The Set and Function Approach to Modeling Authorization in Distributed Systems

Tatyana Ryutov and Clifford Neuman

Information Sciences Institute University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
{tryutov, bcn}@isi.edu
(310)822-1511 (voice) (310)823-6714 (fax)

Abstract. We present a new model that provides clear and precise semantics for authorization. The semantics is independent from underling security mechanisms and is separate from implementation. The model is capable of representing existing access control mechanisms. Our approach is based on set and function formalism. We focus our attention on identifying issues and use our model as a general basis to investigate the issues.

1 Introduction

The Internet has rapidly evolved to a platform that supports business and services such as e-commerce, electronic publishing, and health care. Security compromises now have real world consequences, resulting in release of sensitive or protected information and monetary loss. Attacks on medically critical computing capabilities might even result in loss of human life. The ability to define and enforce fine-grained security policies for systems and services is important in such systems. The ability to understand such security policies is critical if they are to be correctly written or implemented. Unfortunately, as the complexity of the systems grow, these polices are becoming harder to correctly define and more difficult to enforce.

To cope with 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. For doing so, we use a methodology based on concepts of sets and functions.

In our paper we are only interested in the class of authorization policies versus a wider range of policies, such as distributed system management policies. The goal of authorization polices is to govern access to objects. Supporting such

V.I. Gorodetski et al. (Eds.): MMM-ACNS 2001, LNCS 1452, pp. 189–207, 2001.
© Springer-Verlag Berlin Heidelberg 2001

policies takes the form of monitoring and restricting the user activity within the distributed system (access control), making authorization decisions (authorization) and performing necessary actions to modify the behavior of the system (policy enforcement).

An authorization policy specifies conditions, which must be satisfied before, during or after the access right is exercised. 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 a domain B and the process does not use more then 20% of the CPU time. An audit record about the started process must be generated”.

This policy specifies several conditions:

1. *location of the requester*
   This condition must be satisfied before the access right ”process run” is granted.
2. *system load*
   This condition must hold while the process is running.
3. *audit record generation*
   This condition must be met after the process is started.

Our model captures this intuitive notion of authorization policy and provides a formalism for the policy representation and evaluation.

There has been extensive research in 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 [3], collaborative environment [17] or separation of duty [2]. Others are concerned with a particular access control mechanism, such as an ACL [1].

What is still missing, is a unified view of authorization in a distributed, multi-policy environment. Such a environment is composed of connected independent computer systems managed by separate administrative authorities. In a multi-policy environment the policy integration should incorporate diverse authorization models, which can coexist in a distributed system. Administrators of each domain might express security policies by means of different formalism.

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.

Our paper describes an authorization model designed to meet these needs. In particular, our model allows us to represent existing access control models (e.g., ACL and capability) in a uniform and consistent manner.

The model simplifies the specification of complex authorization policies and provides a generic policy evaluation environment. 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.

We keep our model simple and practical to serve as an aid to implementation. We have found that the model suggested ideas for implementations, for example that condition implementation should be based on three phases.

Our final goal is to implement a subset of our conceptual model and provide a programmable framework for different kinds of polices. The framework maps real-world policy entities such as users, resources, and organizational policies, to the representation of these entities in the programming environment. The discussion of the initial implementation can be found in [14].

2 Related work

In this section we review prior research in representation and evaluation of authorization. Formal semantics for policy representation and evaluation has been used by other researches, in particular Woo