual users, hosts, and applications) on whose behalf a request to access an object has been issued.
- **Authentication**
  is the process of validating a user's identity. Authentication is commonly done through the use of passwords or digital certificates.

- **Principal**
  is an identity associated with a subject as a result of some unspecified authentication protocol. It can refer to a person, group, host or application. Several principals can be associated with the same subject.

- **Security Policy**
  A set of rules, measures and procedures that determine the security controls imposed on the management, distribution and protection of objects.

- **Authorization (Access control) Policy**
  A set of rules, part of a security policy, by which access to protected objects is granted or denied. An authorization policy may be defined, for example, in terms of access control lists or capabilities.

- **Condition (restriction)**
  describes the context, in terms of such variables as location, time of day, etc., in which an access to an object is expressed.

- **Delegation**
  is the ability of a principal to give to another principal limited authority to act on its behalf.

- **Credential**
  is a secure store entity (usually implemented as a digitally signed certificate) that maintains security information such as, the identity, group membership, privilege attribute and transfer of privilege.
• Access Control
regulates the access to the objects in the computer system. Access Control refers to
the mechanisms and policies that restrict access to objects.

• Authorization
is defined in the following ways, depending on a context:

1. The process of granting or denying access to an object.
2. Access privileges granted to a subject.
3. An authorization policy that grants access (positive authorizations) or denies
   access (negative authorizations).

• Access Control (Authorization) Mechanism
is an operating procedure that mediates access requests submitted to the system and
determines whether the requests should be granted or denied.

• Access Control List (ACL)
A list of the access privileges that are granted to various principals for a specific
object. ACL protects objects by strictly regulating who has access to an object and
what operations can be performed.

In security literature, the terms access control and authorization are often used inter-
changeably, leading to confusion. To clarify the meaning of the terminology used in this
dissertation we provide an example:

A user requests access to a file on a server and the responsible service checks user’s
credentials against an ACL and grants the access.

The server performs access control to determine whether the request should be honored.
The process of authorization (definition 1) is based on comparing security information in
user’s credentials against the authorizations (definition 3) specified in the ACL. The server
grants the user the authorization (definition 2) to access the file.
2.2 Security Policy Representation

This section describes various policy representation methods used in existing systems and discusses their limitations.

Traditionally, policy conceptualization is based on three basic entity types: objects, access rights and subjects. Some of the possible logical groupings of these entities, such as ACL, capability and compact, have become practical implementation of the Lampson matrix [47], an abstraction that specifies the rights that each subject possesses for each object. Figure 2.1 shows possible realizations of the Lampson matrix. In the ACL-based systems, such as Unix [34] and Windows NT [70], policies are grouped by objects. A typical ACL is associated with an object (or a group of objects) to be protected and enumerates the list of authorized subjects and their rights to access the object. An access request \( q = (\text{object } o, \text{ access right } r, \text{ subject } s) \) is processed by verifying that the subject \( s \) is included in the list of trusted subjects and the access right \( r \) appears in the list of valid access rights associated with the object \( o \).
In the capability-based systems, e.g., Hydra[23] and Amoeba [75], policies are grouped by subjects. Capabilities are associated with subjects. A capability lists sets of objects accessible by the subject along with the types of access rights. An access request \( q = (\text{object } o, \text{ access right } r \text{ capability } c) \) is processed by verifying that the capability \( c \) specifies the object \( o \) and the access right \( r \) is referenced in the list of valid access rights for object \( o \).

Unlike the traditional ACL and capability abstractions that have been recognized by security community for a long time, the compact (communication pact) has been introduced just recently. In the comppact model [62], policies are grouped by subject to subject relationship. The comppact represents agreement between the requester and the owner of the requested resource. Thus, one of the subjects is an owner of the targeted objects and the other one is a party interested in the objects. A compact specifies sets of objects accessible by the subjects participating in the agreement along with the types of granted access rights.

Request of the form \( q = (\text{object } o, \text{ access right } r, \text{ subject } s_1) \) is processed in the following manner: determine an owner of object \( o \), e.g., subject \( s_2 \); find a comppact that describes the agreement between subject \( s_1 \) and \( s_2 \); verify whether the compact authorizes subject \( s_1 \) access right \( r \) to the object \( o \).

The limitations of the traditional access control abstractions become apparent when it is applied in a heterogeneous, administratively decentralized, distributed environment.

The generic access rights may not be sufficient for some applications to express authorization requirements, for example, those that might exist among different autonomous security domains. In addition, these abstractions provide no means to reason about the composition of policies. Conventional authorization concepts must be extended to allow conditional restrictions on access rights and policy composition semantics.
2.3 Security Models

In this section we provide brief description of well-known security models. Confidentiality and integrity of the information are major concerns of the computer system security. However, system security requirements depend on the intended use (military or commercial) of the system of concern.

In the military environment, the important objective is to prevent disclosure of information. The military confidentiality policy states that the classified information must not be disclosed to the unauthorized users and authorized users must not be able to declassify sensitive information.

In the commercial world, integrity of the information is very important. The commercial integrity policy states that the data must not be modified in a way that results in a fraud or error of company assets, accounting or audit records.

It has been widely accepted that there are two approaches to the system security: Discretionary and Mandatory Access Control.

2.3.1 Discretionary and Mandatory Access Control

Discretionary Access Controls (DAC) are often perceived as meeting the security needs of industry and civilian government. The underlying principal behind DAC is that a user can grant or revoke access to the protected objects that he owns.

Mandatory Access Control (or lattice-based access control) models control access to objects based on the security labeling of objects and subjects and are typically employed in multi-level secure military systems. In Mandatory Access Control (MAC), the system imposes an access control policy and object owners cannot change that policy. Mandatory policies govern access on the basis of classification of subjects and objects in the system.
2.3.2 MAC Confidentiality: Bell-LaPadula Model (BLP)

Bell and LaPadula model [10] formalizes the concept of MAC. The model has been used for enforcing access control in government and military applications. For example, a specific application of BLP is used in Multics operating system.

Mandatory policies govern access on the basis of classification of subjects and objects in the system.

These subjects and objects are assigned sensitivity classes, also called security labels that denote their hierarchical sensitivity and need-to-know attributes. Specifically, a security class consists of two components: a hierarchical security level and a possibly empty set of non-hierarchical security categories.

In general there can be any number of totally ordered security levels. The number of the security levels is finite.

Categories are independent of each other, so the set of categories is unordered. When security levels and categories are combined, the resulting security classes represent partially ordered sets.

The security class of an object is called classification. The security class of a user is called a clearance. In order to determine whether a user has access to an object, the user’s clearance is compared to the object’s classification.

The security classes form a partially ordered lattice. Access permissions are defined through an access control matrix and through a partial ordering of security classes. An access request \( q = (object\ o, access\ right\ r, subject\ s) \) is granted if and only if all of the following properties are satisfied:

1. Discretionary security property:

   The cell in the access matrix for row \( s \) and column \( o \) contains \( r \).
2. Simple security property (read down, no read up):

A user can only read an object if the security class of the user dominates the security class of the object.

3. *-property (write up, no write down):

A subject can only write an object if the security class of the subject is dominated by the security class of the object.

The security classes are fixed, they do not change with time. Thus, BLP is applicable for systems with static security classes.

2.3.3 MAC Integrity: Biba Model

Biba model [14] is similar to BLP capturing integrity aspect of mandatory access control. The hierarchy of the classes is based on the integrity rather then disclosure-oriented security. High integrity is placed toward the top of the lattice of security classes and low integrity at the bottom.

An access request \( q = (object \ o, access \ right \ r, subject \ s) \) is granted if and only if all of the following properties are satisfied:

- Discretionary security property:
  
The cell in the access matrix for row \( s \) and column \( o \) contains the requested right (read or write).

- Simple security (read up, no read down):
  
A subject's integrity class must be dominated by the integrity class of the object being read.

- *-property (write down, no read up):
  
Subject's integrity class must dominate the class of the object being written.
Implementation of both Mandatory Confidentiality and Integrity rules can be based on a single security class for both confidentiality and integrity. This would result in a read-equal and write-equal rules. The drawback is reduced flexibility of the resulting system.

2.3.4 Integrity: Clark-Wilson Model

The Clark-Wilson model [22] was developed to address commercial integrity controls.

The model uses two categories of mechanisms to realize integrity: *well formed transactions* and *separation of duty*. The objects to be protected are called *constrained data items* (CDI).

The principle of *well formed transaction* (WFT) is defined as a transaction where the user is unable manipulate data arbitrarily, but only in constrained ways that preserve the integrity of the data. A security system in which transactions are well formed ensures that only legitimate actions can be executed. The system ensures that the internal data is accurate and consistent to what it represents in the real world.

Clark-Wilson model raises issue of separation of responsibility. In a commercial environment, it is undesirable for the same person issuing an order, receiving the goods, and writing a check, because there is a potential for abuse. The required division of responsibilities is called *separation of duty*. The principle of *separation of duty* states no single person should perform a task from beginning to end, but that the task should be divided among two or more people to prevent fraud by one person acting alone.

Both static and dynamic forms of separation exist.

*Static separation of duty* indicates that the same subject can not be allowed a certain set of access rights.

*Dynamic separation of duty* enforces control over how permissions are used at the access time. A subject can potentially execute any operation in a particular set, but not all of them. By executing some of the operations, the subject will automatically rule out
the possibility of executing the others. Which of the operations a subject executes is not predetermined.

The Clark-Wilson model is defined in terms of access triplets:

< UserID, WFT_i, \{CDI_k, CDI_l, ..., CDI_n\} >.

The triplets specify the names of the subjects (UserID) and objects (CDI_k, CDI_l, ..., CDI_n) the user is requesting access to and the names of the programs (WFT_i) that implement well-formed transactions and provide the access.

Once subjects have been constrained so that they can gain access to objects only through specified WFT, the WFT can be embedded with additional logic to impose limitation of privilege and separation of duties.

2.3.5 Change of access rights: Chinese Wall Model

The Chinese wall model [20] is deployed in the operation of many financial institution where analysts deal with a number of clients and have to avoid conflicts of interest.

A consultant can chose a company in order to offer an advice. However, once the company has been chosen, the consultant is denied access to the information about all competing companies. The ability for a consultant to chose a company may be restricted by additional discretionary controls.

Essentially, the objects are grouped into company datasets. Company datasets whose organizations are in competitions are then grouped into conflict of interest (COI) classes. Each company belongs to exactly one COI class. By mandatory ruling all subjects are allowed access to at most one object belonging to each such COI class.

An access request q = (object o, access right r, subject s) is granted if and only if all of the following properties are satisfied:

- Discretionary security property:

  The cell in the access matrix for row s and column o contains the requested right r.
• Mandatory security property:

Subject $s$ can access object $o$ only if $o$ is in the same company dataset as any object already read by $s$.

or

the object $o$ does not belong to any of the COI classes of objects already accessed by subject $s$.

Unlike BLP and Biba models, access to objects in the Chinese Wall model is not constrained by classification of objects and subjects but by what object the subject already holds access rights to. Access rights of subjects change dynamically with every access operation.

2.3.6 Role Based Access Control Model

Role Based Access Control (RBAC) model [67] differentiates between user identity and the set of tasks (role) to which a user is assigned.

With RBAC, access decisions are based on the roles that individual users have as part of an organization. Users take on assigned roles (e. g., clerk and manager).

Access rights are grouped by role name, and the use of resources is restricted to individuals authorized to enter the associated role.

Under the RBAC model, users are granted membership into roles based on their competencies and responsibilities in the organization. The operations that a user is permitted to perform are based on the user’s role. User membership into roles can be revoked and new memberships established as job assignments dictate.

RBAC is often described as a form of MAC in the sense that users are constrained by organization role assignment.
2.3.7 The Principal of Least Privilege

The principle of least privilege that states: “Each principal is given minimum access needed to accomplish its task” [65] is important for meeting integrity objectives. It requires that a user be given only the privilege necessary to perform a job. Ensuring least privilege requires identifying what the user's function is, determining the minimum set of privileges necessary, and restricting the user to only those privileges. By excluding users from actions that are unnecessary for the performance of their duties, those actions cannot be used to circumvent organizational security policy.

2.4 Other Security Services

The ability to enforce advanced policies has practical importance, for example, in computational Grids [32]. Grids are large-scale distributed computing environments that enable applications to use scientific instruments, computational and information resources that are managed by diverse organizations.

System administrators contributing their resources to a Grid will require assurance that the resources are adequately protected. In a Grid setting, the security requirements include:

1. User authentication.

   Authenticated user identity is used to determine who gains access to local resources\(^1\).

2. Resource usage limits (quotas).

   A site-specific resource allocation policy specifies limits on the computational or storage resources to be consumed, such as CPU load, memory usage and disk space. The limits are taken into account when deciding whether to initiate the requested computation. Monitoring execution of the computation on a particular node must

---

\(^1\)Mutual authentication may be required to prove the server identity to the user.
be supported to ensure that the process keeps strictly to the limits imposed by the local policy.

3. Accounting and payment.

Owners of the resources may hold users accountable for the consumed resources. Accounting may include gathering information about executed computations and consumed resources. The accounting information can be used in payment models for remote service providers.

4. Audit.

Audit can provide a means to help accomplish individual accountability and provide data to be analyzed by intrusion or misuse detection systems.

5. Intrusion and misuse detection.

Grids are vulnerable to a large-scale malicious attacks that could cause disruption of the Grid services. Thus, it is essential for Grids to support detection and automatic response to intrusion attempts.


Tools for intrusion detection and fault tolerance can be driven by event services. Alert-level notification messages permit cooperative responses. For example, notification about a computation that exceeds the quotas can signal ongoing denial of service attack. The adequate preventive measures can be taken if the attack is confirmed.

Authentication, authorization, audit, notification and intrusion detection systems are interrelated and should be used together to support effective system security.

The audit data is analyzed by an intrusion detection service. This analysis could yield patterns that could be considered security threats to the computer systems and network. As both the incidence and magnitude of malicious intrusions continue to increase, the
response time to such incidents becomes critical. Timely response can be supported by a notification service that disseminates notifications about particular events to the interested parties. Thus, if suspicious behavior detected, an event notification is generated. The receipt of such notification can signal an alert to the intrusion detection service, thereby triggering higher system threat level. To complete the cycle, upon receiving the notification about the threat, the system policies will adapt by increased auditing or requiring stronger authentication method, or in some cases denying subsequent requests.

2.4.1 Intrusion and Misuse Detection and Response

Most traditional intrusion detection (ID) systems [7] take either a network- or a host-based approach to recognizing and responding to attacks. These systems look for attack signatures, specific patterns that indicate malicious or suspicious intent or deviation from a normal profile (anomaly) that indicates an attack. A network-based ID system looks for these patterns and anomalies in network traffic. A host-based ID system looks for attack signatures and anomalies in operating system audit trails or application logs.

In the general case network- and host-based ID systems provide only incomplete coverage, leaving sophisticated attacks undetected. Other disadvantages include: large number of false positives and inability to preemptively response to attacks.

This dissertation proposes approach to intrusion and misuse detection based on specifying access control policies extended with the capability to identify (and possibly classify) intrusions and respond to the intrusions in real time. Policy enforcement is performed by an access control mechanism that is called by trusted applications to evaluate and enforce the policies.
This policy-based approach to intrusion detection and response offers several important advantages over traditional host- and network-based approaches:

- **Customization.**

  Instead of having ID system look for a restricted set of pre-defined signatures or time-variant statistical profiles, this approach allows each organization to define suspicious events in terms of policies for accessing application-level objects, taking into account the organization's and application's security profiles.

- **Efficiency.**

  Using policy-based intrusion detection is potentially more efficient:

  1. Policy-based intrusion detection is less in need of constant updates than is signature- or anomaly-based intrusion detection. Both acceptable and unacceptable behaviors remain stable if policy does not change.

  2. Previously unseen legitimate behavior will not be classified as intrusion, thus reducing the number of false positives.

- **Flexibility.**

  Policy can be defined in terms of acceptable and unacceptable access patterns to protected resources. For example:

  - **Closed World policy.**

    Everything that is not explicitly authorized is unacceptable and may indicate suspicious behavior.

  - **Open World policy.**

    Everything that is explicitly denied is unacceptable and may indicate suspicious behavior.
In addition, a “Mixed World” policy may recognize some explicitly authorized access patterns as suspicious. The policy may authorize such access only on a closely monitored basis.

- Insider attacks detection.

The ability to interface with the application directly, with significant application-specific knowledge, allows application-based intrusion monitoring to detect suspicious behavior due to authorized users exceeding their authorization or exploitation of application-specific vulnerabilities. Using this approach could potentially result in detecting a custom attack that has never been observed in the past, thus reducing the number of false negatives.

- Preemptive response.

By being integrated with the application and having the ability to control the three processing steps of the requested operation, the access control mechanism can respond to suspected intrusion in real-time. For example, the mechanism can deny the operation, suspend the operation execution and notify about the success or failure of the completed operation.

The disadvantage of this approach is that it requires changes at the application layer: an application has to be integrated with the access control mechanism. However, once the integration is completed, it becomes possible to handle access control decisions and application-level intrusion detection simultaneously.

The proposed approach is based on a generic access control mechanism that can be used by a number of different applications with no modifications to the code of the mechanism. In contrast, traditional application-based ID systems are hard to manage and deploy, as one is required for each type of critical application in the enterprise.
Each of the approaches: host-, network- and application-based has its strengths and weaknesses, each is complementary to the other. A truly effective intrusion detection system will employ all three technologies.

The information generated by the application-level ID system at the access control time can be useful to fine-tune the network- and host-based intrusion detection services. Some of the advantages include:

- support detection of attacks not visible at the application level, but not detectable without the application-level knowledge,
- reduced number of false-positives,
- control the detail level of the audit logs.

Thus, it is useful to interface access control with network- and host-based intrusion detection services.

2.5 Summary

In this chapter we discussed general concepts of authorization and access control. We started with a discussion of the policy representation methods used by the majority of existing systems and discussed their limitations. We then discussed a number of traditional security models, including Bell-LaPadula, Biba, Clark-Wilson, Chinese Wall, and RBAC models. We next argued that related security services such as: authentication, authorization, audit, notification and intrusion detection should be used together to support effective system security. We conclude that integrating access control and intrusion detection can:

- help to discover sophisticated attacks that attempt to subvert critical applications;
- provide useful feedback to network- and host-based intrusion detection systems.
Chapter 3

Conceptual Model

In this chapter we discuss the structural properties of the Conceptual Model. A positive authorization defines the actions that can be performed on target object if all the associated conditions are met. A negative authorization specifies the actions that are forbidden to perform if all associated conditions are met. The negative authorizations complicate the enforcement of the policies in a system. However, there are reasons to support the provision for negative policies.

Our model is based on the closed world default principal. By default any request is denied unless explicitly authorized by a positive authorization. This approach is opposite to the open world default principal whereby a request is always granted unless explicitly denied by a negative authorization. Authorization policies are implemented on the target host by an access control mechanism.

Time dependency appears in our Conceptual Model implicitly. At each instant only a set of policies, which exists at authorization time, is considered. All future or past policies are irrelevant. Note that this does not mean that the current policy does not depend on the past or future events. Some policies must take into account the system execution history or the fact that particular event must have happened for some operation to take place. An example of practical policy taking into account occurrence of some event is “If one reads file A, then he can not read file B”. Some policies may need to know precise time
of the event occurrence, for example for audit purposes. This may require a timestamping of certain occurrences and keeping record of them.

3.1 Authentication

Traditional security thinking has been oriented toward authentication as a prerequisite for authorization. Usually authorization applies after authenticated user identity has been established.

Highly sensitive security applications, such as those involving the financial and banking industries require per-user authentication for access control, audit and accounting purposes. However, unauthenticated access is often desirable. Many web and ftp servers offer anonymous access. The anonymous access does not necessarily mean no access control at all. For example, Windows NT [70] provide means for controlling and monitoring of anonymous access. The request succeeds if the anonymous user has permission to access the requested resource, as determined by the ACL associated with the resource.

We look at the access identity as a condition rather than a separate structural entity of our model. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is “anyone can read file A if $10 is paid”. Strong user authentication method (e.g., Kerberos [59]) can be activated in response to suspicious behavior. Thus, we look at the access identity as a condition rather than a separate structural entity of our model.

3.2 Delegation of Access Rights

Delegation takes place when a principal originally authorized to have access to some object delegates access to some other principal. The main mechanism used in the access control systems (such as KeyNote [15] and SPKI [27]) is delegation of access rights with delegated
credentials. A delegated credential is a signed statement (certificate) with which a principal grants the access rights that it has to another principal. The rights can be passed forward through a chain of credentials: a service provider can issue a delegated credential to some principal, who can further re-delegate the rights to the third principal, and so on. If all the credentials in the chain delegate the same access rights, the rights are passed all the way from the issuer to the grantee of the last credential. However, the credentials may specify different conditions associated with the delegated access rights. These conditions must be satisfied before the credential is considered valid. The rights passed through the chain have the conjunction of all conditions added by each credential issuer in the chain.

Since credentials can be issued by different issuers to different recipients, they do not necessarily form simple chains. Instead credentials form a directed graph. In the graph, there may be many chains of credentials between the same pair of principals. The set of rights passed between the principals is a union of all access rights passed by each individual chain of credentials between them.

3.3 Security Policy Domains

A security policy domain is a set of objects collected together so that an administrator can manage those objects as a unit. Each domain is associated with a particular security policy, which significantly reduces the level of administration necessary in a distributed system.

A domain is an administrative boundary because administrative privileges do not extend to other domains. It is a security boundary because each domain has a security policy that extends to all objects within the domain. In a distributed environment the security policy domains are managed by separate administrative authorities. The lack of a central authority responsible for policy specification and enforcement results in coexistence of diverse security policies.
Access control decisions will require integration of different sets of policies associated with the domain providing resources, the domain requesting resources, and individual users within each domain.

Some objects may belong to more than one security policy domain. Existence of several policy domains introduces the problem of expressing policies in a way that allows them to be composed.

### 3.4 Policy Composition

The composition of the various security policies into a coherent system-wide policy is a major challenge in the management of distributed systems necessary for users and administrators to have a means of expressing their security policies. In a distributed heterogeneous environment, policy requirements can easily be inconsistent. For example, that the same object may be governed by two conflicting policies: one policy allows access to the object during the business hours, the other allows access to the object only in the after hours. Thus, the model should support the composition of policies and conflict resolution between policies.

Woo and Lam [77] describe two types of policy composition: vertical (hierarchical) and horizontal (peer to peer).

*Vertical Policy Composition:* Policy conflicts may occur when policy domains overlap: domain administrators impose domain-specific policies and users of this domain may define individual user policies. Or a policy officer may delegate his responsibilities to a number of subordinate administrators. In both cases the policy authorities are hierarchically related in a supervisor-subordinate fashion.

The Vertical Policy Composition combines policies starting with the root policy (more general policy) and leaf policies (more specific policies).
**Horizontal Policy Composition:** Another source of policy conflicts is interaction of policies from different policy domains.

Multiple parties (stakeholders) can control resources with authority to grant access to the resource, e.g., Akenti [76]. An on-line scientific instruments, for example, X-ray laser at a university may have several stakeholders, each of which brings its own set of authorization requirements.

Horizontal Policy Composition allows each stakeholder to enforce its access control requirements independently of the other stakeholders.

All policy authorities have the same priority. Horizontal Policy Composition combines all policies that complement each other.

The difference between the two approaches is in resolution of policy conflicts.

### 3.5 Authorization Policy

In this section we explore the notion of a policy and abstract it into a Conceptual Model. We start the design of the Conceptual model with specification of the components that are to be modeled. At the conceptual level an authorization policy is a compound entity that regulates access to objects. We assume that some effective organizational security policy specifies: the resources that require protection; the real and perceived threats against those resources and intrusion response actions.

The notion of an object is central to the policy definition. An object is a target of requests and it has to be protected. An object can be a physical resource such as a host or a communication channel, as well as an abstract, higher level entity, e.g., a bank account.

An access right is a particular type of access to a protected object, e.g., read or write. In our model, rights are treated as uninterpreted symbols because the precise nature of the right is implementation and domain specific. Access rights are associated with an object, more precisely an object has a set of applicable rights. Creation of an object creates an
associated set of rights. We consider the creation of the relevant access rights to be an automatic process (a property of an object), rather than the responsibility of a policy administrator.

In our model, we do not allow rights where the underlying action is either infeasible or does not make sense for an object. For instance, in Unix [34] file protection mechanism, access right “withdraw” is not supported. The notion of negative rights is useful to specify many practical policies. Sometimes it is easier to allow access to all and explicitly disallow access for those who should not have access.

A condition describes the context in which each access right is granted. A condition must be satisfied in order to allow an operation to be performed on a target object. However, if the access right is negative, the access is denied if all conditions are met.

The role of the policy administrator is to use the existing rights and conditions to specify the desired policy.

3.6 Conditions

Here we list some of the most important conditions [57]. Some of the conditions are useful in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

- access identity

This condition specifies an authenticated access identity. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is "anyone can read file A if $10 is paid".
• **strength of authentication**

This condition specifies the authentication mechanism or set of suitable mechanisms for authentication. Strong user authentication method (e.g., Kerberos [59]) can be activated in response to suspicious behavior.

• **time**

This condition specifies time periods for which access is granted.

• **location**

This condition specifies location of the user. Authorization is granted to the users residing on specific hosts, domains, or networks.

• **message protection**

Required confidentiality/integrity message protection. This condition specifies a level or mechanism that must be used for confidentiality or integrity if access is to be granted.

• **payment**

This condition specifies a currency and an amount that must be paid prior to accessing an object.

• **quota**

Specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.

• **audit**

Enables automatic generation of audit data in response to access requests. Audit can provide a means to help accomplish individual accountability, such as cost-accounting charges and provide data to be analyzed by intrusion or misuse detection systems.
An audit record should include sufficient information to establish what event occurred and what caused the event. In general, an event record should specify when the event occurred, identity of the user requested the access, the operation requested, the object to be accessed and the result (whether the request was granted or denied or whether the requested operation was successful or not).

- **notification**

  This condition enables automatic generation of notification messages (alerts) in response to access requests. Specifies the receiver and the notification method. Automated pager, e-mail can convey critical-event notifications, pop-up messages or sound alarm.

- **threshold**

  This condition specifies allowable threshold.

- **system threat level**

  This condition specifies the system threat level. For example, low threat level means normal system operational state, medium threat level indicates suspicious behavior and high threat level means that the system is under attack. This condition can be used to activate defensive measures in response to the perceived system treat level. For example, consider authorization policy: “Tom can log-in to host A if the system threat level is low. If the system threat level is medium, the audit record about log-in information must be generated. If the system threat level is high, Tom can not log-in.”

- **trust constraints**

  This condition specifies restrictions placed on security credentials. Allows one to validate the legitimacy of the received certificate chain and the authenticity of the specified keys.
• attributes of subjects

This condition defines a set of attributes that must be possessed by subjects in order to get access to the object, e.g., security clearance or user age.

• privilege constraints

This condition orders a subject to operate with the reduced set of privileges, as a protection against malicious or accidental abuse of the privileges. In general, a subject may belong to different groups and roles. In many systems, a subject by default can operate with the union of privileges of all groups and roles to which it belongs, as well as all of its individual privileges. This condition is used to implement the principle of least privilege and dynamic separation of duty.

Failure of some of these conditions may signal suspicious behavior. For example, access is requested at unexpected times or unusual locations, violations of user quotas, repeated failure of access attempts and exceeding a threshold. Some conditions can trigger defensive measures in response to perceived system threat level. For instance, impose a limit on resource consumption, advanced payment for the allocated resources or increased auditing. In the case of insider misuse (particularly if the intruder's identity has been established) it may be appropriate to let the attacks continue under special conditions. For example, it may be desirable to initiate data collection mechanisms to gather detailed information about user activities that could serve as evidence for possible prosecutions.

The combination of conditions of different types can be used to fine tune audit and notification services. The audit detail and number of alarms should be sensitive to the system threat profile. For example, low system threat level should result in reduced alarm level and amount of generated audit data. It should also depend on the sensitivity of the requested operation and target object.

To enforce policies that several authorities intend to have in place, one must be able to reason about the composition of the policies.
3.7 Evaluation of Conditions

Note that in the implementation, some of these conditions might have side effects. For example, evaluation of payment and quota conditions may reduce a balance. Evaluation of notification condition results in sending a message, which is useful in audit.

Unfortunately, side effects complicate the model. Ignoring the side effects might cause problems when the side effects create a feedback loop, e.g., when an audit record triggers a network threat detection which affects the evaluation of subsequent policies, or where payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Another problem caused by the side effects, is possible inconsistency of the authorization result. For example, consider a policy “Tom can shut down host H only if a notification is sent (notification) and system threat level is low (system\_threat\_level:low)”. Assume that the current system threat level is low. Assume that the notification about Tom shutting down the host triggers high system threat level (this may indicate attempted denial of service attack). There are two ways to evaluate the conditions: first system\_threat\_level:low then notification. This evaluation order results in access grant. Another way is to evaluate notification condition first then system\_threat\_level:low. This evaluation order results in the denial of the access.

3.8 System Variables

All side effects of the condition evaluation are recorded in the corresponding system variables. At the lowest level, a system variable is an abstraction for bits or bytes in the system that change as the result of system execution. For example, to model a system variable affected by the evaluation of the notification condition (a message must be sent), we need better level of abstraction. Thus, a system variable is an abstract notion of a system
entity that represents a data item, e.g., a file, a message or a record in a database. Each system variable has a name and a value.

Generally speaking, security policies and authorization requests can be represented as system variables as well. However, we decided to model them as separate entities and restrict the system variables to represent only the information manipulated by the condition evaluation functions.

We assume that there exists a set of software components $S$. Each software component $s \in S$ can access system variables of particular type. For example, a system variable, which represents a file is accessed by a file system. A system variable, which represents a notification is accessed by a notification protocol, or a transport protocol, such as e-mail or HTTP.

We assume that each software component $s$ has abstract $Read$ and $Write$ operations as a part of its functionality. The read operation $s.Read(X)$ returns the value of the system variable $X$. The write operation $s.Write(X, v\_new)$ assigns a new value $v\_new$ to the system variable $X$.

As the result, the abstract $s.Read(X)$ and $s.Write(X, v\_new)$ operations return $T$ (success), $F$ (failure) or $U$ (uncertain). Uncertainty is introduced into our model by lack of adequate information to evaluate condition. For example, the value of a system variable may be undefined due to condition dependency on an event that has yet to happen, or due to a network failure.

Another source of uncertainty is inability to find the corresponding condition evaluation function, for example if the functions $s.Read(X)$ or $s.Write(X, v\_new)$ are not implemented. Sometimes, it is convenient to return some of the conditions unevaluated for further evaluation by the calling application. Uncertainty at the conceptual level has to be mapped to $T$ or $F$ at the implementation level. In the end, the access must be either granted or denied.
The system variables manipulated by the *Read* and *Write* operations, as well as the operations themselves can be either local or remote. However, our model requires that the *Read* and *Write* operations must be implemented as atomic actions.

### 3.9 System State Representation

To discuss side-effects produced by evaluation of some conditions, we introduce time into our model explicitly. Time is discrete and is represented by a totally ordered set of natural numbers. Each number corresponds to a discrete *time interval*. A time interval begins when a condition evaluation starts and it ends when the condition evaluation is completed with the resulting *T/F/U*. This means that the duration of the time intervals can vary.

All policies for evaluation are collected at the time interval $t_i$ that corresponds to the arrival time of the authorization request $q_i$. We call this time interval *authorization time*.

The condition evaluation process can spawn several time intervals. During this time new policies could be created and the existing ones deleted. In our model, however, we ignore the dynamic behavior of the policies and consider only the policies collected at the authorization time.

Temporal precedence dependencies between authorization requests arise when one step must be completed before the next can begin. These dependencies are a common in business procedures. For example, a credit history verification is required prior to approval of a loan. These two steps must occur in sequence: first the access to the client’s credit history must be obtained, next the permission to grant a loan has to be acquired (the information from the credit history might be needed for evaluation of the request to grant a loan).

To simplify our presentation, we assume that authorization requests do not overlap. The effects of the requests are resolved by serialization, in which the dependent requests are ordered by the cause-effect ordering.
Similarly, we assume that conditions are evaluated consecutively. These two assumptions enable us to concentrate on a single condition evaluation per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes. Figure 3.1 illustrates our representation. All side effects of condition evaluation are recorded in the corresponding system variables.

$S^i$ is a finite dynamic nonempty set of system variables:

$$S^i = \{s_1^i, s_2^i, \ldots, s_n^i\}.$$  \hfill (3.1)

System state $S^i$ is labeled by the time interval $i$ and represents all the information that has been deduced up to the time interval $t_i$.

The information is partial, since some system variables can be undefined at some time intervals.
The dynamic behavior of the system state is formalized by a sequence of system states $S^1, S^2, \ldots, S^k$.

At each time interval $i$ there is a transition from the current system state $S^i$ to the new system state $S^{i+1}$. Each transition is characterized by updating the values of some system variables. The variables can change not only as the result of condition evaluation but also because of other events, e.g., system load is altered.

### 3.10 Taxonomy of Conditions

In this section we present the classification of conditions based on: (i) read or write a system variable property; (ii) the time of the condition enforcement (before, during or after operation execution).

#### 3.10.1 Read, Write and Opaque Conditions

At the conceptual level, all conditions can be categorized as:

- Conditions that require reading some system variable and comparing it with the information specified in the policy. For example, evaluation of the time condition requires obtaining current time and checking if it fits into the time interval specified in the policy. We call this category of conditions **read conditions**.

  A read condition is represented as $X \text{op} P$, where $X$ is the name of a system variable, $P$ is a constant and $\text{op}$ is the operation to be performed on the value of the system variable $X$ and the constant $P$.

  At the conceptual level, we assume that the value for the $X$ is passed with the request. In implementation, this value maybe either obtained from the request or read using the $s.Read(X)$ operation during the condition evaluation.
Condition evaluation function for read conditions returns the result of applying operation \( op \) to the value of the system variable \( X \) and constant \( P \) (\( U \) is returned if the value of \( X \) is undefined).

We restrict condition expressiveness to guarantee policy computability and polynomial-time decidability, the read conditions can be represented in five ways:

- **Numerical Comparison**
  
  Some conditions, such as **system load**, can be represented numerically. These conditions are evaluated by comparing numbers (natural, integer or real). Therefore, we can define the set of operations as \( op \in OP, \ OP = \{=, \neq, <, >, \leq, \geq\} \).

- **String Matching**
  
  Some conditions, such as **access identity**, can be represented as strings. These conditions are evaluated by comparing strings of characters. Therefore, we can define the set of operations as \( op \in OP, \ OP = \{=, \neq, <, >, \leq, \geq\} \).

- **Regular Expression Matching**
  
  Another way to specify a condition, is through the use of regular expressions.
  
  Regular expression matching allows one to test whether a string \( X \) fits into a specific structure, defined by the string \( P \). Operation \( op \) reads as “is in”. Constant \( P \) denotes a regular expression over an alphabet \( A \), which specifies language \( L(P) \). \( A \) is an alphabet from which the strings we will be searching through are composed. The condition evaluation function returns \( T \) if the string \( X \) is in the language \( L(P) \) and \( F \) otherwise.

- **Set-theoretic Membership and Equality**
  
  Conditions can be represented as a set of strings, set of numbers (natural, integer or real) or set of regular expressions. Evaluation of such condition includes checking the set membership or equality. Therefore, we can define the set of operations as \( op \in OP, \ OP = \{=, \neq, \in, \notin, \subseteq, \supseteq, \exists\} \).
For example, `days_of_week: week days` condition can be represented as a set of strings, e.g., \( X \in P = \{ \text{Monday, Tuesday, Wednesday, Thursday, Friday} \} \).

The condition `time_interval: from 8AM till 6PM` can be represented as a subset of real numbers, e.g., \( X \in P = \{ p \in R \mid 8 \leq p \leq 18 \} \).

- Delegation of Access Rights
  This condition specifies a principal that can delegate the access right. For example, condition \( X \leftarrow P \) specifies principal \( P \) can delegate the access right (with which the condition is associated) to principal \( X \). The evaluation of this condition succeeds if there is a chain of delegated credentials that passes the requested access right from the principal \( P \) to principal \( X \).

- Conditions that require writing some information (e.g., audit) or initiating some action (e.g., notification). We call this category of conditions write conditions. A write condition is represented as `X new value`, where \( X \) is the name of the system variable and `new value` is the new value to be assigned.

- The third condition type is called opaque conditions. The structure of these conditions is not known to the authorization mechanism. These conditions represent domain-specific semantics that are evaluated using opaque internal procedures as part of the implementation.

Condition evaluation function for write conditions returns \( T \) if the `s.Write(X, v new)` succeeds, otherwise it returns \( F \).

An obvious relationship between the read and write conditions is if one condition requires reading of a system variable, which is written by the other condition. In our model, the condition evaluation process is totally ordered. The order has to be assessed before condition evaluation starts. Determining the correct order of the conditions in the policy statement is an important issue. Human judgment is a necessary component in this process. We feel that the function of defining the condition order can be best served by
having the policy officer chose a meaningful condition order. In particular, whether the write conditions must be evaluated before or after the read conditions. The goal of the model is to faithfully implement the given organizational security policy.

3.10.2 Pre-, Mid-, Post- and Request-result Conditions

An authorization policy may specify conditions that must be satisfied before, during or after the access right is exercised. Furthermore, evaluation of some conditions must be activated only if authorization request is granted (or denied).

For example, it may be desirable to enforce the following policy: “A process can be run on the host A if the request originates from domain B and the process does not use more than 20% of the CPU. An audit record about the started process must be generated. If the request fails, a notification should be sent to system administrator”. This policy specifies several conditions:

1) location
This condition must be satisfied before the access right “process run” is granted.

2) threshold
This condition must hold while the process is running.

3) audit
This condition must be met after the process is started.

4) notification
This condition must be met only if the request is denied.

Thus, all conditions are classified as:

- **pre-conditions** specify what must be true in order to grant the request. This means that the requested operation is allowed to be executed on the target object. If any of the pre-conditions fails, authorization is denied. Examples: time and access identity conditions.
• **request-result conditions**

In general, these conditions must be activated whether the authorization request is granted or whether the request is denied. Some request-result conditions must be evaluated whether some pre-condition fails or whether the pre-condition succeeds. The value of a request-result-condition have a flag that indicates whether the condition should be evaluated on success or on failure of the request or a particular pre-condition.

• **mid-conditions** specify what must be true during the execution of the requested operation. The mid-conditions can be used for the protection of the critical operations and resources. The mid-conditions allow for real time active monitoring of the operation execution and response. If any of the mid-conditions fails, the operation execution must be affected. The countermeasures are defined in the response methods of the target object. Aggressive responses may include direct countermeasures, such as closing the connections or suspending the processes. This is important to enforce countermeasures against serious attacks. For example, a processes consuming excessive system resources (CPU, memory, and disk space) may indicate impending denial of service attack. More passive responses may include the activating of integrity-checking routines to verify the operating state of the target.

The mid-conditions that we consider in our framework are limited to a set of thresholds, such as duration of connection, CPU and memory usage and severity metrics (e.g., current system threat level).

• **post-conditions** specify what must be true on the completion of the operation execution. The post-conditions can be specified in two ways:

1. The post-conditions that are activated only if the requested operation succeeds.

   These conditions are useful to correctly implement the enforcement of, for example, the payment/quotas constraints.
Here are some examples of the policies with post-conditions:

“A user must pay $1 to read a file. The money must be withdrawn from the user account only after successful file access.”

In this policy, the payment condition must be implemented as a post-condition. If the file read fails for technical reasons (the server crashes in the middle of the read operation), the payment condition is not activated and the user does not lose his money.

“A user is allowed to access file A only once.”

Similarly, the quota condition in this policy must be implemented as a post-condition to ensure that the user can access the file at least once.

2. The post-conditions that are activated only if the requested operation fails. For example, failure of critical operations, such as system shut down may indicate denial of service attack and require immediate notification.

The value of a post-condition have a flag that indicates whether the condition should be evaluated on success or on failure of the execution of requested operation.

The post-conditions along with the request-result conditions are useful to fine tune audit and notification services.

In our model, the whole policy is represented by a set of pre-conditions, a set of request-result conditions, a set of mid-conditions and a set of post-conditions. Any of the sets of conditions, can be empty.
3.11 The Three-Phase Policy Enforcement

The enforcement of the advanced security policies is partitioned into three successive phases.

1. Phase one: access control.
   The pre- and request-result conditions are evaluated during this phase and the decision to grant or deny the access to the requested object is made. The authorization function implements this phase.

2. Phase two: execution control.
   The access to the target object is granted, the requested operation is started and the mid-conditions are evaluated during this phase. This phase allows the controlled execution of the requested operation. The execution.control function implements this phase.

   The post-conditions are evaluated during this phase. The specified actions are performed after the operation is finished. We do not call this phase “post-execution control”, since neither failure nor success of a post-execution action can affect either access decision, or operation execution. The post.execution.actions function implements this phase.

Three status values that describe the policy enforcement process are returned:

1. authorization status $S_a$.
   Indicates whether the request is authorized ($T$), not authorized ($F$) or uncertain ($U$).

2. mid-condition enforcement status $S_m$.
   Indicates the evaluation status of the mid-conditions: met ($T$), not met ($F$) or uncertain ($U$).
3. post-condition enforcement status $S_p$.

Indicates the evaluation status of the post-conditions: met ($T$), not met ($F$) or uncertain ($U$).

Initially the status values are set to $U$.

The policy enforcement process is shown in Figure 3.2. The authorization mechanism evaluates the policies using the current system state. The system state is needed to evaluate authorizations that contain system variables as parameters, e.g., those that contain time and access identity conditions. By a system state we mean not only information describing a particular computer system such as system load, network bandwidth consumption, number of available processors, but also all security-relevant information about real world which is representable in a computer system. For example, bank account balance, temperature and user identity.
3.11.1 Access Control Phase

The access control phase starts with receiving a request \( q = (o, r, context) \) to access object \( o \), requested type of access \( r \) and contextual information (e.g., user identity, current time).

Before a particular access right \( r \) can be performed on an object \( o \), the request must be submitted to an authorization mechanism.

To make the authorization decision, the authorization mechanism first obtains relevant policies that describes authorizations for the object \( o \) and access right \( r \) from the policy database. If no relevant policy found, the authorization status \( S_a \) is set to \( F \) and the request is rejected.

Next the authorization function is called to evaluate pre- and request-result conditions. If there are no pre-conditions (this means that the requested right is granted unconditionally), the authorization status is set to \( T \). Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status \( S_a \).

If request-result conditions are present in the policy, the conditions are evaluated and the intermediate result is stored in variable \( X \). The conjunction of the \( X \) and \( S_a \) is stored in the authorization status: \( S_a = X \land S_a \). If authorization is not granted (\( S_a \neq T \)), the request is rejected.

3.11.2 Execution Control Phase

The execution control phase consists of starting the operation execution process and calling the execution control function.

If mid-conditions are present, the conditions are evaluated. Some mid-conditions are evaluated just once \(^1\), other mid-conditions are evaluated in a loop until either the operation finishes or any of the mid-conditions fails. In the latter case, the operation execution

\(^1\)E.g., locking a file to place a hold on user account.
is suspended and the reactive actions are started. The mid-conditions can be returned unevaluated to be enforced by application. The result is stored in $S_m$.

3.11.3 Post-execution Actions Phase

During the post-execution actions phase the $post\_execution\_actions$ function is called. The operation execution status (indicating whether the operation succeeded/failed) is passed to the $post\_execution\_actions$. If no post-conditions are found, the $S_p$ is set to $T$, otherwise the post-conditions are evaluated and the result is stored in $S_p$.

3.12 Summary

This chapter presented a Conceptual Model for authorization in distributed environments. Among the features of the model are support for conditions that read and write system state, Vertical and Horizontal policy composition and the three-phase policy enforcement scheme. We presented a taxonomy of conditions based on the time of enforcement and the read/write system state property. We also discussed some representative conditions. These conditions can be used to detect and respond to intrusion and misuse attacks. The Conceptual Model will be formally described in the next chapter.
Chapter 4

A Formal Presentation

In this chapter we discuss a formal presentation of our Conceptual Model based on the set and function formalism. The approach to our formal presentation is based on the "moving from simplicity to complexity" strategy. We begin with the simplest model and build complexity gradually ending with the model that more accurately represents the implemented system.

Advantages of this approach:

- The progression allows model boundaries, levels of aggregation, leverage points to be understood by developing their functional importance step by step. They can be discussed as they come into play.

- Learning the development of the model helps one to understand the system the model represents. The progression of models from simple to higher and higher complexity allows both modeler and user to fully understand the entire model.

An additional benefit is increased understanding of the basic concepts of system dynamics.

- Simple models are valid models that can be used to meet the needs of uncomplex policy environments.

- Incremental deployment of more complex model.
We describe five models:

1. Basic Model

We begin with a simplest model that introduces the basic notions and serves as a starting point to approach the problem. We assume only one condition (a pre-condition) in a policy statement and consider only positive access rights.

2. Extended Model

An initially defined model is gradually elaborated to represent more expressive policies by supporting (i) conjunction and disjunction of conditions in a policy statement; and (ii) pre-, request-result-, mid- and post-conditions.

3. Model with Negative Rights

The Extended Model is then modified to support negative access rights.

4. Model with Priorities

The model is further modified by adding new features; namely priority and composition mode to support the policy composition.

5. EACL Model

Finally, we present an EACL Model that provides more complete representation of the system involved.

During the progression of the models, some concepts, as required, can be added to the model, while some may be removed. We attempt to maintain the consistent naming of the sets and functions throughout the model progression.

4.1 Basic Model

In this section, we introduce the basic concepts of our formal representation. We will show in the subsequent sections how these concepts can be extended to support more complex authorization policies.
We start by defining sets of elements. Let \( O \) denote the set of objects, \( R \) the set of access rights, \( C \) the set of conditions, \( P \) the set of policy statements (or authorizations), and \( Q \) the set of authorization requests.

Given a set \( S \), we use the notation \( card(S) \) to denote the cardinality of the set \( S \). All the sets, except for \( C^1 \), are finite and dynamic. The dynamic property means that sets are not fixed, new elements can be added and existing elements can be deleted. The finite property assumption requires that at any particular time, the sets are finite. The empty set, which has no elements at all, is denoted by \( \emptyset \).

\( O \) is nonempty set of object elements:

\[
O = \{ o_1, o_2, \ldots, o_n \} .
\]

(4.1)

\( R \) is nonempty set of access right elements:

\[
R = \{ r_1, r_2, \ldots, r_n \} .
\]

(4.2)

\( C \) is set of condition elements with a special condition element \( c^* \), which represents an empty condition:

\[
C = \{ c^*, c_1, c_2, \ldots, c_n \} .
\]

(4.3)

\( P \) is set of compound elements:

\[
P = \{ p_1, p_2, \ldots, p_n \} .
\]

(4.4)

\(^1\)The conditions can be represented by different entities, including numbers, so we can not state finiteness property.
Each element $p$ of the set $P$ expresses an authorization that grants access right $r$ to access object $o$ under the condition $c$. Formally, a policy element $p$ is an ordered triple:

$$p = < o, r, c >, \ o \in O, \ r \in R, \ c \in C .$$  \hspace{1cm} (4.5)$$

Note that when $c = c^*$ the rights are granted unconditionally. An example of a practical policy with an empty condition is: “anyone can read file $A$”.

$Q$ is finite dynamic set of authorization requests:

$$Q = \{ q_1 , q_2 , \ldots , q_n \} .$$  \hspace{1cm} (4.6)$$

Each element $q$ of the set $Q$ is an ordered triple:

$$q = < o, r, c >, \ o \in O, \ r \in R, \ c \in C .$$  \hspace{1cm} (4.7)$$

The elements correspond to the target object ($o$), requested access right ($r$) and a condition constant ($c$). The condition constant $c$ represents context, which is matched to the requirements, specified in the condition of the relevant policy statement. In practice, the context can be represented by a set of credentials, e.g., authenticated user identity. For example, a policy statement “Tom ($c$) can read ($r$) file $A$ ($o$)” specifies an access identity condition. Consider an authorization request “read ($r$) file $A$ ($o$) on behalf of Tom ($c$)””. The condition constant “Tom” specifies authenticated user identity. The constant is matched against the condition in the policy statement.

We adopt a three-valued logic to deal with incomplete data (not known at the authorization time). Three-valued logic is classical Boolean (true/false) logic extended with a third truth-value - undefined. SQL [19] is an example of a practical system that uses three-valued logic. SQL acknowledges that data can be incomplete or inapplicable and that the truth value of a predicate may therefore be unknown.
We define a set of truth values $B$ consisting of three distinct constants:

$T$, called truth,

$F$, called falsity,

$U$, called uncertainty.

$$B = \{T, F, U\}. \quad (4.8)$$

Table 1 shows the truth tables, when at least one argument is equal to $U$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$P &amp; Q$</th>
<th>$P \lor Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$U$</td>
<td>$U$</td>
<td>$T$</td>
</tr>
<tr>
<td>$U$</td>
<td>$T$</td>
<td>$U$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$U$</td>
<td>$F$</td>
<td>$U$</td>
</tr>
<tr>
<td>$U$</td>
<td>$F$</td>
<td>$F$</td>
<td>$U$</td>
</tr>
<tr>
<td>$U$</td>
<td>$U$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
</tbody>
</table>

Table 4.1: Truth values.

We view the authorization mechanism as a two-place function called $authorization$ that given a set of policies $P$ and an authorization request $q$, returns an authorization decision $b$:

$$authorization(P, q) = b, \; q \in Q, \; b \in B.$$ 

The truth value $b$ denotes whether the request $q$ is granted ($T$), denied ($F$) or neither ($U$) by the set of policies $P$.

### 4.1.1 Basic Functions for Authorization

Next we define functions to express the authorization process.

First the authorization mechanism must find relevant policy statements representing the authorizations to access the requested object. To represent this process, we define a two-place function $by\_object$ that takes a set of policy statements $P$ and an authorization
request \( q \) (\( q \) contains target object \( \hat{o} \)) as arguments and returns a subset of the set \( P \) where the target object appears.

\[
by_{\text{object}}(P, q) = \{ < \hat{o}, r, c > \in P \mid \hat{o} \in q \}, \ q \in Q . \tag{4.9}
\]

Next the authorization mechanism must find policy statements granting the requested access right. The \( by_{\text{right}} \) is a two-place function that takes a set of policy statements \( P \) and an authorization request \( q \) as arguments and returns a subset of the set \( P \) where the requested right appears.

\[
by_{\text{right}}(P, q) = \{ < o, \hat{r}, c > \in P \mid \hat{r} \in q \}, \ q \in Q . \tag{4.10}
\]

For the sake of completeness, we define function \( by_{\text{condition}} \), which takes the set of policies \( P \) and an authorization request \( q \) (\( q \) contains a particular condition \( \hat{c} \)) as arguments and returns a subset of the set \( P \), where this condition appears.

\[
by_{\text{condition}}(P, q) = \{ < o, r, \hat{c} > \in P \mid \hat{c} \in q \}, \ q \in Q . \tag{4.11}
\]

The function \( get_{\text{policy}} \) summarizes the entire process of collecting the relevant policy statements. In implementation, the policy statements may reside on a local machine or may be distributed. The policy can be presented with the request or it can be retrieved by the \( get_{\text{policy}} \) function (possibly from various places). The \( get_{\text{policy}} \) is a composite two-place function that takes a set of policy statements \( P \) and an authorization request \( q \) as arguments and returns a subset of the set \( P \) where the target object and requested access right appear.

\[
get_{\text{policy}}(P, q) = by_{\text{right}}(by_{\text{object}}(P, q), q) = \{ < \hat{o}, \hat{r}, c > \in P \mid \hat{o}, \hat{r} \in q \}, \ q \in Q . \tag{4.12}
\]
Note that it is possible to define the \texttt{get\_policy} function in the following way:

\[
\texttt{get\_policy}(P, q) = \texttt{by\_object}(\texttt{by\_right}(P, q), q).
\]

The returned subset of the set \( P \) will be identical to the one returned by the \( \texttt{get\_policy}(P, q) = \texttt{by\_right} (\texttt{by\_object}(P, q), q). \)

However, in practice, the number of objects to be protected is usually much greater than the number of access rights. So, it is more efficient to select the policies associated with the target object first and then further select the policies with the requested right.

Next an authorization mechanism has to evaluate the condition in the policy statement using the context passed with the request. The \( \texttt{eval\_cond} \) is a two-place condition evaluation function that takes a condition constant and an authorization request as arguments and returns a truth value.

\[
\texttt{eval\_cond}(c, q) = b, \ c \in C, \ q \in Q, \ b \in B .
\]  

(4.13)

The \( \texttt{eval\_policy} \) function is a two-place function that takes a policy statement \( p \) and an authorization request \( q \) as arguments; applies the \( \texttt{eval\_cond} \) to the condition constant from the policy statement \( p \) and returns a truth value.

\[
\texttt{eval\_policy}(p, q) = \texttt{eval\_cond}(c, q) = b, \ c \in p, \ q \in Q, \ b \in B .
\]  

(4.14)

The \( \texttt{get\_policy} \) function may return several policy statements with the matching object and access right. The condition constants in these policy statements have to be evaluated and the results have to be combined. This is done by a two-place function \( \texttt{eval\_policy\_set} \) that applies the \( \texttt{eval\_policy} \) function to each policy statement and takes a conjunction of the returned truth values. Thus, all policy statements returned by the \( \texttt{get\_policy\_set} \) function must be satisfied in order for the authorization to be granted.
\[ \text{eval}_{-}\text{policy}_{-}\text{set}(P, q) = \]
\[ \text{eval}_{-}\text{policy}(p_1, q) \land \text{eval}_{-}\text{policy}(p_2, q) \land \ldots \land \text{eval}_{-}\text{policy}(p_n, q) = b, \ q \in Q, \ b \in B. \quad (4.15) \]

It is possible to use disjunction \( \lor \) of the policy statements instead of \( \land \). Operator \( \lor \) preserves the authorized access of each policy statement: if at least one of the policy statements is satisfied, authorization is granted. We use the partial disjunction of authorizations in the representation of an EACL (see Section 4.5) for convenience. However, we find the purely disjunctive approach nonintuitive.

The \textit{authorization} is a two-place composite function:

\[ \text{authorization}(P, q) = \text{eval}_{-}\text{policy}_{-}\text{set}(\text{get}_{-}\text{policy}(P, q), q) = b, \ q \in Q, \ b \in B. \quad (4.16) \]

### 4.1.2 ACL, Capability and Commpact

We now describe how policy representations discussed in Section 2.2 are represented in our model.

In the ACL-based systems, policies are grouped by objects. The \textit{by}_{-}\textit{object} function returns all policy statements associated with the given object \( \hat{o} \). The returned set of policies conceptually represents an ACL associated with the object \( \hat{o} \).

An ACL is associated with each object, so the object is implicit and can be omitted from the policy elements \( p \). Thus, each policy statement contains only elements, which represent \textit{access identity} conditions (subjects) and access rights.

In the capability-based systems, policies are grouped by subjects. The \textit{by}_{-}\textit{condition} function returns all policy statements associated with the given condition \( \hat{c} \).
If the condition constant \( \tilde{c} \) specifies particular access identity (subject), then the returned set of policies conceptually represents a capability possessed by the subject identified by the condition constant \( \tilde{c} \).

A capability is associated with each subject, so the subject is implicit and can be omitted from the policy element. Thus, each policy statement contains only elements, which represent objects and access rights.

In the comimpact model, policies are grouped by subject to subject agreement. One of the subjects is an owner of the target objects and the other one is a party interested in the objects. A comimpact specifies sets of objects accessible by the subjects participating in the agreement along with the granted access rights.

Thus, to represent a comimpact we have to choose policy statements from the set \( P \) such that (i) all object constants are owned by either \( c' \) or \( c'' \); and (ii) the condition constants in the policy statements are either \( c' \) or \( c'' \).

We assume that a set of access rights associated with each object has a special right \( \bar{r} \) that means “own”. This right is possessed by the owner of an object and permits the owner to grant and revoke access rights for the object. The set \( P' \) contains policy statements where all objects are owned by the subject \( c' \):

\[
by\_condition(by\_right(P, q), q) = P', \quad q = < o, \bar{r}, c' >, \quad q \in Q.
\]

The set \( O' \) contains objects owned by the subject \( c' \)

\[
O' = \{ o \in O | (\exists p \in P', o \in p) \}.
\]

The set \( P'' \) contains policy statements where all objects are owned by the subject \( c'' \).

\[
by\_condition(by\_right(P, q), q) = P'', \quad q = < o, \bar{r}, c'' >, \quad q \in Q.
\]
The set $O''$ contains objects owned by the subject $c''$

$$O'' = \{ o \in O \mid (\exists p \in P''), \ o \in p \} .$$

The set $\tilde{O}$ contains objects owned by the subjects $c'$ and $c''$.

$$\tilde{O} = O' \cup O'' .$$

A compact is represented by the set $\hat{P}$:

$$\hat{P} = \{ < o, r, c > \mid o \in \tilde{O} \text{ and } r \neq \tilde{r} \text{ and } (c = c' \text{ or } c = c'') \} . \quad (4.17)$$

### 4.2 Extended Model

In this section we describe further refinements of our basic concepts.

A policy statement may specify several conditions of different types. For example: “Tom can read file A only between 9am and 6pm”. This policy defines two conditions: **access identity** and **time**.

In the Basic Model we have considered only one condition in the policy statement. All existing conditions were aggregated into one set $C$. Now we extend the notion of a condition to be distinguished not only by an identifier but also by a type. Each condition element has just one type. We assume that at each instant $K$ condition types exist. We represent these different condition types by $K$ disjoint sets:

$$C = \bigcup_{k=1}^{K} C^k, \bigcap_{i,j=1,K, i\neq j} C^i \cap C^j = \emptyset . \quad (4.18)$$

We next define $K$ condition evaluation functions for each condition type:

$$eval_{cond_i}(c, q) = b, \ c \in C^i, \ eval_{cond_i}(c^*, q) = T, \ i = 1, K, \ q \in Q, \ b \in B . \quad (4.19)$$
In our policy example we define two functions: one to check the access identity and the other one to check the time. In the previous section we only allowed one condition in the representation of elements of the set $P$. We now consider the use of disjunctive and conjunctive information in authorization policies.

The disjunction of objects and access rights in a policy can be model by separate policy statements. For example, policies such as "Tom can read either file $A$ or $B$" and "Tom can either read or write file $A$" can be represented as follows:

\[ p_1 = \langle A, \text{read}, \text{Tom} \rangle, p_2 = \langle B, \text{read}, \text{Tom} \rangle, p_3 = \langle A, \text{write}, \text{Tom} \rangle. \]

However, to represent policies such as "Tom or Joe can read file $A$" and "Tom can read file $A$ on Mondays only if he connects from domain $D$", we need to extend our representation of a policy element.

**Condition block** represents conjunction and disjunction of conditions in a policy statement. A condition block consists of one or more totally ordered condition sets included in either round or square brackets. The round brackets represent the disjunction of conditions $(c_1, c_2, c_3)$, the square brackets denote the conjunction of conditions $[c_1, c_2, c_3]$. If a condition block contains just one element we omit the brackets.

A condition block $k$ is defined over the elements of the set $C$, $k = \mathbb{R}(C)$.\(^2\)

The condition block is ordered. This property is important to deal with possible side effects caused by the condition evaluation.

Condition blocks can be defined with nested disjunctive and conjunctive condition sets. For example, a policy: "Anybody connecting from domain $D$ can read the file $A$. Tom, Joe or Ken can read file $A$ only if an audit record is generated" is represented by a condition block: $(D, [(Tom, Joe, Ken), \text{audit}])$. As we discussed in Chapter 3, a policy may contain four sets of conditions: pre-, request-result-, mid- and post-conditions. To represent these

\(^2\)We prefer this representation to the traditional logical formula representation because the formula does not mandate an evaluation order.
sets, we introduce four condition blocks to our policy statement representation. We redefine the structure of the elements of the set \( P \) given in (4.5) in the following way:

\[
p = < o, r, k_{\text{pre}}, k_{\text{rr}}, k_{\text{mid}}, k_{\text{post}} >, \quad o \in O, \quad r \in R, \quad k_{\text{pre}}, k_{\text{rr}}, k_{\text{mid}}, k_{\text{post}} = \mathcal{R}(C) . \quad (4.20)
\]

Recall, that condition evaluation function requires the information from the context is passed with an authorization request. Now a policy statement can have several conditions of different types. Therefore, we need to extend the representation of the request to include an ordered set of condition constants to support evaluation of several conditions.

We redefine the structure of the elements of the set \( Q \) given in (4.7) in the following way:

\[
q = < o, r, < c_1, c_2, ... , c_n >>, \quad o \in O, \quad r \in R, \quad c_i \in C, \quad i = 1, n . \quad (4.21)
\]

The \texttt{eval\_cond\_block} is a two-place function that takes a condition block \( k \) and an authorization request \( q \) as arguments; applies the \texttt{eval\_cond}_i to the corresponding condition constants from the condition block \( k \); combines the results (see Algorithm 1) and returns a truth value \( b \).

\[
\texttt{eval\_cond\_block}(k, q) = b, \quad k = \mathcal{R}(C), \quad q \in Q, \quad b \in B . \quad (4.22)
\]

A condition block defines the order of condition evaluation unambiguously. The evaluation of a condition block works from the right to the left: condition constants that appear earlier in the block are evaluated first.

As an example of the calculation of \texttt{eval\_cond\_block}, lets \( k \) be the condition block \( ([c_1, c_2], [c_3, c_4, c_5, c_6]) \); let the condition types be such that: \( c_1, c_2 \in C^i \), \( c_3, c_5 \in C^n \) and \( c_4, c_6 \in C^m \); and let the condition evaluation values be such that:

\[
\texttt{eval\_cond}_i(c_1, q) = T
\]

\[
\texttt{eval\_cond}_i(c_2, q) = F
\]
\[ eval_{\text{cond}}(c_3, q) = T \]
\[ eval_{\text{cond}}(c_4, q) = T \]
\[ eval_{\text{cond}}(c_5, q) = F \]
\[ eval_{\text{cond}}(c_6, q) = T \]

We want to compute \( eval_{\text{cond \ block}}([c_1, c_2], ([c_3, c_4, c_5], c_6), q) \).

Figure 4.1 shows the tree that displays the evaluation order. We assign value \( eval_{\text{cond}}(c_i) \) to each leaf \( c_i \) of the tree. Working from the bottom up, we can assign to each vertex of the tree value \( eval_{\text{cond \ block}}(k', q) \), where \( k' \) is a sub-block of a condition block \( k \). At the first step we compute \([c_1, c_2] = F\). Next we compute \([c_3, c_4, c_5] = F\), and so forth. Finally, at the top of the tree we arrive at \( eval_{\text{cond \ block}}([c_1, c_2], ([c_3, c_4, c_5], c_6), q) = T \).

We replace the function \( eval_{\text{policy}} \) given in (4.14) with four functions that apply the function \( eval_{\text{cond \ block}} \) to appropriate condition block from a policy statement \( p \):

\[ eval_{\text{pre \ policy}}(p, q) = eval_{\text{cond \ block}}(k_{\text{pre}}, q) = b, \quad (4.23) \]

\[ k_{\text{pre}} = R(C), \; k_{\text{pre}} \in p, \; p \in P, \; q \in Q, \; b \in B. \]

Figure 4.1: Condition evaluation tree.
\[
eval_{rr\text{-}policy}(p, q) = \eval_{\text{Cond\-}block}(k_{rr}, q) = b, \quad (4.24)
\]
\[
k_{rr} = \mathcal{R}(C), \; k_{rr} \in p, \; p \in P, \; q \in Q, \; b \in B.
\]
\[
eval_{\text{mid}\text{-}policy}(p, q) = \eval_{\text{Cond}\text{-}block}(k_{\text{mid}}, q) = b, \quad (4.25)
\]
\[
k_{\text{mid}} = \mathcal{R}(C), \; k_{\text{mid}} \in p, \; p \in P, \; q \in Q, \; b \in B.
\]
\[
eval_{\text{post}\text{-}policy}(p, q) = \eval_{\text{Cond}\text{-}block}(k_{\text{post}}, q) = b, \quad (4.26)
\]
\[
k_{\text{post}} = \mathcal{R}(C), \; k_{\text{post}} \in p, \; p \in P, \; q \in Q, \; b \in B.
\]

Similarly, we replace the \texttt{eval\_policy\_set} function given in (4.15) with four functions that apply appropriate function to each policy statement and take a conjunction of the returned truth values:

\[
eval_{\text{pre\-}policy\_set}(P, q) = \eval_{\text{pre\-}policy}(p_1, q) \land \\
\land \eval_{\text{pre\-}policy}(p_2, q) \land \ldots \land \eval_{\text{pre\-}policy}(p_n, q) = b, \; q \in Q, \; b \in B. \quad (4.27)
\]
\[
eval_{rr\text{-}policy\_set}(P, q) = \eval_{rr\text{-}policy}(p_1, q) \land \\
\land \eval_{rr\text{-}policy}(p_2, q) \land \ldots \land \eval_{rr\text{-}policy}(p_n, q) = b, \; q \in Q, \; b \in B. \quad (4.28)
\]

62
\[
\text{\texttt{eval\_mid\_policy\_set}}(P, q) = \text{\texttt{eval\_mid\_policy}}(p_1, q) \land \\
\text{\texttt{eval\_mid\_policy}}(p_2, q) \land \ldots \land \text{\texttt{eval\_mid\_policy}}(p_n, q) = b, \ q \in Q, \ b \in B. \quad (4.29)
\]

\[
\text{\texttt{eval\_post\_policy\_set}}(P, q) = \text{\texttt{eval\_post\_policy}}(p_1, q) \land \\
\text{\texttt{eval\_post\_policy}}(p_2, q) \land \ldots \land \text{\texttt{eval\_post\_policy}}(p_n, q) = b, \ q \in Q, \ b \in B. \quad (4.30)
\]

We redefine the \textit{authorization} function as follows:

\[
\text{\texttt{authorization}}(P, q) = \text{\texttt{eval\_pre\_policy\_set}}(\text{\texttt{get\_policy}}(P, q), q) \land \\
\text{\texttt{eval\_rr\_policy\_set}}(\text{\texttt{get\_policy}}(P, q), q) = b, \ q \in Q, \ b \in B. \quad (4.31)
\]

The \textit{execution\_control} is a two-place composite function:

\[
\text{\texttt{execution\_control}}(P, q) = \text{\texttt{eval\_mid\_policy\_set}}(\text{\texttt{get\_policy}}(P, q), q) = b, \ q \in Q, \ b \in B. \quad (4.32)
\]

The \textit{post\_execution\_actions} is a two-place composite function:

\[
\text{\texttt{post\_execution\_actions}}(P, q) = \text{\texttt{eval\_post\_policy\_set}}(\text{\texttt{get\_policy}}(P, q), q) = b, \ q \in Q, \ b \in B. \quad (4.33)
\]

Note that in the implementation, the relevant policies should be retrieved just once and stored for further evaluation by the listed functions.
4.2.1 Algorithm 1

Next we present the algorithm for the \textit{eval\_cond\_block} function.

\textbf{Input:}

- $k$ is the condition block from the relevant policy statement.
- $q$ is the authorization request.

\textbf{Output:}

- $T/F/U$ (true, false or undefined).

\textbf{Variables:}

- $n$ is a structure that represents a node of a tree.
- $n.o = AND/OR/U$ specifies operation defined in the node. If the node represents a leaf, the operation is $U$ (undefined).
- $n.m$ contains the number of children of the node $n$. If $n.m = 0$, the node represents a leaf.
- $n.children[m]$ is an array of pointers to the children of the node. All children are enumerated from left to right.
- $n.value = AND/OR/U$ contains the value of the node.

\textbf{t} is a tree structure.

\textbf{root} is the root of a tree structure.

\textbf{i} is an index variable.

\textbf{Functions:}

- \textit{build\_tree}(k) takes a condition block in the format described in Section 4.2 and returns a tree structure.
- \textit{get\_root}(t) takes a tree as input and returns a root node.
- \textit{eval\_cond}(n, q) is a function that evaluates one condition. The function returns $T/F/U$.
- \textit{evaluate}(n, q) is a function that evaluates the nodes of the tree recursively. The function returns $T/F/U$. 

64
Algorithm 1.a: node evaluation function $evaluate(n,q)$:

begin
if ($n.m = 0$) then return $eval\_cond(n.value, q)$ /* this is a leaf */
else
for all $i$: $1 \leq i \leq n.m$
n.value = $evaluate(n.children[i], q)$
if ($n.value = U$) then return $U$ /* We stop the condition evaluation as soon as we either
encountered $U$ or the outcome is known */
else
if ($(n.o = \wedge)$ and ($n.value = F$)) then return $F$ endif
if ($(n.o = \lor)$ and ($n.value = T$)) then return $T$ endif
endif
endfor

if ($n.o = \wedge$) then return $T$
else return $F$
endif
endif
end

---

*We do not continue evaluation just to be on the safe side. $U$ can be returned when we could not evaluate condition. If the unevaluated condition is a write condition, we do not know how it affects the system state and, therefore, the evaluation of the remaining conditions.*
Algorithm 1: condition evaluation function $eval\_cond\_block(k, q)$:

begin
\hspace{1em} t = build\_tree(k)
\hspace{1em} root = get\_root(t)
\hspace{1em} return evaluate(root, q)
end

4.2.2 Summary

We have defined the Basic Model and the Extended Model. Sets $O$, $B$ and $R$ will be the same for all five models. The sets $Q$ and $P$ are the same for the Extended Model as for the Basic Model, however, the structure of the members of these sets is changed according to (4.20) and (4.21). The set $C$ is redefined as given in (4.18). The $eval\_cond$ function has been redefined as given in (4.19). New function $eval\_cond\_block$ is added.

The functions $by\_object$, $by\_right$ and $get\_policy$ are redefined as:

\[
by\_object(P, q) = \{< \hat{o}, r, k_{pre}, k_{rr}, k_{mid}, k_{post} > \in P | \hat{o} \in q \}, \ q \in Q \ . \quad (4.34)
\]

\[
by\_right(P, q) = \{< o, \hat{r}, k_{pre}, k_{rr}, k_{mid}, k_{post} > \in P | \hat{r} \in q \}, \ q \in Q \ . \quad (4.35)
\]

\[
get\_policy(P, q) =
\]

\[
= by\_right(by\_object(P, q), \ q) = \{< \hat{o}, \hat{r}, k_{pre}, k_{rr}, k_{mid}, k_{post} > \in P | \hat{o}, \hat{r} \in q \}, \ q \in Q \ . \quad (4.36)
\]

The function $eval\_policy$ is replaced with four functions $eval\_pre\_policy$, $eval\_rr\_policy$, $eval\_mid\_policy$ and $eval\_post\_policy$.

The function $eval\_policy\_set$ is replaced with four functions $eval\_pre\_policy\_set$, $eval\_rr\_policy\_set$, $eval\_mid\_policy\_set$ and $eval\_post\_policy\_set$.  

66
The authorization function has been redefined as given in (4.31). New functions \texttt{execution.control} and \texttt{post-execution-actions} are added.

We ignore the \texttt{by-condition} function for the rest of this dissertation.

### 4.3 Model with Negative Rights

It is possible to give users negative authorizations on an object, thus specifically denying access rights to users or groups. This is useful when a policy officer wishes to restrict a given member of a group's access while not affecting the whole group. For instance, if the officer gives a permission to read a file to Group1, but denies it to Tom, who is a member of the group, Tom will not be able to read the file. Andrew File System [40] and WindowsNT [70] are examples of systems that use negative access rights.

In this section we consider the effect of allowing negative authorizations.

In our model, negation is applied only to the elements of the set \( R \) to model negative rights. We do not define negative conditions. To model negative rights we introduce set \( \overline{R} \). \( \overline{R} \) is a finite dynamic nonempty set of negative access rights. Set \( \overline{R} \) is constructed from the set \( R \) by applying negation to each element of the set \( R \).

\[
\overline{R} = \{ \neg r_1, \neg r_2, \ldots, \neg r_n \} . \tag{4.37}
\]

Note that \( R \cap \overline{R} = \emptyset \).

We next extend the representation of a policy element \( p \) given in (4.20) to contain positive or negative access right:

\[
p = < o, r, k_{pre}, k_{rr}, k_{mid}, k_{post} >, \quad o \in O, \quad r \in R \cup \overline{R}, \quad k_{pre}, k_{rr}, k_{mid}, k_{post} = \mathbb{R}(C) . \tag{4.38}
\]
If a policy element contains a positive access right we call it a positive policy statement (or positive authorization). Similarly, if a policy statement contains a negative access right we call it a negative policy statement (or negative authorization).

We modify the definition of the $by\_right$ function given in (4.35) to return positive and negative policy statements with the matching access right:

$$by\_right(P, q) =$$

$$= \{ p \in P \mid (p = < o, \bar{r}, k_{pre}, k_{rr}, k_{mid}, k_{post}> \text{ or } p = < o, \neg \bar{r}, k_{pre}, k_{rr}, k_{mid}, k_{post}> \text{ and }$$

$$\text{and } \bar{r} \in q \}, q \in Q \text{.} \quad (4.39)$$

Next we redefine the policy evaluation functions given in (4.23),(4.24), (4.25) and (4.26) to modify the value returned by the $eval\_cond\_block$ to capture positive and negative authorization semantics. Each of the functions returns $F$ if the access right contained in the policy statement is negative and the value returned by the $eval\_cond\_block$ function is $T$, otherwise the function returns the value of the $eval\_cond\_block$ function. This can be expressed by the following equations:

$$eval\_pre\_policy(p, q) = \begin{cases} 
  eval\_cond\_block(k_{pre}, q), & r \in R \\
  F, & r \in \mathcal{R} \text{ and } eval\_cond\_block(k_{pre}, q) = T
\end{cases} \quad (4.40)$$

$$k_{pre} = \mathcal{R}(C), \ k_{pre}, r \in p, p \in P, q \in Q \text{.}$$

$$eval\_rr\_policy(p, q) = \begin{cases} 
  eval\_cond\_block(k_{rr}, q), & r \in R \\
  F, & r \in \mathcal{R} \text{ and } eval\_cond\_block(k_{rr}, q) = T
\end{cases} \quad (4.41)$$

$$k_{rr} = \mathcal{R}(C), \ k_{rr}, r \in p, p \in P, q \in Q \text{.}$$
\[ \text{eval}_{\text{mid\_policy}}(p, q) = \begin{cases} \text{eval}_{\text{cond\_block}}(k_{\text{mid}}, q), & r \in R \\ F, & r \in \overline{R} \text{ and } \text{eval}_{\text{cond\_block}}(k_{\text{mid}}, q) = T \end{cases} \] (4.42)

\[ k_{\text{mid}} = \mathcal{R}(C), \quad k_{\text{mid}}, r \in p, \quad p \in P, \quad q \in Q. \]

\[ \text{eval}_{\text{post\_policy}}(p, q) = \begin{cases} \text{eval}_{\text{cond\_block}}(k_{\text{post}}, q), & r \in R \\ F, & r \in \overline{R} \text{ and } \text{eval}_{\text{cond\_block}}(k_{\text{post}}, q) = T \end{cases} \] (4.43)

\[ k_{\text{post}} = \mathcal{R}(C), \quad k_{\text{post}}, r \in p, \quad p \in P, \quad q \in Q. \]

Note that we do not just apply negation \( \neg \) to the value returned by the \text{eval}_{\text{cond\_block}} function. Doing so contradicts the “closed world” approach used in our model: by default all access is denied. Only explicit positive authorizations can grant access. This is because if we have a negative policy element that evaluates to \( F \) the \text{eval}_{\text{pre\_policy}} function returns \( T \) and the authorization could be granted.

In our model, we accept the simultaneous presence of authorizations that differ only by the sign. For example, there may exist a positive authorization \( p = < o, r, k > \) and a negative authorization \( p = < o, -r, k > \). Since we take the conjunction of the authorizations as defined in (4.27), the conflict is resolved in favor of the negative authorization. Therefore, the access will be denied.

### 4.3.1 Summary

We have introduced the Model with Negative Rights.

The sets \( C, R \) and \( Q \) are exactly the same as for the Extended Model. The set \( P \) remains the same for the Model with Negative Rights, however the structure of the members of is changed according to (4.38). The new set \( \overline{P} \) is introduced.
The \texttt{by\_right} function is redefined as given in (4.39). The \texttt{eval\_pre\_policy}, \texttt{eval\_rr\_policy}, \texttt{eval\_mid\_policy}
and \texttt{eval\_post\_policy}, functions are redefined as given in (4.40), (4.41),(4.42) and (4.43).

The functions \texttt{eval\_cond}, \texttt{eval\_cond\_block}, \texttt{by\_object}, \texttt{get\_policy}, \texttt{eval\_pre\_set\_policy},
\texttt{eval\_rr\_set\_policy}, \texttt{eval\_mid\_set\_policy}, \texttt{eval\_post\_set\_policy}, \texttt{authorization},
\texttt{execution\_control}, \texttt{post\_execution\_actions} are unchanged.

### 4.4 Model with Priorities

Recall, from discussion in Section 3.3 that objects and policies are organized into security domains, each of which defines a distinct scope with a common set of security policies.

A domain may be divided into sub-domains to form a security domain hierarchy. For security domains to interact with each other, their inter-domain security policies need to be defined and enforced.

In our model, security domains are organized into peer-peer and supervisor-subordinate relationships to form a hierarchy of levels. A supervisor domain belongs to a higher hierarchical level than that of its subordinate domain. Security domains that belong to the same hierarchical level are peers. To enforce a hierarchical relationship we use the pre-determined hierarchical levels of security domains for assigning priorities to each domain’s policies.

An example of a three-level hierarchy with assigned priorities is shown in Figure 4.2. At the top of the hierarchical level are domains that represent countries, which are assigned the highest priority $w_1$. The universities belong to the second hierarchical level and are assigned priority $w_2$. At the lowest hierarchical level are departments that are assigned the lowest priority $w_3$.  

70
The structure of the policy domains that contribute the policies is not specified explicitly in our framework. Only the hierarchical relationship (priority of the policy) among the domains is taken into consideration.

Recall, that the composition of policies that belong to security domains with equal priorities is called the Horizontal Policy Composition, which is represented in our model as conjunction of policy statements as defined in (4.27), (4.28), (4.29) and 4.30). Note that the Horizontal Policy Composition represents the union of policies from different domains.

The composition of policies that belong to security domains with different priorities is called the Hierarchical Policy Composition. In our model, the Vertical Policy Composition is modeled by assigning priorities and policy composition modes. The policy priority determines which authorization should take precedence when conflicting authorizations (e.g., identical rights and objects but different condition blocks) exist. Recall that in our
model the policy evaluation is sequential, therefore the higher policy priority the earlier it will be evaluated.

To express policy priorities, we define set $W$. $W$ is a totally ordered set of elements that can be compared (e.g., integers).

$$W = \{w \mid w_i < w_j, \, i < j, \, i, j = 1, \ldots, n\}. \quad (4.44)$$

When policy domains are arranged hierarchically, the composition modes may include:

1. **expand**
   A policy statement can be expanded with the policy statement of a lower priority. A policy administrator may specify policies that can be broaden by the policies of the subordinate administrators. This may be useful for setting up the restrictive default policies. Specifying permissive policy with the expand mode is impractical. Thus, the resulting policy represents the disjunction of the policies.

2. **narrow**
   A policy statement can be narrowed with a policy statement of a lower priority. The policy that controls access to an object may have both mandatory and discretionary components. Generally, mandatory policy is set by domain administrator, while discretionary policy is set by individuals. The mandatory policies must always hold. The discretionary policies of individual users must be satisfied in addition to the mandatory policies. Thus, the resulting policy represents the conjunction of the mandatory and discretionary policies.

3. **stop**
   Discard the policies of the lower priority altogether. A policy officer may require complete overriding of the lower-level policies with the high-level policies. This is useful, for example, when a policy officer wants to, for example, allow access to a
file only to himself. If he specifies it using the \textit{expand} mode, than additional access can be granted by the lower-level policies. If he uses \textit{narrow} mode, the lower-level policies could add additional restriction that can result in deny of the access. Thus, the resulting policy represents the conjunction of the mandatory and discretionary policies.

We next introduce the set $M$ consisting of three elements that represent the composition modes. In particular, a policy statement can be expanded ($\lor$), narrowed ($\land$) or stopped ($\Box$).

$$M = \{\lor, \land, \Box\}. \quad (4.45)$$

We next extend the element $p$ defined in (4.38) to include additional components $w$ and $m$. The component $w$ represents hierarchical priority of this policy element. The component $m$ denotes the hierarchical composition mode of the policy element.

$$p = \langle o, w, m, r, k_{pre}, k_{err}, k_{mid}, k_{post} \rangle,$$

$$o \in O, \ r \in R \cup R_i, \ w \in W, \ m \in M, \ k_{pre}, k_{err}, k_{mid}, k_{post} = \Re(C). \quad (4.46)$$

The policy set evaluation functions are now expressed as follows:

$$\text{eval\_pre\_policy\_set}(P, q) = \text{eval\_pre\_policy}(p_1, q).\Diamond_1$$

$$\Diamond_1 \text{eval\_pre\_policy}(p_2, q) \Diamond_2 \ldots \Diamond_n \text{eval\_pre\_policy}(p_n, q) = b, \ q \in Q, \ b \in B. \quad (4.47)$$
\[ \text{eval}_{rr\text{-policy-set}}(P, q) = \text{eval}_{rr\text{-policy}}(p_1, q) \diamond_1 \]

\[ \diamond_1 \text{eval}_{rr\text{-policy}}(p_2, q) \diamond_2 \ldots \diamond_{s-1} \text{eval}_{rr\text{-policy}}(p_s, q) = b, q \in Q, b \in B. \tag{4.48} \]

\[ \text{eval}_{mid\text{-policy-set}}(P, q) = \text{eval}_{mid\text{-policy}}(p_1, q) \diamond_1 \]

\[ \diamond_1 \text{eval}_{mid\text{-policy}}(p_2, q) \diamond_2 \ldots \diamond_{s-1} \text{eval}_{mid\text{-policy}}(p_s, q) = b, q \in Q, b \in B. \tag{4.49} \]

\[ \text{eval}_{post\text{-policy-set}}(P, q) = \text{eval}_{post\text{-policy}}(p_1, q) \diamond_1 \]

\[ \diamond_1 \text{eval}_{post\text{-policy}}(p_2, q) \diamond_2 \ldots \diamond_{s-1} \text{eval}_{post\text{-policy}}(p_s, q) = b, q \in Q, b \in B. \tag{4.50} \]

where the policy statements are ordered by priorities:

\[ \forall p_j \forall p_k (w_j \geq w_k, w_j \in p_j, w_k \in p_k, w_j, w_k \in W), j, k = 1, m, j < k. \]

Symbol \( \diamond_i \) represents either disjunction \( \vee \) or conjunction \( \wedge \) of the results returned by the \( \text{eval}_{\{pre/rr/mid/post\}\text{-policy}} \) functions applied to the policy elements \( p_i \) (left element) and \( p_{i+1} \) (right element) and authorization request \( q \). The representation of the \( \text{eval}_{\{pre/rr/mid/post\}\text{-policy}} \) functions has two properties:

- **Property 1**

  The symbol \( \diamond \) represents disjunction \( \vee \) if the composition mode of the left element is equal to \( \vee \) and the priority of the left element is greater than the priority of the right element. This rule along with the Property 2 described below enforce the Hierarchical Policy Composition.
The symbol $\Diamond$ represents conjunction $\land$ if the priorities of the left element and right elements are equal. Thus, the composition modes of the policies with equal priorities are ignored. This rule enforces the Horizontal Policy Composition.

$$\Diamond_i = \begin{cases} \forall, & \text{if } m_i = \forall \text{ and } w_i > w_{i+1} , \\ \land, & \text{if } w_i = w_{i+1} , \end{cases} \quad (4.51)$$

$$i = 1, s, m_i, w_i \in p_i, w_{i+1} \in p_{i-1}, m_i \in M, p_i, p_{i-1} \in P.$$  

- **Property 2**

The maximum number of the evaluated policy elements $s$ is equal to:

(i) $h$, If there exists a policy element $p_h$ such that the composition mode $m_h$ of this element is equal to $\square$ and the priority of this element is higher than the priority of the next element $p_{h+1}$. If there are several policy statements with the composition mode $\square$, then we consider the first that appear in the ordered list. Simply put, the evaluation of the policies stops when we reach first policy element with the composition mode $\square$ and the priority of the element is higher than the priority of the next policy element.

(ii) The cardinality of the set $P$, otherwise. The actual number of evaluated policy statements can be known only at the evaluation time.

$$s = \begin{cases} n, & \text{if } \exists p_h : (m_h = \square \text{ and } w_h > w_{h+1}) , \\ \min(\forall h : p_h : (m_h = \square \text{ and } w_j > w_{h+1})) , \end{cases} \quad (4.52)$$

$$h = 1, n, n = \text{card}(P), w_h, m_h \in p_j, w_{h+1} \in p_{h+1}, p_h \in P.$$
For our semantics, the policy composition reduces to:

1. assessing the policy evaluation order based on the priority of the current policy element.

2. computation of the next composition mode $\Diamond_i$ based on the priority and the composition mode of the current policy element.

As an example of the calculation of $eval_{\text{pre-policy}} \text{set}$, let the authorization request $q$ be such that $q = < o, r, < c1, c2, c3, c4 > >$; lets the policy set $P$ be such that:

\[
\begin{align*}
    p_{\text{USA}} &= < o, w_1, \lor, r, k_{pre1} > \\
    p_{\text{RUSSIA}} &= < o, w_1, \lor, r, k_{pre2} > \\
    p_{\text{USP}} &= < o, w_2, \Box, r, k_{pre3} > \\
    p_{\text{MGU}} &= < o, w_2, \Box, r, k_{pre4} > \\
    p_{\text{CS}} &= < o, w_3, \land, r, k_{pre5} > \\
    p_{\text{EE}} &= < o, w_3, \land, r, k_{pre6} >
\end{align*}
\]

and let the policy evaluation values be such that:

\[
\begin{align*}
    eval_{\text{pre-policy}}(p_{\text{USA}}, q) &= T \\
    eval_{\text{pre-policy}}(p_{\text{RUSSIA}}, q) &= T \\
    eval_{\text{pre-policy}}(p_{\text{USP}}, q) &= F \\
    eval_{\text{pre-policy}}(p_{\text{MGU}}, q) &= T
\end{align*}
\]

The $eval_{\text{pre-policy}} \text{set}$ function can be expressed as follows:

\[
\begin{align*}
    eval_{\text{pre-policy}} \text{set}(P, q) &= eval_{\text{pre-policy}}(p_{\text{RUSSIA}}, q) \land eval_{\text{pre-policy}}(p_{\text{USA}}, q) \lor \nonumber \\
    & \quad \lor eval_{\text{pre-policy}}(p_{\text{USP}}, q) \land eval_{\text{pre-policy}}(p_{\text{MGU}}, q) = T.
\end{align*}
\]

Note that the evaluation order of the policy statements with equal priorities is arbitrary.
4.4.1 Summary

We have defined the Model with Priorities. The sets $C$, $R$, $\overline{R}$ and $Q$ are the same as defined for Model with Negative Rights.

The set $P$ is the same for Model with Priorities, however the structure of its members is changed according to (4.46). The new sets $W$ and $M$ are introduced. We redefined the functions $\text{eval.pre-policy.set}$, $\text{eval.rr-policy.set}$, $\text{eval.mid-policy.set}$ and $\text{eval.post-policy.set}$ as given in (4.47), (4.48), (4.49) and (4.50).

The functions $\text{by.object}$, $\text{by.right}$, $\text{eval.ond}$, $\text{eval.ond.block}$, $\text{get.policy}$, $\text{eval.pre.policy}$, $\text{eval.rr.policy}$, $\text{eval.mid.policy}$, $\text{eval.post.policy}$, $\text{authorization}$, $\text{execution.control}$, $\text{post.execution.actions}$ are unchanged.

4.5 EACL Model

In practice, policies associated with an object are combined into a unified entity to enable the consistent policy specification and their centralized management. For example, to avoid having a separate policy for each access right, an aggregated policy structure for an object can specify positive and negative authorizations with different access rights defined for the object. To maintain policies in this way, we introduce the notion of an $\text{Extended Access Control List (EACL)}$.

An $\text{EACL}$ for object $\hat{o}$ is an ordered tuple that consists of priority element $w$ and disjunction of policy elements $p_i$ called $\text{EACL entries (EACE)}$.

$$ e = < w, p_1 \lor p_2 \lor \ldots \lor p_n >, \ w \in W, \ p_i \in P, \ i = 1, n \ . \quad (4.53) $$

An EACL representation has two properties:

1) The priority is specified at two levels: at EACL level and at the EACE level.
The priority of an EAACL supports the Vertical Policy Composition and is mandatory assigned according to the policy domain hierarchy.

The priority of EACEs specifies the evaluation order within the EAACL. The assigning of the EACEs priorities is at the discretion of the creator the EAACL.

2) EACEs are ordered by the priority. This means that \( \forall p_i \) and \( \forall p_j \), included in the disjunction \( p_1 \lor p_2 \lor ... \lor p_n \) the following is true:

\[
w_i > w_j, \; w_i \in p_i, w_j \in p_j, \; i < j, \; i, j = 1, n.
\]

We use this property for representation convenience. In general, the EACEs comprising the EAACL can be listed in arbitrary order, however, the evaluation takes place according to the priorities of EACEs.

EACEs that disallow operations are generally placed before EACEs that allow operations in the EAACL.

Given an EAACL \( e \) and object \( o \), we use the notation \( e \leftrightarrow o \) to denote that \( e \) is associated with \( o \), in other words object \( o \) is protected by an EAACL \( e \).

There maybe several EAACLs associated with each object.

We next introduce a finite dynamic nonempty set of EAACLs \( E \):

\[
E = \{ e_1, e_2, \ldots, e_n \}.
\]

We redefine the \( \text{by-object} \) function to take the set \( E \) and request \( q \) (\( q \) contains target object \( \hat{o} \)) as arguments and return a subset of the set \( E \) containing EAACLs associated with the target object \( \hat{o} \).

\[
\text{by-object}(E, q) = \{ e \in E \mid e \leftrightarrow \hat{o} \text{ and } \hat{o} \in q \}, \; q \in Q.
\]
We can omit object from the element $p$ definition given in (4.46):

$$
p = < w, m, r, k_{pre}, k_{rr}, k_{mid}, k_{post} > ,
$$

$$
w \in W, \ m \in M, \ r \in R \bigcup \overline{R}, \ k_{pre}, k_{rr}, k_{mid}, k_{post} = \mathcal{R}(C) . \quad (4.56)
$$

We next introduce the \textit{prune} function that takes an EACL $e$ and a positive access right $r$ as arguments and returns a new EACL $e'$ that represents the reduced version of the original EACL $e$.

The function only saves the EACEs with the matching access right (either positive or negative), all other policy statements are omitted from the resulting disjunction of the entries. Note that the ordering of the elements in the reduced subset is not changed.

$$
\text{prune}(e, \hat{r}) = < \hat{w}, \ p_1 \lor p_2 \lor \ldots \lor p_m > ,
$$

$$
\forall p_i (p_i = < w, m, \hat{r}, k_{pre}, k_{rr}, k_{mid}, k_{post} > \text{ or } p_i = < w, m, \neg \hat{r}, k_{pre}, k_{rr}, k_{mid}, k_{post} > ) ,
$$

$$
i = \overline{1, m}, \ \hat{w} \in e, \ e \in E, \ \hat{r} \in R . \quad (4.57)
$$

We redefine the \textit{by} \textit{right} function given in (4.39) to return the set of truncated EACLs:

$$
\text{by} \textit{right}(E, q) = \{ e' \mid (\forall e \in E), \ e' = \text{prune}(e, \hat{r}) \text{ and } \hat{r} \in q \}, \ q \in Q . \quad (4.58)
$$
The functions `eval_pre_policy`, `eval_r_policy`, `eval_mid_policy` and `eval_post_policy` are redefined as follows:

\[ \text{eval}_\text{pre-policy}(e, q) = \text{eval} \mathcal{L} B(k_{\text{pre}_1}, q) \lor \text{eval} \mathcal{L} B(k_{\text{pre}_2}, q) \lor \text{eval} \mathcal{L} B(k_{\text{pre}_n}, q) , \]

\[
\text{eval} \mathcal{L} B = \begin{cases} 
\text{eval}_\text{cond block}(k_{\text{pre}_i}, q), & r \in R \\
F, & r \in \overline{R} \text{ and eval}_\text{cond block}(k_{\text{pre}_i}, q) = T 
\end{cases} \tag{4.59}
\]

\[ k_{\text{pre}_i} = \mathcal{R}(C), k_{\text{pre}_i}, r \in p_i, p_i \in e, i = 1, n \in E, q \in Q . \]

\[ \text{eval}_\text{r-policy}(e, q) = \text{eval} \mathcal{L} B(k_{\text{rr}_1}, q) \lor \text{eval} \mathcal{L} B(k_{\text{rr}_2}, q) \lor \text{eval} \mathcal{L} B(k_{\text{rr}_n}, q) , \]

\[
\text{eval} \mathcal{L} B = \begin{cases} 
\text{eval}_\text{cond block}(k_{\text{rr}_i}, q), & r \in R \\
F, & r \in \overline{R} \text{ and eval}_\text{cond block}(k_{\text{rr}_i}, q) = T 
\end{cases} \tag{4.60}
\]

\[ k_{\text{rr}_i} = \mathcal{R}(C), k_{\text{rr}_i}, r \in p_i, p_i \in e, i = 1, n \in E, q \in Q . \]

\[ \text{eval}_\text{mid-policy}(e, q) = \text{eval} \mathcal{L} B(k_{\text{mid}_1}, q) \lor \text{eval} \mathcal{L} B(k_{\text{mid}_2}, q) \lor \text{eval} \mathcal{L} B(k_{\text{mid}_n}, q) , \]

\[
\text{eval} \mathcal{L} B = \begin{cases} 
\text{eval}_\text{cond block}(k_{\text{mid}_i}, q), & r \in R \\
F, & r \in \overline{R} \text{ and eval}_\text{cond block}(k_{\text{mid}_i}, q) = T 
\end{cases} \tag{4.61}
\]

\[ k_{\text{mid}_i} = \mathcal{R}(C), k_{\text{mid}_i}, r \in p_i, p_i \in e, i = 1, n \in E, q \in Q . \]
\[
\text{eval}_{\text{post-policy}}(e, q) = \text{eval}_{\mathcal{L}B}(k_{\text{post}_1}, q) \lor \text{eval}_{\mathcal{L}B}(k_{\text{post}_2}, q) \lor \text{eval}_{\mathcal{L}B}(k_{\text{post}_n}, q),
\]

\[
\text{eval}_{\mathcal{L}B} = \begin{cases} 
\text{eval}_{\text{cond-block}}(k_{\text{post}_r}, q), & r \in R \\
F, & r \in \mathcal{U} \text{ and } \text{eval}_{\text{cond-block}}(k_{\text{post}_i}, q) = T 
\end{cases}
\tag{4.62}
\]

\[
k_{\text{pre}_i} = \mathfrak{F}(C), k_{\text{post}_i}, r \in p_i, p_i \in e, i = 1, n \in E, q \in Q.
\]

We present an algorithm that describes these functions in the next section. The policy set evaluation functions can be expressed as follows:

\[
\text{eval}_{\text{pre-policy-set}}(E, q) = \text{eval}_{\text{pre-policy}}(e_1, q) \triangledown_1
\]

\[
\triangledown_1 \text{eval}_{\text{pre-policy}}(e_2, q) \triangledown_2 \ldots \triangledown_{n-1} \text{eval}_{\text{pre-policy}}(e_n, q) = b, q \in Q, b \in B. \quad (4.63)
\]

\[
\text{eval}_{\text{rr-policy-set}}(E, q) = \text{eval}_{\text{rr-policy}}(e_1, q) \triangledown_1
\]

\[
\triangledown_1 \text{eval}_{\text{rr-policy}}(e_2, q) \triangledown_2 \ldots \triangledown_{n-1} \text{eval}_{\text{rr-policy}}(e_n, q) = b, q \in Q, b \in B. \quad (4.64)
\]

\[
\text{eval}_{\text{mid-policy-set}}(E, q) = \text{eval}_{\text{mid-policy}}(e_1, q) \triangledown_1
\]

\[
\triangledown_1 \text{eval}_{\text{mid-policy}}(e_2, q) \triangledown_2 \ldots \triangledown_{n-1} \text{eval}_{\text{mid-policy}}(e_n, q) = b, q \in Q, b \in B. \quad (4.65)
\]

\[
\text{eval}_{\text{post-policy-set}}(E, q) = \text{eval}_{\text{post-policy}}(e_1, q); \triangledown_1
\]

\[
\triangledown_1 \text{eval}_{\text{post-policy}}(e_2, q) \triangledown_2 \ldots \triangledown_{n-1} \text{eval}_{\text{post-policy}}(e_n, q) = b, q \in Q, b \in B. \quad (4.66)
\]
The EACLs $e_i$ are ordered by the EACL's priority:

$$
\forall e_j \forall e_k : (w_j \geq w_k, w_j \in e_j, w_k \in e_k, w_j, w_k \in W), j, k = 1, s, j < k .
$$

The properties of these functions given in Section 4.4 are not changed. We present an algorithm that describes these functions in Section 4.5.2.

### 4.5.1 Algorithm 2

We next present the algorithm for the `eval_pre_policy` function. The evaluation algorithms for the functions `eval_rr_policy`, `eval_mid_policy` and `eval_post_policy` are defined in a similar manner.

**Input:**

- $e$ is an EACL.
- $q$ is an authorization request.

**Output:**

- $T/F/U$ (true, false or undefined).
- `next_mode` the composition mode for the next EACL.

**Variables:**

- $e$ represents an EACL as defined in (4.53): 
- $e.num$ contains the number of policy statements in the disjunction.
- $e.p[j]$ represents a policy element as defined in (4.56).
- $e.p[j].m$ contains the composition mode of the policy statement.
- $e.p[j].k_{pre}$ contains the condition block with pre-conditions.
- $neg = T/F$ is a Boolean variable that indicates whether the current policy statement is positive or negative.
- $was\_positive \_U = T/F$ is a Boolean variable that indicates whether there was a positive policy statement that had been evaluated to $U$. 

82
was\_negative\_U = T/F is a Boolean variable that indicates whether there was a negative policy statement that had been evaluated to U.

j is an index variable.

\textit{answer} = T/F/U indicates current evaluation result.

Functions:

negative(e.p[j]) takes a policy element as input and returns T, if the policy element is negative, F, otherwise.

Algorithm 2: policy evaluation function eval\_pre\_policy(e,q,next\_mode):

\begin{verbatim}
begin
  was\_negative\_U = F
  was\_positive\_U = F
  for all j: 1 \leq j \leq e.num
    neg = negative(e.p[j])
    answer = eval\_cond\_block(e.p[j],k\_pre,q) \textsuperscript{4}
    if ((answer = T) and (neg = T)) then answer = F /* modify the answer if the policy statement is negative */
    endif
    if (answer = U) then if (neg = T) then was\_negative\_U = T
      else was\_positive\_U = T
    endif
  endif
  /* Decide whether to stop */
  if (answer = T) then goto label endif
endfor
label:
  if ((answer = T) and (was\_negative\_U = T))
\end{verbatim}

\textsuperscript{4}The algorithm for this function is defined in Section 4.2.1.
\textbf{then} \ \text{answer} = \text{U} \ \textbf{endif}

\textbf{if} (\text{answer} = \text{F}) \ \text{and} \ \text{was\_positive,}_U = \text{T}) \ \textbf{then} \ \text{answer} = \text{U} \ \textbf{endif}

\text{return} \ \text{answer}

\textbf{end}

\subsection{Algorithm 3}

Next we present the algorithm for the \textit{eval\_pre\_policy\_set} function.

The evaluation algorithms for the functions \textit{eval\_rr\_policy\_set}, \textit{eval\_mid\_policy\_set} and \textit{eval\_post\_policy\_set} are defined in a similar manner.

\textbf{Input:}

\textit{E} is a set of relevant EACLs.

\textit{q} is an authorization request.

\textbf{Output:}

\textit{T/F/U}.

\textbf{Variables:}

\textit{EACL\_array} is an array that represents a partially ordered set of EACLs.

\textit{EACL\_array[i].w} specifies the priority of an EACL.

\textit{EACL\_array[i].m} contains the composition mode of the EACL.

\textit{current\_mode} = \text{V} / \text{Ł} / \text{□} is a current policy composition mode.

\textit{next\_mode} = \text{V} / \text{Ł} / \text{□} is a policy composition mode for the next EACL.

\textit{í} is an index variable.

\textit{answer} = \text{T/F/U} indicates current evaluation result.

\textit{previous\_answer} = \text{T/F/U} stores previous evaluation result.

\textbf{Functions:}

The \textit{by\_priority(E)} function takes a set \textit{E} as an argument and returns a partially ordered
set of EACLs $EACL\_array$. The EACLs with the highest priority are placed in the beginning of the array:

$\forall i, j, \text{such that } i < j, 1 \leq i, j \leq n \text{ and } n = card(E)$ the following is true:

$EACL\_array.e[i].w \leq EACL\_array.e[j].w$

The $eval\_pre\_policy\_set$ function stops further evaluation of the array of EACLs if:

1. Current composition mode is $\Box$ and the priority of the current EACL is greater than the priority of the next EACL in the array.

2. Either current or next composition mode is $\land$ and answer is not $T$.

3. If it finds an EACL that evaluates to $T$, the current composition mode is $\lor$ and the next composition mode is not $\land$. Then the function stops and returns $T$.

4. If it gets through the entire EACL array, then it return:
   - $T$ if the last EACL evaluates to $T$;
   - $F$ if the last EACL evaluates to $F$ and the following is NOT true: current mode is $\lor$ and previous EACL evaluates to $U$;
   - $U$ otherwise;

Algorithm 3: policy set evaluation function $eval\_pre\_policy\_set(E, q)$:

begin
answer = U
EACL\_array = by\_priority(E)
for all i: 1 $\leq$ i $\leq$ n
   previous\_answer = answer
   answer = $eval\_pre\_policy(EACL\_array[i], q, next\_mode)$
   /* Decide whether to stop based on the current and next composition modes */
   if (i > 1) /* it is not the first EACL in the array */
if (in_channel_mode = ∨) and (answer = T) and (next_mode ≠ ∧) then return answer endif
if (in_channel_mode = ∧) and (answer ≠ T) then return answer endif
endif
/* Decide whether to stop based on the next composition mode */
if ( (next_mode=□) and (EACL_array[i].w > EACL_array[i+1].w)) or ((next_mode = ∧)
and (answer ≠ T)) then return answer endif
if (EACL_array[i].w = EACL_array[i+1].w) then current_mode = ∧ /* priorities are equal */
else current_mode = next_mode endif
endfor
if ( (answer = F) and (previous_answer = U) and (EACL_array[n-1].m = ∨)) then return U
endif
return answerend

4.5.3 Summary

We have defined the EACL model. The sets $C$, $R$, $R$, $Q$, $W$, $M$ are the same as defined
for Model with Priorities.

Set $P$ is redefined as given in (4.56). The new set $E$ and function prune are introduced.
The functions by right and by object are redefined according to (4.58) and (4.55). We
redefined the functions eval pre policy set, eval rr policy set, eval mid policy set
and eval post policy set as given in (4.63), (4.64), (4.65) and (4.66).
We redefined the functions \texttt{eval\_pre\_policy}, \texttt{eval\_rr\_policy}, \texttt{eval\_mid\_policy} and \texttt{eval\_post\_policy} as given in (4.59), (4.60), (4.61) and (4.62).

The functions \texttt{get\_policy}, \texttt{authorization}, \texttt{execution\_control} and \texttt{post\_execution\_actions} are redefined as follows:

\[
\text{get\_policy}(E, q) = \text{by\_right}(\text{by\_object}(E, q), q) = \{ < \hat{\alpha}, \hat{r}, k_{pre}, k_{rr}, k_{mid}, k_{post} > \in P \mid \hat{\alpha}, \hat{r} \in q \} , \quad q \in Q .
\]

\[
\text{authorization}(E, q) = \text{eval\_pre\_policy\_set}(\text{get\_policy}(E, q), q) \land \\land \text{eval\_rr\_policy\_set}(\text{get\_policy}(E, q), q) = b , \quad q \in Q , \quad b \in B .
\]

\[
\text{execution\_control}(E, q) = \text{eval\_mid\_policy\_set}(\text{get\_policy}(E, q), q) = b , \quad q \in Q , \quad b \in B .
\]

\[
\text{post\_execution\_actions}(E, q) = \text{eval\_post\_policy\_set}(\text{get\_policy}(E, q), q) = b , \quad q \in Q , \quad b \in B .
\]

The functions \texttt{eval\_cond}, \texttt{eval\_cond\_block} are not changed.
4.6 Changes in the Set Membership

Exercising access rights can result in creating new objects and deleting existing objects. The creation of an object $o$ entails:

1. addition of $o$ to the set $O$.

2. creating sets of access rights $R' = \{r_1, r_2, \ldots, r_m\}$ ($R' = \{-r_1, -r_2, \ldots, -r_m\}$) defined for the object $o$. Each element of the set $R'$ ($\overline{R'}$) is added to the set $R$ ($\overline{R}$) only if the set does not contain the element already. For example, lets consider a new file created with associated rights $\{\text{read, write, execute}\}$ ($\{\text{-read, -write, -execute}\}$). If our set of objects contains at least one other file with the same rights, the set $R$ already have the specified rights.

3. creating a set of condition constants $C' = \{c_1, c_2, \ldots, c_m\}$ included into condition blocks, associated with the sets of access rights. Each element of the set $C'$ is added to the set $C$ only if the set does not contain this element.

4. creating an EACL $e$ (or a set of EACLs $E' = \{e_1, e_2, \ldots, e_k\}$ with different priorities) associated with the object $o$.

5. addition of $e$ (or the elements of the set $E'$) to the set $E$.

Similarly, the deletion of an element $o$ from the set $O$ results in:

1. deletion of $o$ from the set $O$.

2. deletion of elements of the associated set of access rights $R' (\overline{R'})$ from the set $R (\overline{R})$ only if there are no other objects that have the same sets of associated rights. Going back to our example, if the set of objects contains at least one other file with the same rights, the rights can not be deleted from the set $R (\overline{R})$. 

88
3. deletion of each element of the set $C'$ from the set $C$ only if there are no other objects that have these condition constants included into condition blocks, associated with the sets of access rights.

4. deletion of all EACLs associated with the object $o$ from the set $E$.

The sets $B$, $W$ and $M$ are fixed.

The creation and deletion of all EACLs associated with the object is done automatically when an object is created or deleted. The issue of how the policies in the newly created EACLs are assigned is not directly addressed in our model.

If we allow rights to be applied to the elements of $E$, we will have to consider a policy management model.

Policy management determines who is responsible for granting and revoking of the authorization policies. The policy management problem is dealt with by introducing a special set of control rights $\hat{R}$, which is disjoint from the sets of access rights $R$ and $\overline{R}$ defined in (4.2) and (4.37). The minimal set of control rights associated with a policy object, includes create, revoke, modify and query.

We model policy management by applying our conceptual model to the set of objects $O = E$ and considering the set of control rights $\hat{R}$.

The management of EACLs, including giving authority to modify an EACL, is supported through inclusion of EACEs, which specify the control rights with associated conditions:

$$\{p_1 \lor p_2 \lor \ldots \lor p_m\}, \quad r_i \in p_i, \quad r_i \in \hat{R}, \quad i = 1, m,$$

Thus, each EACL $e$ is associated with itself $e \leftrightarrow e$. Note that the control right that corresponds to creation of a new EACL has to be governed by a separate policy. In implementation, a default policy can be associated with newly created objects. Then the policy can be further modified by authorities specified in the default policy.
The *authorization* function is called to determine whether the EACL management request should be granted or not:

\[
\text{authorization}(E, q) = \\
= \text{eval\_pre\_policy\_set(get\_policy(E, q), q)} \land \text{eval\_rr\_policy\_set(get\_policy(E, q), q)} = b ,
\]

\[
q = < a, r, < c_1, c_2, ...c_n >>, a \in O , O = E, r \in \hat{R}, c_i \in C, i = 1, n, b \in B .
\]

For example, a policy such as “Tom can read file A and query the EACL for the file A. Joe can read and write file A. Joe can query and modify the EACL for the file A.” can be represented by the following EACL associated with the file A:

\[
ed = (w, m, < w_1, m_1, \text{read}, (Tom, Joe) > \lor < w_2, m_2, \text{query\_EACL}, (Tom, Joe) > \lor \]

\[
\lor < w_3, m_3, \text{write}, Joe > \lor < w_4, m_4, \text{modify\_EACL}, Joe > .
\]

The EACEs contain only pre-conditions. Note that the access rights \text{read} and \text{write} are defined for the file \text{A}, whereas the control rights \text{query\_EACL} and \text{modify\_EACL} are applied to the EACL \text{e}.

### 4.7 Summary

This chapter presented a formal description of the Conceptual Model using the set and function formalism. The formal presentation was based on progression of models from simple to higher complexity. The most complex and expressive of the presented models is the EACL model that unambiguously describes the implemented system. The relationship of the modeled features to properties found in existing systems was discussed. The next chapter demonstrates the expressiveness of the model by presenting examples of how existing models can be mapped to the EACL model.
Chapter 5

Emulation of Various Models

The expressive power of our model depends on how well security policies map to objects, access rights and conditions. We believe that common security policies map well. In this chapter we provide examples of using our model to express the security models discussed in Chapter 1.

5.1 Lattice-based Policies

Our model allows incorporation of Mandatory Confidentiality, Mandatory Integrity models and their combination.

Recall, that the mandatory policies govern access on the basis of classification of subjects and objects in the system. Objects and subjects are assigned security labels.

Let $C = \{c_1, c_2, ..., c_n\}$ be a partially ordered set of confidentiality labels, such as unclassified, secret, top-secret, with ordering relation $\leq$.

Let $I = \{i_1, i_2, ..., i_3\}$ be a partially ordered set of integrity labels, such as low-integrity, medium-integrity, high-integrity, with ordering relation $\leq$.

Let $M = \{m_1, m_2, ..., m_n\}$ be a set of single security labels for both confidentiality and integrity, such as top-secret/low-integrity, secret/medium-integrity and so on, with ordering relation $\leq$. 

91
Every object and subject in the system bears one of the labels from the sets $C$, $I$ or $M$. Assume that the labels $c_o \in C$, $i_o \in I$, and $m_o \in M$ denote object’s classification, integrity label and combined classification/integrity labels, respectively. Similarly, assume that the labels $c_s \in C$, $i_s \in I$, and $m_s \in M$ denote subject’s clearance, integrity label and combined clearance/integrity labels, respectively.

All access rights are divided into read-class and write-class\(^1\).

To represent the lattice-based models, we introduce a number of conditions. A single read condition can represent BLP and Biba models. The combined confidentiality/integrity model can be represented by either a single read condition or a condition block that consists of a conjunction of two read conditions. Recall, that a read condition is represented as $X \, op \, P$, where $X$ is the name of a system variable, $P$ is a constant and $op$ is the operation.

The constant $P$ that represents object’s security label can be specified as a numeric or string constant when a policy for the object is created. The system variable $X$ that represents the subject’s security label has to be either passed with the request or read using the abstract $s.Read(X)$ operation. The operation $op$ is either $\leq$ or $\geq$. In implementation, to prove eligibility to access an object, a subject has to hold a valid credential, stating subject’s security label.

1. Conditions to represent MAC Confidentiality (BLP)

- $c_o \leq c_s$

  This condition is associated with the read-type access rights and is used to enforce ”read down” mandatory confidentiality rule. It specifies that a subject, wishing to get read-class access to the object has to have security clearance no less than the object’s confidentiality label.

\[^1\text{This may not be always possible. Some access rights require both read and write access to object, e.g., credit and debit to the account balance.}\]
\[ c_0 \geq c_s \]
This condition is associated with the write-type access rights and is used to enforce "write up" mandatory confidentiality rule. It specifies that a subject, wishing to get write-class access to the object has to have security clearance less or equal to the object’s confidentiality label.

2. MAC Integrity (Biba)

\[ i_0 \geq i_s \]
This condition is associated with the read-type access rights and is used to enforce "read up" mandatory integrity rule. It specifies that a subject, wishing to get read-class access to the object has to have integrity clearance less or equal to the object’s integrity label.

\[ i_0 \leq i_s \]
This condition is used to enforce "write down" mandatory integrity rule. It specifies that a subject, wishing to get write-class access to the object has to have integrity clearance no less than the object’s integrity label.

3. Combined MAC

\[ m_0 = m_s \]
This condition is associated with the read- and write-type access rights and specifies that a subject, wishing to get read- or write-class access to the object has to have security label equal to the object’s security label.

A more useful situation where both confidentiality and integrity are of concern is to use independent labels. So, each security class consists of a confidentiality label and an integrity label, with BLP controls applied to the former and Biba controls to the latter.
• \([c_o \leq c_s, i_o \geq i_s]\)

This condition block is associated with the read-type access rights and is used to enforce "read down" mandatory confidentiality and "read up" mandatory integrity rule. The condition specifies that a subject, wishing to get read-class access to the object has to have security clearance no less than the classification of the object; and integrity label less or equal to the one of the object’s integrity label.

• \([c_o \geq c_s, i_o \leq i_s]\)

This condition block is associated with the write-type access rights and is used to enforce "write up" mandatory confidentiality and "write down" mandatory integrity rule. The condition specifies that a subject, wishing to get write-class access to the object has to have security clearance no less than the classification of the object; and integrity label less or equal to the object’s integrity label.

5.2 Chinese Wall Model

The Chinese Wall model requires that a consultant not be able to read information for more than one company in any given Conflict Of Interest class (COI), for example, banks and toy companies. Thus, if a consultant accesses Bank1 information, he is not allowed to access any information within the same COI, for example that of Bank2. However, the consultant can access information of another COI class, for example, toy Company1. Thus, the representation of the Chinese Wall model requires the knowledge of the information (history) the consultant accessed in the past.
We describe the solution in the context of the specific example illustrated in Figure 5.1, which contains two COI classes with two entities in each class. Assume that object fileA belongs to the COI class Banks and to the dataset Bank1. The Chinese Wall model can be implemented using:

1. read conditions \( accessed_{DS} = P \) and \( accessed_{COI} = P \).

   These condition read the system variables \( accessed_{DS} \) and \( accessed_{COI} \), which represent the information about the datasets and a COI class accessed by a user. The value is compared to the constant \( P \). These conditions must be implemented as pre-conditions.

2. write conditions \( upd\_accessed_{DS}, new\_value \) and \( upd\_accessed_{COI}, new\_value \).

   These conditions update the system variables \( accessed_{DS} \) and \( accessed_{COI} \) with the new information about the datasets and a COI class accessed by a user. These conditions must be implemented as post-conditions. The value of these conditions must indicate whether the condition is evaluated on success or failure of the execution of requested operation.
The history information is maintained by the system and is accessed by the implementation of the abstract operations \( \text{Read}(X) \) and \( \text{Write}(X, \text{new\_value}) \). The information can be centralized or distributed.

Assume that objects \( \text{fileA} \) and \( \text{fileB} \) are governed by an EACL that includes the following EACFs:

\[
< w_j, m_j, \text{read}, [\text{Tom}, \text{accessed\_DS} = \emptyset]_{\text{pre}},
\]

\[
[\text{upd\_accessed\_DS : on\_success}/\text{Tom\_Bank1}, \text{upd\_accessed\_COI : Tom\_Banks}]_{\text{post}} > \bigvee
\]

\[
\bigvee < w_k, m_k, \text{read}, [\text{Tom}, \text{accessed\_DS} = \text{Tom\_Bank1}]_{\text{pre}} > \bigvee
\]

\[
\bigvee < w_i, m_i, \text{read}, [\text{Tom}, \text{accessed\_COI} \neq \text{Tom\_Banks}]_{\text{pre}},
\]

\[
[\text{upd\_accessed\_DS : on\_success}/\text{Tom\_Bank1}, \text{upd\_accessed\_COI : Tom\_Banks}]_{\text{post}} >
\]

These EACFs specify that \( \text{Tom} \) can read \( \text{fileA} \) only if:

- \( \text{Tom} \) has not accessed any object in the system. If this condition is met and the access succeeds, the system variables \( \text{accessed\_DS} \) and \( \text{accessed\_COI} \) are updated, or

- Any object already accessed by \( \text{Tom} \) is only from the dataset \( \text{Bank1} \).

- \( \text{Tom} \) has not accessed any object from the COI \( \text{Banks} \). If this condition is true and the access succeeds, the system variables \( \text{accessed\_DS} \) and \( \text{accessed\_COI} \) are updated.

96
Assume that objects \textit{fileC} and \textit{fileD} are governed by an EACL that includes the following EACEs:

\[
<w_j, m_j, \text{read}, [\text{Tom, accessed}_{DS} = \emptyset]_{\text{pre}},
\]

\[
[\text{upd}_{\text{accessed}_{DS}} : \text{Tom\_Bank2}, \text{upd}_{\text{accessed}_{COI}} : \text{Tom\_Banks}]_{\text{post}} > \bigvee
\]

\[
\bigvee < w_k, m_k, \text{read}, [\text{Tom, accessed}_{DS} = \text{Tom\_Bank2}]_{\text{pre}} > \bigvee
\]

\[
\bigvee < w_l, m_l, \text{read}, [\text{Tom, accessed}_{COI} \neq \text{Tom\_Banks}]_{\text{pre}},
\]

\[
[\text{upd}_{\text{accessed}_{DS}} : \text{Tom\_Bank2}, \text{upd}_{\text{accessed}_{COI}} : \text{Tom\_Banks}]_{\text{post}} >
\]

The sequence of three requests on behalf of Tom to read \textit{fileA}, read \textit{fileB} and read \textit{fileC} are evaluated as follows:

1. The first request succeeds: The EACL for the object \textit{fileA} is evaluated. The history of accessed datasets is checked, and since initially it is empty, the right is granted and if the access succeeds the access information is stored in the system variables.

2. The second request succeeds: The EACL for the object \textit{fileB} is evaluated. The pre-condition in the first EACE fails. Next the pre-condition in the second EACE is evaluated. The history of the accessed datasets is checked. Since the only accessed dataset is \textit{Tom\_Bank1} the right is granted.

3. The third request is denied: The EACL for the object \textit{fileC} is evaluated. The pre-condition in the first EACE fails. Next the pre-condition in the second EACE is evaluated. The history of accessed COI is checked, and since the accessed COI is \textit{Tom\_Banks} the right is denied.


5.3 Clark-Wilson Model

Clark-Wilson access triples \(<User ID, WFT_i, \{CDI_k, CDI_l, ..., CDI_n\}>\) are rather simply expressed using our model.

The constrained data items are represented in our model as the objects to be protected. The well formed transactions that manipulate the constraint data items are represented as access rights. Thus, for each of the objects \(CDI_k, CDI_l, ..., CDI_n\) we define a policy \(p = \{WFT_i, User ID\}\), where the access right \(WFT_i\) has an associated access identity condition \(User ID\).

In practice, a possible way to ensure the authenticity of the trusted programs that modify the objects is using an additional condition that represents the checksum or indicates that a valid certificate, stating that the program has been endorsed by the specified endorser, must be presented.

To ensure static separation of duty, each user must be permitted to use only certain set of programs. This rule can be enforced by the security administrator when assigning the authorizations.

The history of the invoked operations by the same subject can be used to implement the dynamic separation of duty. Read and write conditions (similar to the ones described in the previous section) can be used to read and update the system variables that represent the history of executed operations.

5.4 Groups and Roles

A group is a convenient method to associate a name with a set of subjects and to use this group name for access control purposes.

In general, a principal may be a member of several groups. By default, a principal operates with the union of privileges of all groups to which it belongs, as well as all of his individual privileges.
Some applications adopt role-based access control. The concept of roles is not consistent across different systems. Several definitions of roles are present in the literature. In general, a role is named collection of privileges needed to perform specific tasks in the system. Role properties [43] include:

- A user can be a member of several roles
- Role can be activated and deactivated by users at their discretion.
- Authorizations given to a role are applicable only when that role is activated.
- There may be various constraints placed on the use of roles, e.g. a user can activate just one role at a time.

Sandhu et al. [67] view roles as a policy and groups as a mechanism for role implementation. We adopt this point of view. In our model we suggest implementing different flavors of roles using the notion of group and a set of restrictions on granted privileges.

In general, a role brings together a collection of subjects and a collection of access rights. Consider a role-based policy, which assigns users: Tom, Joe, and Ken role Bank-Teller. This role allows a legitimate user to perform deposit and withdraw operations on objects account1 and account2.

This policy may be easily expressed using our EACL Model:

A group Bank.Teller is defined which includes Tom, Joe, and Ken, who are issued the group membership certificates. The EACLs for objects account1 and account2 include the following entries:

\(< w_i, m_i, deposit, X = Bank.Teller > \lor \langle w_j, m_j, withdraw, X = Bank.Teller >.\)

The condition \(X = Bank.Teller\) is a read pre-condition. The evaluation of this condition involves checking the valid group membership certificate.

In expressing role-based policy using groups, the issue of constraints on role activation and use should be addressed by defining conditions that support privilege constrains.
In assigning privileges, one can choose to have the subject operate with the privilege of only one group at a time. This can be used to reduce privileges as a protection against accidents. For example, a person is a member of two groups: *Programmers* and *System managers*. The person may act with the privileges of the group *Programmers* most of the time, and enable privileges of the *System managers* group only on occasion.

This policy can be represented by two read conditions: $X = Programmers$ that is associated with normal access rights; and $X = System managers$ associated with more critical access privileges. Similarly, privilege constraints may allow a subject to operate with privileges of several specified groups at a time. This policy can be represented by a read condition, for example $X \subseteq \{Programmers, Users\}$. The implementation of these conditions will require a history of active groups to be kept for each subject.

Another example of the privilege constraints is endorsement: concurrence of $N$ of $M$ ($N \leq M$) principals to perform some operation. For example, a policy requiring endorsement by any two persons from the set of endorsers $\{Tom, Joe, Ken\}$ can be represented by a condition block consisting of a disjunction of three read conditions: $\ (X = \{Tom, Joe\}, X = \{Joe, Ken\}, X = \{Tom, Ken\}$).

In implementation, the concurrence of several subjects to perform some operation can be accomplished with several digital signatures.

### 5.5 Summary

This chapter has shown how different models including Lattice-based models, Chinese Wall, Clark-Wilson and Role-based models can be mapped to the EACL model. This demonstrates the flexibility and expressive power of the EACL model.
Chapter 6

Implementation Framework

In this chapter we describe our implementation framework that applies the concepts of the Conceptual Model and EACL Model to the design of an authorization system. The chapter begins by describing the major components of the architecture. The chapter continues with an overview of the implementation, describing the principal components of the implemented system: the GAA-API and EACL language and showing how they fit together. The integration with different applications is described. We conclude the chapter by discussing the prototype evaluation.

6.1 Architecture

Our implementation framework is applied to distributed systems that span multiple autonomous administrative domains without a central management authority. Applications may impose their own security policies and use different authentication services, e.g., Kerberos [59], DCE [26] or X.509 [21] certificates.

The individual security requirements of each application are reflected in application-specific security policies. There might exist common policies that apply to sets of applications.
Therefore, we designed a flexible and expressive mechanism for representing and evaluating authorization policies. It is general enough to support a variety of security mechanisms based on public or secret key cryptosystems, and it is usable by multiple applications supporting different operations and different kinds of protected objects.

The major components of the architecture are:

- An authentication service performs authentication of users and supply identity credentials.

- A group service maintains group membership information. The service supports the definition of user groups that can be structured into a hierarchy. A user can be member of a set of groups with different security profiles, which aids administrators in the definition, assignment, and maintenance of security policies for a group of users.

- An intrusion detection service monitors system behavior for evidence of malicious or suspicious application activity in real time. The service maintains system threat level information.

- The GAA-API. Applications call GAA-API routines to check authorization against an authorization model. The API routines obtain policies from local files, distributed authorization servers, and from credentials provided by the user. They combine local and distributed authorization information under a single API based on the requirements of the application and applicable policies, and request credentials if needed, and may also have side effects.

- Delegation is supported by delegated credentials, such as restricted proxies [57], or through other delegation methods.
6.2 Generic and Specific Conditions

The GAA-API may not understand some of the conditions, expressed in EAACL entries. The problem of condition evaluation is addressed by using generic or application-specific evaluation functions.

Generic conditions are evaluated within the GAA-API. If generic conditions are not sufficient for expressing application-specific security policies, applications specify their own conditions. Anything that can be expressed as an alphanumeric string can be a condition. The guidelines for the condition specification are given in Section 3.10 describing the pre-, request-result-, mid- and post conditions; read and write conditions.

The application must provide evaluation rules for the application-specific conditions and register them with the GAA-API, or be prepared to evaluate the conditions once the GAA-API calls complete.

6.3 Implementation of the Set Elements

The elements of the set of priorities $W$ defined in (4.44) are represented by positive integers. The elements of the set of composition modes $M$ given in (4.45) are represented by integers:

- 0 corresponds to $\text{expand}$ mode;
- 1 corresponds to $\text{narrow}$ mode;
- 2 corresponds to $\text{stop}$ mode.

The elements of the set of truth values $B$ given in (4.8) are represented by character strings $\text{GAA.C.YES (T)}$, $\text{GAA.C.NO (F)}$ and $\text{GAA.C.MAYBE (U)}$. The elements of the sets of positive and negative rights $R, \overline{R}$ and conditions $C$, defined in (4.2), (4.37) and (4.18) are represented in our framework by tokens. Each token consists of:

- **Token Type**

  Defines the type of the token: access right or condition. For conditions, the type also defines membership in a particular condition set (e.g., access identity or time)
and specifies whether the condition is a pre-, request-result-, mid- or post condition. For access rights, the type specifies whether the right is positive or negative.

- **Defining Authority**

  Indicates the authority responsible for defining the value within the *Token Type*, e.g., Kerberos [59] or X.509 [21] for access *identity* condition type.

- **Value**

  The value of the token. The token type determines its semantics. The name space for the value is defined by the *Defining Authority* field.

An example of a positive access right token:

*Token Type*: *pos_access_right*

*Defining Authority*: *UNIX*

*Value*: *r*

An example of a negative access right token:

*Token Type*: *neg_access_right*

*Defining Authority*: *ssh*

*Value*: *host_login*

An access right with both fields *Defining Authority* and *Value* equal to *, matches any access right.

An example of a condition token:

*Token Type*: *pre_cond_access_id,CA*

*Defining Authority*: *X509*

*Value*: */C=US/O=Globus/CN=Globus CA*

The read vs. write distinction shows up implicitly in the condition type. A condition evaluation function registered with a condition type knows whether the condition is **read** or **write**. It then parses the condition value and calls the concrete functions that implement
the abstract \textit{Read} and \textit{Write} operations described in Section 3.7. The system variables manipulated by the \textit{Read} and \textit{Write} operations, as well as the operations themselves can be either local or remote. However, our framework requires that the \textit{Read} and \textit{Write} operations must be implemented as atomic actions. The GAA-API is structured to support the addition of modules for evaluation of new conditions.

\subsection{Specification and Evaluation of Access Identity Condition}

The \textbf{access identity} condition is used when user authentication is required. Our authorization framework supports the following types of \textbf{access identity}: USER, HOST, APPLICATION, CA (Certification Authority) and GROUP.

Our implementation supports multiple existing principal naming methods. Different administrative domains might use different authentication mechanisms, each having a particular syntax for specification of principals. Therefore, \textbf{Defining Authority} for \textbf{access identity} indicates the underlying authentication mechanism used to provide the identity of a subject. Value represents the particular subject's identity (principal).

It is still necessary that a common authentication mechanism be supported between two communicating systems. The framework we present enables the syntactic specification of multiple authentication policies and the unambiguous identification of principals in each, but it does not translate between heterogeneous authentication mechanisms.

In order to evaluate \textbf{access identity} condition, the application must obtain the authenticated subject’s identity and store it in the security context. The construction of the context is briefly described in Section 6.5.3.
6.4 Extended Access Control Lists (EACLs)

The policy language that we implemented is called Extended Access Control List (EACL).

The EACL language is equivalent to the one described in Section 4.5 with a few minor modifications. An **EACL** represents an element $e$ (defined in (4.53)) that consists of:

1. a composition mode;
2. an ordered set of disjunctive EACL entries.

The EACL priority is implicit and is omitted from the EACL specification.

An **EACL entry (EACE)** represents an element $p$ defined in (4.56)) that consists of:

1. An optional composition mode field;
   If this field is omitted, the EACE composition mode is equal to the composition mode of the EACL that contains the EACE.
2. An optional priority field;
   By default, entries that appear in the beginning of an EACL have higher priorities.
   The priority field is used to explicitly specify the priority of an entry and to override the default setting. The priority consists of two numbers: a priority and the order within the priority.
3. A positive or negative access right;
4. An optional condition block of pre-conditions;
5. An optional condition block of request-result-conditions;
6. An optional condition block of mid-conditions;
7. An optional condition block of post-conditions.
Note that a condition block can be empty. If all condition blocks in an EACL entry are empty, the right is granted unconditionally. An example of a practical policy with empty condition blocks is: “anyone can read file *index.html*”.

To simplify the implementation of a condition block defined in Section 4.2, we allow only a conjunction of conditions in a condition block. Therefore, an EACL is equivalent to a disjunctive normal form consisting of a disjunction of conjunctions where no conjunction contains a disjunction. For example, a policy “Tom or Joe can read file *A* only if they connect from **.isi.edu domain” can be represented by an EACL (attached to the file *A*) with two EACL entries:

< $m_i, <$read, [Tom, *.isi.edu]$pre > $\lor$ <$read, [Joe, *.isi.edu]$pre > $.$

### 6.4.1 EACL Syntax

We use the Backus-Naur Form to denote the elements of our EACL language. Items inside round brackets, ( ), are optional. Curly brackets, { }, surround items that can repeat zero or more times. A vertical line, |, separates alternatives. Items inside double quotes are the terminal symbols.

An EACL is specified according to the following format:

\[
\text{eacl} ::= \text{eac1}\_\text{mode} \{\text{eac1}\_\text{entry}\}
\]

\[
\text{eac1}\_\text{entry} ::= (\text{entry}\_\text{mode}) (\text{priority}) \text{positive}\_\text{access}\_\text{right} \text{conditions} | (\text{entry}\_\text{mode}) (\text{priority}) \text{negative}\_\text{access}\_\text{right} \text{conditions}
\]

\[
\text{positive}\_\text{access}\_\text{right} ::= "pos\_\text{access}\_\text{right}" \text{ def}\_\text{auth} \text{ value}
\]

\[
\text{negative}\_\text{access}\_\text{right} ::= "neg\_\text{access}\_\text{right}" \text{ def}\_\text{auth} \text{ value}
\]

\[
\text{conditions} ::= \text{pre}\_\text{conds mid}\_\text{conds rr}\_\text{conds post}\_\text{conds}
\]

\[
\text{pre}\_\text{conds} ::= \{\text{condition}\}
\]

\[
\text{mid}\_\text{conds} ::= \{\text{condition}\}
\]

\[
\text{rr}\_\text{conds} ::= \{\text{condition}\}
\]
post_conds ::= \{condition\}
eacl_mode ::= "0"|"1"|"2"
entry_mode ::= "0"|"1"|"2"
priority ::= "order" numeric_string numeric_string
condition ::= cond_type def_auth value
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string

Next we present an example of an EACL that governs access to a host.
Entry 1 specifies that Tom can not login to the host.

Entries 2 and 3 mean that logins from the specified IP address range are permitted, using either X509 [21] or Kerberos [59] for authentication if the number of previous login attempts during the day does not exceed 3. If the request fails, the number of the failed logins for the user should be updated. The connection duration time must not exceed 8 hours.

Entry 4 means that anyone, without authentication, can check the status of the host if he connects from the specified IP address range.

Entry 5 specifies that host shut downs are permitted, using Kerberos for authentication. If the request succeeds, the user ID must be logged. If the operation fails, the sysadmin must be notified by e-mail.

# EACL for host malta.isi.edu
eacl_mode 0 # composition mode is set to expand
# EACL entry 1
neg_access_right test host_login
pre_cond_access_id_USER KerberosV.5 tom@ORGB.EDU
# EACL entry 2

pos_access_right test host_login
pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id_USER X509 "/C=US/O=Trusted/OU=orgb.edu/CN=partnerB"
pre_cond_threshold local ≤3failures/day/failed_log
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≤8hrs

# EACL entry 3

pos_access_right test host_login
pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id_USER KerberosV.5 partnerb@ORGB.EDU
pre_cond_threshold local ≤3failures/day/failed_log
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≥8hrs

# EACL entry 4

pos_access_right test host_check_status
pre_cond_location IPsec 10.1.1.0-10.1.200.255

# EACL entry 5

pos_access_right test host_shut_down
pre_cond_access_id_GROUP KerberosV.5 trusted@ORG.A.EDU
rr_cond_audit local on:success/info:userID
post_cond_notify local email/to:sysadmin/on:failure
6.4.2 Delegation

To support the delegation of access rights, we define `pre.access_id_deleg` condition. This condition specifies authenticated access identity that can delegate the access right.

The evaluation of such condition succeeds if:

(i) the GAA-API security context contains an identity credential for the principal (this means that the request has been issued on behalf of this principal);

or

(ii) the GAA-API security context contains a chain of delegated credentials that passes the requested access right from the principal specified in the value of the condition to any of the principals specified in the identity credentials contained in the security context.

For example, consider an EAACL associated with a file doc.txt, that contains a following EACE:

```plaintext
pos.access_rights UNIX w
pre_cond_access_id_deleg KerberosV.5 tom@ISI.EDU
```

Assume the following credentials are stored in the security context:

Identity credential:

```plaintext
access_id_USER kerberos.v5 Joe@ISI.EDU
```

Group membership credential:

```plaintext
access_id_GROUP kerberosV5 admin@ISI.EDU
```

Delegated credentials:

```plaintext
grantor: grantor_id_USER kerberosV5 tom@ISI.EDU
grantee: access_id_USER kerberosV5 ken@ISI.EDU
objects: doc.txt
rights: pos_access_right UNIX w
```
grantor: grantor_id_USER kerberosV5 ken@ISI.EDU
grantee: access_idGROUP kerberosV5 admin@ISI.EDU
objects: doc.txt
rights: pos_access_right UNIX w

Let us consider a request from a user Joe to write to the file doc.txt. The EACL grants
the requested operation, however there is a condition. The evaluation function for the
pre_cond_access_id_deleg condition will first check the security context for the identity
credential for principal tom@ISI.EDU. The proper credential is not found. Next, the eval-
uation function will look for a chain of delegated credential rooted at the credential issued
by joe@ISI.EDU that passes access right w for the file doc.txt to principals joe@ISI.EDU
or admin@ISI.EDU. The appropriate chain is found. The condition is satisfied, so the
requested access is granted.

In many systems the redelegation must be controlled. This condition can be combined
with other conditions to restrict the re-delegation rules. For example, it may be useful to
limit authorities that can re-delegate and the length of a credential chain.

6.4.3 EACL Specification and Evaluation

The implementation of the EACL evaluation mechanism is based on the Conceptual Model
corcepts described in Chapter 3:

- The closed world model interpretation;
- Both negative and positive authorizations are allowed;
- Inconsistencies are resolved according to the ordered interpretation of EACLs [71].
- The three-phase policy enforcement model.
Evaluation of ordered EACL starts from the first to the last in the list of EACL entries. The authorizations that already have been examined take precedence over new authorizations. Other interpretations were possible, but we found that for many such policies, resolution of inconsistencies was either NP-Complete or undecidable.

There may be interactions when independent credentials are used, e.g., one set of credentials causes denial, but the other causes accept. A user may choose to withhold credentials that it believes may result in a denial. The administrator must deal with these issues by carefully setting policies in an EACL and by carefully combining attributes in credentials. Conflicts may arise when more than one entry applies. For example, one matching entry specifies individual subject (user, host or application), and another matching entry specifies a certain group name. In this case, we would require the entry for the individual subject to be placed before the entry for the group (assuming the policy expressed for the individual subject entry is an exception to the policy expressed for the group entry).

An ordered evaluation approach is easier to implement as it allows only partial evaluation of an EACL and resolves the authorization conflicts. The problem with this approach is that it requires total ordering among authorizations. It requires careful writing of the EACL by the security administrator and is error-prone. An improper order of the EACL entries may result in discrepancies between the intended policy and the one that results from evaluation of the EACL. It might be useful to have a separate module [43], [16], that would help verify and debug the EACL to assure that it expresses the policy the administrators intend the system to enforce.

### 6.4.4 Policy Composition

Our implementation provides a hierarchical security policy scheme which supports both local (user- or application-specific) security policies and a system-wide security policy defined by the administrator.
System-wide policy is defined globally: it grants or restricts access to resources to all
users (or groups of users) of the system. Local policy provide the possibility for users and
applications to define their own policy in addition to the global one. The system-wide
policy is stored at a global location while the local policy is stored locally and is accessible
for the user or application. The GAA-API stores policies as ASCII text files (expressed
using the EACL syntax described in Section 6.4.1) with the name of the object being the
name of the file.

The composed policy is constructed by merging the system-wide and local policies.
First, system-wide policies are retrieved and placed in the beginning of the list of policies.
Then the local policies are retrieved and are added to the list. Thus, system-wide policies
implicitly have higher priority than the local policies.

6.5 Generic Authorization and Access-control API (GAA-
API)

The GAA-API provides a general-purpose execution environment in which EACLs are
evaluated. In this section we provide a brief description of the main GAA-API routines
and structures.

6.5.1 Callback Functions

The GAA-API expects the application to provide callback functions that are used to:

- retrieve (and possibly translate) the policies;

The application maintains authorization information in a form understood by the ap-
lication. It can be stored in a file, database, directory service or in some other way.
The specific mechanism for retrieving the policies is passed as a callback function pro-
vided for the GAA-API and accessed through the GAA-API control structure. The
callback function retrieves the policy information and translates it into the internal
representation (EACL) understood by the GAA-API. In the present implementation
the policy is written at the object level, the callback function must collect all the per
object policies and order them by priority. How the policies are stored and retrieved
is opaque to the GAA-API and is not reflected in the EACL.

- support diverse credentials supplied by different security services;

The GAA-API has as a goal independence from authentication mechanisms. There-
fore it provides transparent conversion of authentication tokens to identity creden-
tials. The credential management interfaces are implemented by callback functions
that retrieve credentials, verify and evaluate them. The callbacks can also request
additional credentials, if needed.

- interpret the non-standard application-specific conditions;

The GAA-API supports generic conditions including: access identity, group mem-
berships, time, audit, expressions (including regular expressions) and location. If these
conditions are not sufficient for application to enforce application-specific policies,
the application can register specific condition evaluation functions (implemented as
callbacks).

The GAA-API provides a mechanism to register a particular callback function. This
is done using a configuration file that lists the concrete functions that implement the
callbacks. The file is read at the GAA-API initialization time and the functions are reg-
istered with the specific condition evaluation, policy retrieval and credential management
interfaces.

6.5.2 GAA-API Concepts and Typical Usage

A simple GAA-API application will do the following:

- Perform some initialization at the beginning to create a GAA-API control structure
  and security context. The GAA-API control structure includes information about
callback routines (to be used to evaluate conditions, find policy information, etc.). Callback routines may be installed in this structure by the GAA-API implementation itself or explicitly by the application at any time.

The *security context* data structure contains information about the current subject’s credentials. Credentials may be added to this structure by the GAA-API implementation itself (in the course of evaluating conditions) or explicitly by the application at any time.

- Each time the application receives a request, it will determine what access rights are necessary to fulfill that request and then call GAA-API routines to create a list of requested rights, find the relevant policy, and determine whether or not the policy grants those rights.

A *requested right* data structure includes a value and defining authority (it is a positive access right by default). A request right may also include a list of options (additional information about the request, to be used as context when evaluating conditions). The requested right structure implements the “$r, < c_1, c_2, ..., c_n>$” part of the authorization request element $q$ defined in (4.21).

An EACL is represented internally by a *policy* data structure that contains a composition mode and an ordered list of policy entries. The structure implements an element $e$ defined in (4.53).

A *policy entry* data structure implements a policy element $p$ defined in (4.56). The structure contains a priority, a composition mode and an ordered list of policy rights. A *policy right* data structure implements the “$r, k_{pre}, k_{rr}, k_{mid}, k_{post}$” part of the element $p$. The structure consists of a type ($pos\_access\_right$ for a right that’s explicitly allowed; $neg\_access\_right$ for a right that’s explicitly denied), a defining authority, a value and four lists of conditions under which the policy right applies.
Each condition list implements a condition block that contains a conjunction of conditions.

A condition data structure consists of a type, a value, a defining authority and an evaluation status that specifies whether the condition is evaluated or not. If the condition is evaluated the status further specifies whether the condition is met or not.

In a typical request, the policy will be retrieved with the `gaa_get_object_policy_info` function. If no relevant policy was found, the request is rejected.

Next the `gaa_check_authorization` function is called to determine whether the requested rights are granted or denied by the policy. If authorization is not granted, the request is rejected.

If the request is granted the `gaa_check_authorization` function stores the evaluation information in the state structure, setting flags that indicate whether mid- and/or post-conditions are present in the policies.

The GAA-API is implemented as a library, and is directly linked with the application. In the client-server implementation, the `gaa_get_object_policy_info` and `gaa_check_authorization` functions can be combined into a single client side call. On the server, the GAA-API deamon still retrieves the policy. However, it does not return it to the GAA-API client. The benefit of having two calls is the ability to retrieve and parse the policy once and to perform several access control checks.

- When the operation is started, the application calls the `gaa_execution_control` function to perform policy enforcement during the operation execution.

- When the operation is completed, the application calls the `gaa_post_execution_actions` to perform post execution policy enforcement.
• When the application is finished using GAA-API, it will call cleanup routines to release resources.

Most of the GAA-API structures have a field to store entities (e.g., policies that are not in EA CL format, credentials, request rights and so on) in their “raw” format. The raw structures are opaque (all the GAA-API sees is a handle), they supports any storage format.

The requirement to be able to convert the structures into a canonical format is not mandatory, however, the evaluation callback functions for these structures must be registered with the GAA-API. This allows the GAA-API to evaluate policies in both formats: the canonical format and the raw format.

6.5.3 GAA-API Security Context

The security context stores information used by the condition evaluation functions. Some of its constituents are listed here:

**Verified identity credentials**

Identity credentials describe a set of mechanism-specific principals, and give their holder the ability to act as any of those principals. Each of the identity credentials contains information needed to authenticate a single principal.

**Verified security attribute credentials**

These credentials specify principal’s various security attributes, such as security clearance or user’s age.

**Verified group membership credentials**

These credentials list all groups principal is a member of.

**Verified non-membership credentials**

These credentials list some groups principal is not a member of.
Verified authorization credentials

This type of credentials used when individuals grant delegated credential or generate a capability. The authorized credentials specify:

- a principal who issued the credential (grantor);
- a principal for whom the credential was issued (grantee);
- a list of objects which may be accessed by the grantee;
- a list of granted access rights and associated conditions.

Unevaluated credentials

Evaluation of the acquired credentials can be deferred till the credentials are needed to perform the operation.

All of the listed credentials may specify conditions, e.g., validity time periods.

Credentials are translated to the GAA-API internal format and placed into the GAA-API security context. When evaluating an EACL, the security context is searched for the necessary credentials.

6.5.3.1 Creation of the GAA-API Security Context

Prior to calling the GAA-API functions, the application obtains the authenticated subject’s identity and stores it in the security context. This context may be constructed from credentials obtained from different mechanisms, e.g., GSS-API [48], Kerberos [59], or others. This scenario, however, places a heavy burden on the application programmer to provide the integration of the security mechanism with the application. A second scenario is to obtain the authentication credentials from a transport protocol that already has the security context integrated with it. For example, the application can call SSL [33] or authenticated RPC. In this case, it is the implementation of the transport mechanism (usually written by someone other than the application programmer) which calls the security API requesting subject’s identity.
The subject’s authentication information is placed into the security context and passed to the GAA-API. When additional security attributes are required for the requested operation, the list of required attributes is returned (as a list of unevaluated conditions) to the application, which may further request them.

The application may provide the GAA-API with a callback functions for requesting required additional credentials. The credentials are pulled by the GAA-API are verified and added to the security context by the callback functions.

6.5.4 Main GAA-API Functions

Next we provide a brief description of the main GAA-API functions.

6.5.4.1 The gaa_get_object_policy_info Function

The gaa_get_object_policy_info function is called to obtain the security policy associated with the object.

- **Input:**
  
  - *Reference to the object to be accessed.* The identifier for the object is from an application-dependent name space, it can be represented as unique object identifier, or symbolic name local to the application.
  
  - *GAA-API control structure.*

- **Output:**

  - *GAA\_YES/GAA\_NO*, indicating success or failure.

  - *A handle to an ordered list of policies.*

This function implements the abstract function *by_object* defined in (4.55).
The resulting policy that is passed to the GAA-API for evaluation represents the combination of several policies possibly from different domains and individual users of the system.

6.5.4.2 The gaa_check_authorization Function

The gaa_check_authorization function checks whether the requested rights are authorized under the specified policy, or if additional application-specific checks are required.

- **Input:**
  - *The handle to the ordered list of policies*, returned by the gaa_get_object_policy_info function.
  - *GAA-API Security context*.
  - *Rights for authorization*. This argument indicates a list of requested access rights.

- **Output:**
  - *GAA_C_YES/GAA_C_NO/GAA_C_MAYBE*.
  - *GAA-API evaluation state*.

The state structure contains information, such as GAA-API control structure, security context, a list of requested rights, mid- and post-conditions (if any) and indication of whether mid- and post-conditions are present in the policy.

- *Detailed answer contains:*
  - Authorization valid time period.

    If the access is granted, the output includes the time period during which the authorization is granted. It is returned to be checked by the application.
The validity time is calculated by the GAA-API, based on:

1. Time-related conditions in the object policy, e.g., EACL matching entries.
2. Time-related restrictions in the security attributes, identity and authorization credentials.

* The output list of all matching policy rights and associated conditions is returned, with flags set to indicate whether each condition was evaluated and/or met. Unevaluated conditions provide information about additional security attributes required. Additional credentials might be required from clients to perform certain operations, e.g., group membership or delegated credentials.

The `gaa_check_authorization` function is called to evaluate pre- and request-result conditions for each requested right in the list. This function evaluates each requested right (finding the relevant policy rights and calling the appropriate condition-evaluation callback routines to see whether they apply) and then aggregates the results. When an authorization request is made to GAA-API, the condition-evaluation functions are passed a pointer to the requested right, which may contain a list of options. The condition-evaluation function can then look through the list of options to find any that are relevant. For example, if a condition requires that "file.size \( \leq \) 10K", then a condition-evaluation function for that condition could check to see whether there was a "file.size" option in the request (if there was, the function can return `GAA.C.YES` or `GAA.C.NO` based on the option's value; if there was no relevant option, that function must return `GAA.C.MAY_BE`).

The `gaa_check_authorization` function implements the abstract function `authorization` defined in (4,68), which is called in a loop (for each requested access right). To be precise, the `gaa_get_object_policy_info` implements the `by_object` function and is called first, so the `gaa_check_authorization` function implements the remaining part.
In particular, for each requested right the `gaa_check_authorization` function does the following:

- finds the relevant policy entries (positive and negative) in the policy list. This is equivalent to the function by_right specified in (4.58).
- evaluates pre-conditions for each relevant policy entry. This is equivalent to calling the `eval_pre_policy_set` function defined in (4.63). If there are no pre-conditions, the authorization status is set to `GAA_C.YES`. If the request-result conditions are present in the policy, the conditions are evaluated. This is equivalent calling the function `eval_rr_policy_set` given in (4.64).
- returns the conjunction of the evaluation results for pre- and request-result-conditions.

This function saves the evaluation information in the state structure for further evaluation, if needed.

### 6.5.4.3 The `gaa_execution_control` Function

The `gaa_execution_control` function performs policy enforcement during the operation execution.

- **Input:**
  - *The handle to the evaluation state*, returned by the `gaa_check_authorization` function.

- **Output:**
  - `GAA_C.YES/GAA_C.NO/GAA_C.MAY_BE`.

This function checks whether the mid-conditions associated with the granted access rights are met implementing the abstract `eval_mid_policy_set` function defined in (4.65). If no mid-conditions are found the `GAA_C.YES` is returned.
6.5.4.4 The gaa\_post\_execution\_actions Function

The `gaa\_post\_execution\_actions` function performs policy enforcement after the operation completes.

- **Input:**
  - *The handle to the evaluation state*, returned by the `gaa\_check\_authorization` function.
  - *The operation execution status* indicates whether the operation succeeds/fails.

- **Output:**
  - `GAA\_YES/GAA\_NO/GAA\_MAY\_BE`.

This function enforces the post-conditions associated with the granted access rights, implementing the abstract `eval\_post\_policy\_set` function defined in (4.66). If no post-conditions are found the `GAA\_YES` is returned.

6.6 Security Concerns

The validity of the authorization decision is dependent on the correct policies being given to the GAA-API engine, as well as the correct interpretation of the returned values. All parameters, including all policy statements, are given to the GAA-API with each query. Thus, the application is solely responsible for giving the evaluation engine correct information. Thus the GAA-API relies on the security of the applications that uses it. If the application is not secure, the application can pass GAA-API bogus policy (through callback) or bypass the GAA-API altogether.
6.7 History of the GAA-API Implementation

The first GAA-API prototype implementation was experimental. We gathered experience from integration of the GAA-API with test applications. Next two versions were integrated with Grid Security Infrastructure [37] and used a pseudo-object-oriented approach. The following version was implemented by a member of a Globus project [35]. This version abandoned the object-oriented approach and adopted a call-back approach where the structures are opaque and only the calls to create and manipulate the structures are defined. This is useful for binary compatibility of different GAA-API implementations in the future. The current version of the GAA-API has been separated from GSI to exist as a stand-alone version.

6.8 Integrating the GAA-API into Applications

To prove the usability of the GAA-API, we integrated the API with several applications.

We integrated the GAA-API with Apache server [4] and ssh daemon [79]. The integration supports Apache and ssh native policies as well as new policies.

6.8.1 Integration with Alternative Authentication Mechanisms

We have integrated the GAA-API with the Prospero Resource Manager (PRM) [58], a metacomputing resource allocation system developed at University of Southern California. PRM uses Kerberos [59] to achieve strong authentication. PRM uses calls to the Asynchronous Reliable Delivery Protocol (ARDP) [64], a communication protocol which handles a set of security services, such as authentication, integrity and payment. ARDP calls the Kerberos library through a security API, requesting the principal's authentication information.

In addition, we have integrated the framework with the Grid Security Infrastructure (GSI), a component of the Globus metacomputing Toolkit [31]. GSI is implemented on
top of the GSS-API [48], which allows the integration of different underlying security mechanisms. Currently, GSI implementation uses SSL [33] authentication protocol with X.509 [21] certificates.

Public key authentication requires consideration of the trustworthiness of the certifying authorities for the purpose of public key certification. Authentication is not based on the public key alone, since anybody can issue a valid certificate.

Certificates can comprise a chain, where each certificate (except the last one) is followed by a certificate of its issuer. Reliable authentication of a public key must be based on a complete chain of certificates which starts at an end-entity (e.g. user) certificate, includes zero or more Certification Authorities (CA) certificates and ends at a self-signed root certificate. A policy must be specified to validate the legitimacy of the received certificate chain and the authenticity of the specified keys. The following is an example of an EACE used for describing the Globus policy for what CAs are allowed to sign which certificates:

```plaintext
pos_access_rights globus CA:sign
pre_cond_access_id CA X509 /C=US/O=Globus/CN=GlobusCA
pre_cond_subjects globus "'/C=us/O=Globus/*' '/C=us/O=Alliance/*"
```

The Globus CA can sign certificates for Globus and the Alliance. In the present GSI implementation, the GAA-API is used as a core piece of the Community Authorization Service [30].

### 6.8.2 Implementation of the Delegation of Access Rights

To implement the evaluation of the `pre_cond_access_id_deleg` described in Section 6.4.2 we integrated the GAA-API with the KeyNote toolkit [15].
KeyNote is a simplified version of the PolicyMaker trust management system [16]. KeyNote specifies a simple language for expressing trust relationships between arbitrary principals, which may be cryptographic keys.

A policy in KeyNote is expressed as a combination of unsigned and signed policy assertions (signed assertions are also called credentials).

Below is an example of the use of policy assertion that delegates authority to \textit{tom} to do any action in which the attribute called \texttt{appl\_dom} is equal to the string \textit{test}.

\begin{verbatim}
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: "tom"
Conditions: app\_dom == "test"
\end{verbatim}

Delegation of some authorization from principal \textit{tom} to principal \textit{joe} is expressed as a KeyNote assertion with principal \textit{tom} given in the Authorizer field, principal \textit{joe} given in the Licensees field, and the authorization to be delegated encoded in the Conditions field:

\begin{verbatim}
KeyNote-Version: 2 Authorizer: "tom"
Licensees: "joe"
Conditions:
app\_domain == "test" &&
access\_right == "read"
Signature: "RSA-SHA1:d10i46h2"
\end{verbatim}

This KeyNote assertion means that the principal \textit{tom} authorizes the principal \textit{joe} to "read" and the attribute called \texttt{appl\_domain} is equal to the string \textit{test}. 

126
When an action is requested of a KeyNote-based application, the application submits a description of the action, called an action attribute set in KeyNote terminology. The action attribute set (described as attribute/value pairs) specifies the context of the action along with a copy of its local security policy and a set of credentials to the KeyNote interpreter that approves or rejects the action.

The semantics of KeyNote evaluation can be thought of as involving the construction a directed graph of KeyNote assertions rooted at a POLICY assertion that connects with at least one of the principals that requested the action. An action is authorized if assertions that approve the action can link the POLICY principal with the principal that authorized the action.

The root of this graph is a special (implicitly trusted) principal POLICY. The assertions and credentials form the edges of the graph, while the nodes represent the principals. Based on the action or action attributes given to the system, the graph is traversed to find a path leading from the root to the requesting principal under the conditions described in the action attributes.

When the `gaa_check_authorization` function finds the relevant policy entry with the `pre_cond_access_id_deleg` condition, the function calls the condition evaluation callback routine that translates the GAA-API structures to the KeyNote format and initiates a KeyNote session. The straightforward translations performed by the callback function include:

- The requested access right is translated to a KeyNote action string. The options associated with the requested right are translated to the action attribute set.
• The policy access right and the `pre.cond.access_id.deleg` condition are translated into KeyNote policy assertion. For example, an EACE:

```plaintext
pos_access_right file read
pre_cond_access_id_deleg keynote ken
```

is translated to KeyNote policy assertion:

```
Authorizer: "POLICY"
Licensees: "ken"
Conditions:
  access_right == "fileread"
```

• The GAA-API identity and group membership credentials are represented as a list of KeyNote action authorizers.

• The GAA-API delegated credentials are translated to the KeyNote assertions. For example, a delegated credential in the canonical format:

```plaintext
objects doc.txt
grantor assertion tom
grantee assertion ken
pos_access_right file read
pre_cond_expr keynote "@dollars<100"
```

is translated to the KeyNote assertion:

```
Authorizer: "tom"
Licensees: "ken"
```
Conditions:

access_right == "fileread"
pre_cond_exprkeynote == "@dollars)<100"

6.8.3 Outside Integration

Researches at Brown University experimented with the GAA-API for the Internet2 Shibboleth web server authorization project.

They have done several extensions to the prototype implementation:

1. Application-specific policy retrieval callback;
   They created a callback that retrieves the policy in a SQL [19] database.

2. Using the GAA-API in the client-server environment;
   The code was split into stubs and RPC routines, using Sun’s ONC RPC mechanism. The GAA-API engine can now be run on a secured hardened centrally managed server, and applications can run on arbitrary machines and submit requests.

3. Policy management;
   They implemented the policy management scheme similar to the one described in Section 4.6. The policy consists of two parts:

   - Application-specific authorization policy (in XML format) associated with the protected objects.
   - Management policy describing who can edit and manage the authorization policy.

   They provide a policy management interface that allows authenticated and authorized users to create and edit the authorization policy.

They plan to integrate the GAA-API into other applications developed at the Brown University.
6.8.4 Current Work on the Application Integration

We are integrating the GAA-API into FreeS/WAN Linux IPsec implementation. In IPsec [46], when two hosts want to communicate, they have to set up a security association (SA). An SA is an agreement between two hosts on how to process certain traffic between them. This processing involves encapsulation, authentication, encryption, or compression.

The Internet Exchange Key (IKE) protocol sets up IPsec connections after negotiating appropriate parameters.

The negotiation of SA requires a number of choices that involve tradeoffs between security, convenience, trust, and efficiency. Each host has its own policy (EACL) governing SA creation. This policy describes the SA properties, such as acceptable cryptographic algorithms and key sizes, the lifetime of the SA, logging and accounting requirements.

The GAA-API calls are added to the IPsec architecture (in FreeS/WAN a "pluto" daemon running IKE) to control the proposed choices and the elected SA characteristics during the IKE. The GAA-API is called by both hosts (Initiator and Receiver) in order to establish an SA:

- **Outbound processing**: an Initiator proposes a list of choices.
  
  The Initiator calls the GAA-API that consults local policies and returns acceptable characteristics for SA.

- **Inbound processing**: the Responder replies with one choice that it has selected.
  
  The Responder queries its GAA-API implementation to determine whether the proposed SA attributes comply with local policy and, if they do, creates the SA containing the specified parameters, otherwise returns NO_PROPOSAL_CHOSEN.
In the present FreeS/WAN IPsec implementation, the policies for SA characteristics are hardwired into the code of pluto. The GAA-API provides an alternative to the hardwired policy selection. Here is an example of FreeS/WAN EACL policy:

```plaintext
pos_access_right FREESWAN ISAKMP
pre_cond_access_method CONNECTION_NAME "B, A"
pre_cond_access_method OAKLEY_AUTH "RSA_SIG, PRESHARED_KEY"

pos_access_right FREESWAN IPSEC
pre_cond_access_method IPSEC_COMPRESS DEFLATE
pre_cond_access_time local "MON-FRI 8am-11:30am.,1pm-5:45pm"
```

Other parameters can be controlled by EACL, for example SA_LIFE_TIME, which is now hardcoded (8 hours). Furthermore, additional policies are provided by the GAA-API (for example, audit, time constraints) that are not supported by the FreeS/WAN.

Our integration approach is similar to the one taken by the KeyNote-controlled FreeBSD IPsec implementation [18]. However, the structure of the FreeS/WAN code posed additional challenges, since the pluto code is less structured than FreeBSD IPsec code. In particular, some policies are read from configuration files and some are hardwired.
6.9 Intrusion and Misuse Attack-Response Examples

In this section we describe several intrusion attack examples to illustrate how our framework can be deployed to enable fine-grained response to the attacks. First, we address a simple scenario of a generic attack, next we consider three most common categories of attacks [7]:

1. Surveillance

Intruders use techniques to scan for information, but does not do anything harmful yet. These include ping sweeps, TCP or UDP port scans, and possibly indexing of public web servers to find CGI holes.

2. Penetration

Intruders will take advantage exploits of hidden features or bugs to gain access to the system. Examples: compromise a CGI script by sending shell commands in input fields;
look for login accounts with easily guessable passwords;
explore buffer-overflow holes by sending large amounts of data.

3. Denial-of-Service (DoS)

Intruders attempt to crash a service (or the machine), overload network links, overloaded the CPU, or fill up the disk.

In the context of these attacks, we consider an enterprise LAN. We assume that the enterprise deployed a network-based intrusion detection system (NIDS) that monitors network traffic looking for patterns of intrusive behavior.

The NIDS is placed at a network gateway to the LAN. In this location, the NIDS can monitor all incoming and outgoing traffic.

Our approach focuses on the application level response to attacks. To demonstrate the attack-response scenarios, we use two security critical applications: SSH and Apache
server. These applications may have to be available from outside of organization around the clock, thus they are best targets for an intruder to gain access to a system.

6.9.1 Network Lockdown

This scenario demonstrates how our system adapts the applied authentication policies to require more information from a user when potentially dangerous activity has been detected.

This scenario is designed for organizations with the following characteristics:

- Mixed access to web services. Access to some web resources require user authentication, some do not. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it.

- Authenticated SSH connections from the Internet are allowed to access the LAN.

- An NIDS supplies a system threat level. For example, low threat level means normal system operational state, medium threat level indicates suspicious behavior and high threat level means that the system is under attack.

- Policy: When system threat level is higher than low, one needs to lock down the system and require user authentication for all accesses within the network. Strong authentication protects against outside intruders. To some extent, authentication may help to reduce insider misuse. In particular, insiders are discouraged if the identity of a user can be established reliably.
The policy requirements can be represented by the following EACL that protects all SSH and web server connections within the LAN:

```
# EACL entry 1
pos_access_right apache *
pre_cond_system_threat_level local >low
pre_cond_accessID_USER apache *
```

```
# EACL entry 2
pos_access_right ssh *
pre_cond_system_threat_level local >low
pre_cond_accessID_USER ssh *
```

The read pre-conditions in EACL entries 1 and 2 mean that all Apache and ssh accesses have to be authenticated if the system threat level is higher than low.

### 6.9.2 NIDS fine-tuning

Data collection countermeasures, e.g., network packet logging, are too costly to apply all the time. This is particularly true for NDIS that maintains information about a large number of hosts:

1. So much logged data is created that important events can not be found.
2. NIDS becomes susceptible to resource starvation attacks (DoS).

The following example shows how application-level feedback can reduce the number of logged packets and help discover more attacks.
Assume that EACLs that govern hosts within the LAN contain the following EACL entry:

```
pos_access right ssh host_login
pre_cond location IP 10.1.1.0-10.1.200.255
rr_cond notify local on:failure/NIDS/info:IP,failed.login
```

Assume that when a system threat level is low, the NIDS uses reduced packet logging. Every request from outside of the valid IP address range will result in access denial and a notification message containing the IP address and the reason will be sent to NIDS. If NIDS receives such notifications from different hosts at the same or subsequent time, this may mean ongoing surveillance or/and penetration attacks. To effectively manage multiple alerts, they must be aggregated and correlated. Thus, as a countermeasure, NIDS should elevate the system threat level and modify logging behavior, for example, to record all packets coming from the suspicious IP addresses.

However, the increased logging has to be done in a controlled manner. It is possible that the target of the attack was the NIDS. Perhaps, the attacker wanted to disable the NIDS with the DoS attack.

### 6.9.3 Application-level Intrusion Detection

In this section, we demonstrate how our framework provides real-time application-level intrusion/misuse detection capabilities.

We next show how an EACL that protects a CGI directory can help prevent Apache penetration and/or surveillance attacks by detecting CGI script abuse.

Entry 1 specifies that members of the group **BadGuys** are denied access. Evaluation of the read pre-condition **pre_cond_group** includes reading a log of suspicious IP addresses and trying to find an IP address that matches the IP address from the request.
Entry 2 contains a read pre-condition `pre_cond_regex` that examines the request for occurrence of regular expressions `*phf` and `*test-cgi`. If no match is found, the GAA-API proceeds to the next EACL entry that grants the request.

If this condition evaluates to `T` (the GAA-API detects attempts to access well-known vulnerable CGI scripts), the request is rejected and two write request-result conditions are evaluated. The `rr_cond_notify` condition calls, for example, Sendmail to e-mail the system administrator. The mail alert contains date, time, remote IP address of the attacker and URL attempted. Next, the `rr_cond_update_log` updates the group `BadGuys` to include new suspicious IP address from the request.

```plaintext
# EACL for directory /usr/local/apache/cgi-bin
# EACL entry 1
neg_access_right apache *
pre_cond_group apache BadGuys

# EACL entry 2
neg_access_right apache *
pre_cond_regex gnu '``*phf'' *test-cgi'''
rr_cond_notify local on:failure/email:sysadmin/info:CGIexploit
rr_cond_update_log local on:failure/BadGuys/info:IP
# EACL entry 3
pos_access_right apache *
```

The advantages of looking for the CGI exploits at the application level are [3]:

(i) the ability to access decrypted information about request when the request is transported to the application through encrypted SSL channel and therefore not visible to the NIDS.
(ii) the knowledge about how the request will be handled by the server. For example, whether the requested file is interpreted as a CGI script or HTML file. NIDS could not make this distinction and if configured to look for strings matching *phf and *test-cgi, would produce false positives.

The next example demonstrates detection and response to a particular DoS attack. Opening a large number of simultaneous connections to the ssh server starves the number of available sockets, disallowing new connects.

Assume that EACLs that govern hosts within the LAN contain the following EACL entry:

```plaintext
pos_access_right ssh host_login
pre_cond_access_id USER X509 *
pre_cond_threshold local <=20/active_sessions_per_user
rr_cond_update_log local on:failure/failed_log/info:userID
rr_cond_notify local on:failure/email:sysadmin/info:ssh,DoS
mid_cond_update_log local sessions_per_user/info:userID+1
post_cond_update_log local on:success/sessions_per_user/info:userID-1
```

Evaluation of the read pre-condition `pre_cond_access_id USER` asserts a proper user authentication.

The read pre-condition `pre_cond_threshold` reads the log of active sessions to determine the number of sessions with the user ID field equal to the one in the user ID credentials. If the number is greater than 20, the request is rejected and the two write request-result conditions perform actions similar to the ones described in the previous example.

If the number of such sessions is less than 20, the request is granted, the connection is established and the read mid-condition `mid_cond_update_log` is evaluated. This condition is evaluated just once, it updates the number of active ssh connections for the user. After a
connection is closed, the write post-condition \texttt{post\_cond\_update\_log} updates the number of connections reducing it by 1.

6.10 Prototype Evaluation

We evaluated the flexibility and expressive power of the model in Chapter 5. In this section we present the evaluation of the GAA-API prototype.

Evaluation is essential to judge the performance and usability the GAA-API implementation and to explore optimizations that reduce the overhead of access control. There are three levels at which a system such as this needs to be evaluated: correctness, performance and extensibility. The primary goal of the prototype is the correctness of access control decisions.

1. Correctness

To judge the system correctness characteristics, we designed a series of scenarios that cover a wide spectrum of decision situations. The scenarios include examples of various policies expressed in EACL form. For example, we tested whether the system makes correct authorization decision when negative and positive rights are involved, when some conditions are met, not met and undefined. We have tested the evaluation of EACLs with different composition modes. The prototype returned correct decisions for these scenarios.

2. Performance

Next we focused on the performance evaluation of the GAA-API.

- Overall Performance

We conducted tests to find out the impact on the performance of user applications. The end-to-end overhead of access control was determined by measuring the execution times (user and system CPU time) of the GAA-API calls. These

138
measures did not take into account the user authentication process and credential verification.

The GAA-API initialization and evaluation overhead is very low: with a small number of policies (retrieved from local files or files accessed across NFS [74]) is approximately 15 microseconds on a SUN Ultra-2 workstation. The potential performance degradation is the overhead required to fetch the policies that are stored in a database or directory service.

The policy evaluation overhead (with a small number of conditions) is approximately 10 microseconds. This number will increase with the size of policies and number of conditions that are evaluated.

- Relative Performance

While performance was not the most important issue in the design of GAA-API, it was important that the performance be comparable to that of existing access control systems. In the interest of determining the relative overhead of the GAA-API mechanism, the prototype was compared with KeyNote. We conducted tests to compare the cost for processing a request using the GAA-API and the KeyNote system. A number of simple policies with a small number of conditions where specified to test the performance. The conditions included numeric comparison, string and regular expression matching. These policies were translated to the EACL and KeyNote policy formats.

We next ran the GAA-API and KeyNote on SUN Ultra-2 workstation to evaluate these policies. The KeyNote evaluation overhead is about 10 microseconds. The GAA-API evaluation overhead is approximately 15 microseconds.

The GAA-API initialization step requires parsing a configuration file and registering call-back functions with the evaluation engine. In contrast, KeyNote system, is a closed system that does not support addition of new modules.
Therefore, the KeyNote requires less initialization overhead. As policies become larger in size, the relative GAA-API initialization overhead becomes proportionally smaller. Thus, we conclude that the GAA-API performance numbers are similar to those of the Keynote system.

3. Extensibility

We included scenarios that involve the addition of a new module - KeyNote to the GAA-API to test its extensibility. KeyNote system is called by the GAA-API to support delegation of access rights. The addition of KeyNote contributed only about 200 lines of code. Almost all of this code is related to formatting the GAA-API structures for communicating with the KeyNote.

6.11 Summary

This chapter described the prototype implementation of the EACL model. We started with a discussion of an overall architecture of the implementation. We then described how structural components of the model - sets and functions are implemented using text files (EACLs) and the GAA-API routines. The GAA-API features include extensibility, support for application-specific conditions, support for non-EACL policy evaluation and a policy composition framework. We next described the implementation of the policy composition model and delegation of access rights.

We presented several intrusion attack-response scenarios that demonstrate the usefulness and power of our framework:

- real-time response to state generated by intrusion detection engines.

In particular, active policies can adapt the applied authentication policies to require more information from a user when suspicious activity has been detected.

- fine-tuning of intrusion detection services.
The access control decisions provide useful information to intrusion and misuse detectors, making dynamic, fine-grained responses feasible. In particular, active policies can adapt the level of detail of the audit records generated so that audit overhead is reduced until an intrusion detection engine notices that something is amiss, though not necessarily what it is. In addition, we show how application-level feedback supported by the policies can help find more attacks and reduce the number of false-positives.

- application-level intrusion detection coverage.

Active policies can assist in the application-based category of intrusion detection, which monitors critical applications such as SSH or web servers.

Insider misuse tends to be strongly application-specific. Thus, protective measures should dictate what is to be monitored at the application layer. This support must be at the application because often only the application has knowledge of the operations that are requested, and the objects to be manipulated by the request.

We concluded with the evaluation of the GAA-API prototype.
Chapter 7

Related Work

In this chapter, we present the related work for this dissertation.

7.1 Relationship to Existing Models and Policy Languages

Formal semantics for policy representation and evaluation has been used by other researches, in particular Woo and Lam [77]. Their work addresses many general concerns as ours, in particular, dealing with incomplete authorizations, positive and negative authorizations and providing computable semantics. In our model, authorization is given a precise semantics independent of underlying policy requirements. This differs our work from [77] where formal notion of an authorization policy has different semantics for each set of authorization requirements.

The work by Huang and Shan [41] describes a SQL-like policy definition language. The policy enforcement process allows refining of the initial authorization request (request enhancement) and suggesting alternatives (request rewriting) if the requested resource is unavailable. These actions are performed by the policy enforcement mechanism before submitting the actual resource retrieval request to the resource manager.
This approach is different from ours in that:

1. The point of the policy enforcement is at the creation of the resource request (based on the enhancement/rewriting of initial request), which complies with existing policies. Then the resource is retrieved without any further checks. In our framework, the request is checked against the policies and is denied/granted or uncertain. No request modifications exist.

2. The approach has a limited condition representation model that does not support side effects.

The Policy Maker system described in the papers by Blaze et al. [16], [17] focuses on construction of a practical algorithm for determining trust decisions. Policies and credentials encode trust relationships among the issuing sources.

In Policy Maker's terminology, "proof of compliance question" asks if the request \( q \), supported by a set of credentials complies with a policy \( P \). This is equivalent to the authorization question that we consider in our work: "is request \( q \) authorized by the policy \( P \) (in our model the credentials are contained in the request)". Their approach, however, is different from ours.

In our approach, the information passed to the authorization engine with the authorization request is used to evaluate conditions in the relevant policy statements. The order of condition evaluation is important.

The Policy Maker system is based on the logic programming approach. The goal is to infer the desired conclusion from given assumptions in a computationally viable manner. In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in an arbitrary order and produce intermediate results that then can be fed into other assertions. Policies, representable in the Policy Maker, are restricted to the set of policies which do not produce
side-effects, resulting in change of the system state. Policy maker neither supports negative authorizations nor policy composition.

Ponder [25] is an object-oriented policy specification language that is suited for role-based access control policies, as well as general-purpose management policies. Ponder is targeted for different types of policies, including obligations, authorizations, delegation and filtering policies, and grouping these policies into aggregate structures. The obligation policies, for example, specify what actions (e.g., notification or logging) are carried out when specific events occur within the system.

To some extent, the request-result and post-conditions in our framework serve a similar purpose. However, there are several significant differences between Ponder’s and our approaches. First, in our framework the security requirements are expressed in the same policy statement, whereas in the Ponder approach authorization and obligation policies can be specified independently. This may lead to unpredictable interactions between the two policy types.

Second, the policy in our framework is enforced by the same access control mechanism. The three-phase policy enforcement model allows for parts of policy (particular conditions) to be enforced at different times. In contrast, the Ponder uses a separate enforcement mechanism for each policy type.

Finally, the Ponder obligation policies are triggered by system events whereas in our framework the actions are triggered by other conditions in the same policy statement, such as threshold or system threat level.

Security Policy Specification Language (SPSL) [24] is an attempt by IETF to define a policy specification language for specification of firewall rules, or to define allowed IPsec [46] secure connections and IKE key management. The SPSL language was designed primarily with communication security in mind and is not suited for other application domains. The policy evaluation is based on the order, the first matching policy is enforced.
Minsky and Ungureanu [54], [53] define the policy in terms of messages that only a restricted set of agents is permitted to exchange. Furthermore, the message exchange is controlled by a set of rules that is included in the policy. The policy enforcement mechanism is based on a set of trusted agents that interpret the rules and enforce them by regulating the message exchanges and the effect that the messages have on the control state (attributes and permissions) of the participating agents.

The ability to communicate and change the state resembles our concept of the read and write conditions. Our approach is different in that the “state” has a wider meaning. It includes all security-relevant information about real world that is representable in a computer system, e.g., bank account balance, temperature and user identity. Another difference is that the reading and writing of the state in our model is based on the ordered synchronous evaluation of the conditions, rather than controlled message exchange.

Jajodia et al. [43] have proposed a logical language for the specification of authorizations. The concerns addressed in this work are orthogonal to the ones in this paper. In particular, they focus on modeling integrity constraint checking and derivation rules (that derive implicit authorizations from explicit ones), while our work focuses on the representation and enforcement of authorization policies enhanced with detection and management of security violations.

Detailed formal language specification based on set and function formalism is given in the paper by Sandhu [2] for specific constraints of separation of duty in role based environment. The language semantics is defined by restricted form of the first order logic. The formal language provides a useful model to study properties of conflict of interests, in particular separation of duty.

The paper by Abadi et al. [1] presents a logical language for access control lists. They study the notions of delegation, roles and groups using their logical language and rules for making access control decisions.
Schneider's [69] paper focused on a set of policies enforceable by monitoring the system execution. Policy implementation mechanism monitors the behavior of the system. If the system is about to perform a step which contradicts the embedded policy, the system execution is terminated. This policy model is applicable primarily to the operating system and hardware-based security mechanisms. State automata abstraction is chosen for policy representation. Their implementation approach is based on modifying the system code by merging it with the policy enforcement code. While this approach allows for effective low level integration, the requirement for object level code modification of the target system, makes this approach impractical.

Bertino et al. [12] uses a directed acyclic graph of group memberships to determine authorization based on explicit positive and negative authorizations. They introduce a notion of strong and weak authorizations. The work outlines the rules for resolving authorization inconsistencies:

- Conflicting strong authorizations are not allowed.
- Strong authorizations override weak ones.
- Weak authorizations lower on a single group path override those higher up.
- Conflicting authorizations deny access.

The exploratory work by Moffet and Sloman [55] is aimed to understanding policy semantics. The two aspects of a policy are considered motivation and actual ability to carry out actions.

Summary of the research of audit-based intrusion and misuse detection is given by Lunt [51]. Sandhu and Samarati [68] discuss authentication, access control and intrusion detection technologies and suggest that combination of the techniques is necessary in order to build a secure system.
7.2 Relationship to Existing Systems

The design concepts of the KeyNote [15] system are similar to those of PolicyMaker [16], however KeyNote's features have been simplified to support efficient implementation. In particular, there is no repeated evaluation of assertions and explicit inter-assertion communication as in PolicyMaker. KeyNote also requires that credentials and policies be written in a specific assertion language. The comparison of the GAA-API and PolicyMaker features given in Section 7.2 also apply to KeyNote. The GAA-API focuses on authorization and does not address a systematic way in which to determine the rights that should be delegated. We integrated the KeyNote system with the GAA-API to support the delegation of access rights.

Hayton and colleagues [39] proposed a role-based access control system called OASIS. OASIS services specify policy for role activation using Role Definition Language (RDL) that is defined in terms of axioms in proof system. These axioms are used to prove user's eligibility to enter a set of roles.

The policy for each set of services is specified at administrative domain level, with service level agreements between domains. The role names are local to each service. A role can be specified as being permitted only for those who can prove membership of other roles issued by this and other services. The services are responsible for issuing certificates, verifying their validity and notifying other services about the certificate state changes. A policy defines a set of conditions under which a user can activate a role. Condition evaluation is achieved by presenting a corresponding certificate. The role revocation is accomplished through membership conditions. Some of the membership conditions must continue to hold while the role remains active. If any of the membership conditions associated with the activated role fails, the role is deactivated. In some sense, the OASIS membership conditions are similar to our mid-conditions that must hold during operation execution.
RDL is not as generic and expressive as our approach and not as well suited to representing complex access control policies and those that include mandatory access control.

Policies, representable in KeyNote and RDL, are restricted to the set of policies which do not produce side effects, resulting in change of the system state.

There has been work elsewhere on access control systems for Internet user agents [56], [36]. These systems apply to the JavaKey utility as an authentication mechanism and use public key digital signatures. Our system is general enough to use a variety of security mechanisms based on public or secret key cryptosystems. Also, our system is application-independent whereas the systems in [56] and [36] apply primarily for browser-like applications.

The Generalized Access Control List (GACL) framework described by Woo and Lam [78] presents a language-based approach for specifying authorization policies. The main goal of the GACL framework is merging policies associated with different objects and to resolve complex dependencies. GACL allows specification of the inheritance rules; access rights can be propagated from one object to the other. A gacl may reference other gcals in its entries. The benefit of the GACL approach is the ability to omit redundant information but it may require the retrieval and evaluation of more than one gacl. Specification of policy dependencies with inheritance is error-prone and may result in circular dependency of the policies and inconsistency may result.

More importantly, the expressive power of GACL is limited to that of ACL-based scheme. The GACL model supports limited set of conditions within which rights are granted, such as current system load and maximum number of copies of a program to be run concurrently. This may not be sufficient for distributed applications. Our system allows fine-grained control over the conditions.
Simple Distributed Security Infrastructure (SDSI) [61] has been proposed as a simpler alternative to X.509\(^1\) [21]. SDSI combines a simple public-key infrastructure that supports group definitions and group membership certificates. SDSI design relies on linked local name spaces rather than a hierarchical global name space. Though SDSI groups provide a basis for ACLs, SDSI does not address the general problem of policy specification and enforcement.

Akenti [76] is an authorization system designed to support access to distributed resources that are controlled by multiple remote stakeholders. Examples of such resources include computing and data storage systems and on-line instruments such as electron microscopes or medical diagnostic systems that have been enabled for remote operation. Akenti enables stakeholders to securely create and distribute policy statements authorizing access to the resources for which they have responsibility. Akenti is a public key based architecture that provides credential management infrastructure. Akenti uses signed certificates to express user identity, resource use-conditions, and user attributes. However, Akenti does not directly address the determination of rights to be granted.

Both restricted proxies [57] and the use-condition model [44] allow conditions and privilege attributes to be embedded in authorization credentials or certificates. These mechanisms can be readily integrated with the authorization system presented here: the restrictions or conditions carried in the proxy or certificate are evaluated by the GAA-API in addition to the restrictions in the matching EACL entry.

SESAME is a multi-domain distributed security architecture built around the use of authentication and privilege certificates [5]. Both users and applications are controlled in the same way when accessing protected resources - they must first obtain proof of their privileges in the form of a Privilege Attribute Certificate (PAC) and then present it to a target application when requesting resource access. The target application may in

\(^1\)The X.509 standard specifies a format for digital certificates, but does not define associated policy logic and decision-making aspects.
turn access another target using the delegated privileges. Access control information is represented in a generic fashion to support mapping to the different types of access controls on targeted resources. SESAME follows a delegation-only model for authorization.

The Community Access Service (CAS) [30] is a proposed Grid authorization service that a user calls prior to mailing a request for the Grid resources. CAS returns a signed capability that indicates that the authorization request is granted. The GAA-API is a core piece of the CAS.

The CRISIS architecture [11] is a security system based on public key cryptography. Types of access in CRISIS ACLs are related to the type of protected object. CRISIS ACLs do not support specification of constraints placed on resources that principals are allowed to consume. Access requests to an object are mediated by contacting the object’s reference monitor. Reference monitors are service-specific and implemented as separate modules. The emphasis of our work is on providing a general framework for representing security policies and facilitating authorization decisions for various applications. Our system provides a uniform authorization mechanism that is capable of supporting different operations and different kinds of protected objects.

The Domain and Type Enforcement (DTE) [8] is a kernel-level access control mechanism that associates a domain with each running process and a type with each object. Access tables represent allowed access modes between domains and between domains and types. At run time a DTE subsystem compares a process’s domain with the type of any object or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process’s domain does not include a right to the requested access mode for that type. DTE enforces mandatory access control through operating system modifications. The GAA-API is targeted for application-level access control. Not only can GAA-API support simple labels like domains and types, it can also provide additional features and supports fine-grained access control policies.
The Flask Security Architecture is part of the Flask [73] microkernel-based operation system. In the Flask architecture, the security policy logic is encapsulated within a separate component of the operating system, along with a general interface for obtaining security policy decisions. Flask includes a security policy server to make access control decisions and a policy-flexible enforcement framework in the microkernel and other object managers in the system. In Flask, the access decisions are based on the current system state, allowing the requested operation to proceed, denying the operation or even injecting certain operations of its own. The Flask architecture provides support for the definition and enforcement of operating-system-level security policies, whereas the GAA-API is targeted as a generic authorization tool to be used at the application level.

The Tivoli Management Environment (TME 10) [42] commercially available security system that uses a role-based approach to security. TME roles are named capabilities, containing a list of objects and access permissions to those objects. Objects can have default access and can be associated with more than one role. Each role will have a different level of access to the object. Roles are defined to support a particular job function within an organization, e.g. customer support or management. Groups are assigned roles, thus giving members of those groups access capabilities to the objects assigned to those roles. The TME approach can be easily mapped to our framework.

TME lacks flexibility in supporting user-defined security policies. It has a fixed pre-defined set of object types and generic access permissions that are available on each object type. In addition, the TME system requires creation of a new role to include each possible combination of objects and access rights. This becomes very cumbersome for systems where a large number of operations exist on various objects.
The aznAPI [6] is implemented in the Tivoli Policy Director product, which is available commercially. A typical aznAPI calling sequence includes two calls: `get_credentials`, which translates authentication data such as a certificate into authorization credentials (containing groups, roles, clearances, etc.) and `access_allowed` (which makes the decision). The aznAPI hides authentication and authorization data from applications by moving most of the processing down underneath the interface into the vendor’s implementation. The aznAPI can be layered on top of the GAA-API to hide the details of EACLs from programmers and to use access control functions at a higher level of abstraction.

### 7.3 Summary

This chapter reviewed related work on authorization and access control. We started with a discussion of the existing models and policy languages, with emphasis on similarities and differences between our model and the discussed work. The majority of models do not support policies with side-effects and have limited expressiveness. We then discussed the existing authorization systems and compared them to the GAA-API. Most of the implemented systems are domain specific (e.g., specifically designed for mandatory access control or public-key based environments), whereas the GAA-API can be used by different applications in public or secret key environments. The GAA-API is an open system that can be easily extended to support new types of policies.
Chapter 8

Conclusion

A security policy is the essential basis on which an effective and comprehensive system security can be developed. This critical component of the overall security architecture, however, is often overlooked.

In this dissertation we have argued that existing access control models are concerned with a static policy, meaning that the policy is checked only when an operation is requested to determine whether the operation should be permitted or forbidden. The active aspects of security policy, which define actions to be performed when events such as security violations are detected, are often specified as procedures to be followed by administrators or are coded into security components. A clear formal specification of security policy is needed that not only specify legitimate user privileges but also aid in the detection of the abuse of the privileges and adapt to perceived system threat conditions. Access control policies can assist in the application-based category of intrusion detection, which monitors critical applications. Traditional access control policies simply specify whether the access is granted or whether the request is denied. A new policy specification approach with intrusion detection in mind (in addition to defining actions that are and are not permitted) will identify specific application-level events that constitute malicious or suspicious activity. Furthermore, such policies will specify the countermeasures to be taken to respond to the
suspected or detected attacks. We conclude this dissertation with an emphasis on the main contributions of this work and some insights on future work.

8.1 Contributions

The contributions of this dissertation are:

- The recognition that most important, and often least understood, aspect of distributed system security is the security policy. Security policies are becoming increasingly important in modern networked computer systems as the means to manage risks such as theft, fraud and denial of service risks. To fully address these risks the policies should integrate auditing, intrusion detection, authentication and notification services. Furthermore, the policies allow more efficient utilization of these services.

- The recognition that distributed services should be more concerned with the authorization of the requested access rather than with the identity of the entity on whose behalf the request is issued. Conventionally, it has been assumed that the parties who are participating in the authorization process have already gone through an authentication phase. In this dissertation we argued that if a policy does not require authenticated user identity, authentication process can be ignored or differed until the policy explicitly requests it.

- The recognition that the points of the policy enforcement may include three time phases: before requested operation starts; during the execution of the authorized operation; and when the operation is completed.

- The presentation of the Conceptual Model, a new model for authorization that supports active policies and policy composition.
- The formal presentation of the Conceptual Model based on progression of models from simple to higher complexity provides greater comprehension of both the problem and the implemented system.

- The demonstration of the translation of security policies across multiple authorization models, such as Chinese wall, Clark-Wilson and lattice-based policies, to the canonical representation described by the formal EACL Model.

- The design and implementation of the GAA-API, a prototype based on the Conceptual Model that implements our formal EACL Model. The GAA-API supports addition of modules to support application-defined condition evaluation, policy retrieval and credential management interfaces.

- The integration of the GAA-API with different applications to validate the ideas embodied in the Conceptual Model and ensure that the API, as defined, is useful and usable.

- The application-level intrusion/misuse detection and response capabilities. Applications delegate access control and application-level intrusion/misuse detection to the GAA-API. We demonstrated how our system can be used within different intrusion attack-response scenarios.

The overall contribution of this dissertation is investigation of the issues that are raised by the support of the policies that allow active actions to be performed when security violations are detected in multi-organizational distributed systems.

### 8.2 Future Directions

The future work falls into two categories: extending the Conceptual Model to address the scalability issues; and extending the GAA-API implementation to create more usable system.
The first category, extending the Conceptual Model includes:

- Further investigation of the execution control phase of our three-phased policy enforcement framework. We need to gain more understanding of how the mid-condition concept can be utilized in real distributed systems.

- Our model allows the specification of conditions in credentials that are evaluated during the condition evaluation. This causes recursive condition evaluation process that may result in complicated and unexpected results, in particular if the conditions in credentials are write conditions. We need to investigate these issues.

- Solving problems that are raised by the totally ordered policy evaluation approach:
  1) The approach is not scalable.

  The requirement to allow only one condition evaluation process at a time results in inefficient policy evaluation process. This leads to systems that cannot scale to large numbers of objects. This approach does not allow the replication of the policy evaluation mechanism to achieve high availability and to improve performance.

  2) The current model does not lend itself to a distributed asynchronous access control architecture.

  Our current approach may be appropriate for some client-server applications, where the server is an autonomous agent, in complete charge of its resources. The server maintains the security policy and is responsible for the policy evaluation. Some distribution of the policy evaluation process can be achieved through the condition evaluation function implemented as, for example, an RPC call that is performed synchronously.

  But this approach is not suitable for the truly distributed architectures where a set of servers implement the policy and the policy evaluation processing can be distributed
over several servers. Each server is responsible for enforcing of a part of the whole access control policy.

Here are the issues that our model has to address:

- Scalability

To address the scalability issues, we have to extend our model to support:

- concurrent requests.
- replication of the evaluation mechanism.
- concurrent evaluation of conditions within the same request. This means that if a policy statement has several conditions, some of them can be evaluated in parallel (e.g., send a notification and at the same time log the information about the request). This will allow us to improve performance of the policy evaluation.

Our model has to cater for consistent updates of the system variables (used as parameters in some conditions, e.g., number of failed logins) in the presence of the concurrent requests, replicated policy evaluation mechanism or concurrent evaluation of conditions in one policy statement.

Although many conditions can be safely evaluated in parallel, the detection of possible conflicts is necessary. We need to specify rules to control which conditions can run in parallel and which can not (or maybe they can with some restrictions).

- Distributed policy enforcement

We needed to extend our model with special rules that govern the interaction between the members of a distributed community of servers involved in a policy evaluation process, along with a mechanism that provides for the explicit formulation of such rules, and for their scalable enforcement.
- Policy partitioning

We have to decide how to partition the whole policy across several access control servers and coordinate the policy evaluation processes. By the "whole" policy we mean only relevant policy statements where the target object and requested access right appear (the object and the right match the ones in the authorization request).

We have two candidate approaches to partition the whole policy.

In the first, we will consider a coarse-grained policy distribution. We will partition the policy at the granularity of a policy statement. Each server gets a policy statement (or several policy statements).

Our second approach is to perform a fine-grained policy distribution, for this we will partition the whole policy at the granularity of parts of a condition block within one policy statement. This means that a condition block in a policy statement has to be partitioned into smaller condition blocks that are then distributed among the servers.

We will investigate the advantages and disadvantages of the both approaches.

- Partial policy evaluation

The results of the evaluation of the partial policies have to be exchanged and the final decision has to be made. The policy composition approach that we are using in our model complicates distributed policy enforcement.

For our semantics, the policy composition reduces to: (i) building an ordered list of policies based on the policy priority; and (ii) computing the next composition mode based on the priority and the composition mode the current policy statement.

This approach adds a dynamic aspect to our policy evaluation algorithm. The concrete evaluation semantics can be known only at the run time. Thus the
communication of the partial policy evaluation information among the servers is very important to correctly enforce the intended policy.

- Insuring policy integrity

It is hard to ensure that an heterogeneous set of servers implement the whole policy correctly. The policy evaluation may be performed asynchronously, thus, the order in which the partial policies are evaluated is hard to assess. We will extend our model with the means to ensure the integrity of the distributed policy enforcement process.

Planned experiments with GAA-API include:

- Adding support for additional types of conditions.

- Making additional services available to the GAA-API, in particular we plan to integrate the GAA-API with an intrusion detection service (possibly Snort) and events service.

- Adding support for replication of the GAA-API modules.

- Adding support for concurrent evaluation of conditions.

- Adding support for caching of evaluated permissions. The access vector cache can be used to limit the number of application calls to the GAA-API. This vector is cached, from which the needed policy decision is obtained. Subsequent policy requests that can be serviced by the cached vector can be completed without the involvement of the GAA-API. Cache Expiration time should be maintained.

8.3 Concluding Remarks

The condition-based conceptual model described in this dissertation provides a powerful framework within which adaptive fine-grained authorization policies can be represented.
The GAA-API prototype (that implements this model) makes the framework available for supporting such policies. The contributions of the model and the prototype rest in encouraging and enabling users to define policies in ways that make it easier to interpret, compose and enforce policies in a distributed multi-policy environment. The real contribution of this work will be measured by the extent to which the model is adopted by service providers and used for access control. The early signs are positive in this regard.
Bibliography

A calculus for Access Control in Distributed Systems.

The RSL99 Language for Role-Based Separation of Duty Constraints.


http://www.apache.org

Authorization For a Large Heterogeneous Multi-Domain System.


Practical Domain and Type Enforcement for UNIX.

The CRISIS wide area security architecture.

The CRISIS wide area security architecture.

Supporting Multiple Access Control Policies in Database Systems.

The Set-Function Approach to Conceptual Modeling.

Integrity Considerations for Secure Computer Systems.

The KeyNote Trust-Management System Version 2.
http://www.crypt.com/papers/rfc2704.txt

Decentralized Trust Management.

[17] M. Blaze, J. Feigenbaum, M. Strauss,
Compliance Checking in the Policy Maker Trust Management System.

Trust Management for IPSEC.


The Chinese Wall Security Policy.

162
[21] CCITT.

Non Discretionary Controls Commercial Applications.

Protection in the Hydra Operating System.


The Ponder Policy Specification Language.

[26] DCE 1.2.2 Introduction to OSF DCE.

SPKI certificate documentation.
http://www.clark.net/pub/cme/html/spki.html

A Mathematical Introduction to Logic.

Elements of Set Theory.

Community Authorization Service.

Globus: A metacomputing infrastructure toolkit.

The Grid: Blueprint for a New Computing Infrastructure.
Morgan Kaufmann, 1999.
The SSL 3.0 Protocol.
Netscape Communications Corporation, November, 1996.

[34] S. Garfinkel, G. Spafford.
Practical UNIX & Internet Security.

http://www.globus.org

Implementing Protection Domains in the Java Development Kit 1.2.

http://www.globus.org/security/overview.html

[38] M. A. Harrison, W. L. Ruzzo and J. D. Ulman.
Protection in operating systems.

OASIS: Access Control in an Open, Distributed Environment.

An Overview of the Andrew File System.

[41] Y. Huang and M. Shan.

[42] IBM.
TME 10 security management.

A logical Language for Expressing Authorizations.

A use-condition centered approach to authenticated global capabilities: Security arch-
chitectures for large-scale distributed collaboratory environments.
LBNL Report 38850.
A linear time algorithm for deciding Security.


[47] B. Lampson
Protection.

Generic Security Service Application Program Interface.

[49] Linux FreeS/WAN.
http://www.freeswan.org

On Three-Valued Logic. 1920.

A Survey of Intrusion Detection Techniques.

[52] A. Massimo, W. Cazzola, E. B. Fernandez
A History-Dependent Access Control Mechanism Using Reflection.

[53] N. Minsky and V. Ungureanu.
Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems.

[54] N. Minsky and V. Ungureanu.
Unified Support for Heterogeneous Security Policies in Distributed Systems.

The representation of Policies as System objects.
Resource access control for Internet user agent.

[57] B.C. Neuman.
Proxy-based authorization and accounting for distributed systems.

[58] B. C. Neuman and S. Rao

[59] B.C. Neuman and T. Ts’o.
Kerberos: An authentication service for computer networks.

Three-Valued Logic and Future Contingents.

SDSI - A Simple Distributed Security Infrastructure.

A Communication Agreement Framework of Access Control.

[63] T. V. Ryutov and B. C. Neuman
Representation and Evaluation of Security policies for Distributed system Services.

The Performance of a Reliable, Request-Response Transport Protocol.

Protection of Information in Computer Systems.

Access Control: The Neglected Frontier.
Role-Based Access Control: A Multi-Dimensional View. 
*Proceedings of 10th Annual Computer Security Applications Conference*, pages 54-62, 

Authentication, Access Control, and Intrusion Detection. 

[69] F. B. Schneider. 
Enforceable security policies. 

[70] T. Sheldon. 

[71] W. Shen and P. Dewan. 
Access Control for Collaborative Environments. 

Separation of Duty in Role-Based Environments. 


[74] Sun Microsystems, Inc. 
NFS version 2 protocol specification. 

Jansen and G. van Rossum. 
Experience with Amoeba distributed operating system. 

Certificate-based Access Control for Widely Distributed Resources. 

[77] T. Y. C. Woo and S.S. Lam. 

Designing a Distributed Authorization Service. 
SSH Protocol Architecture.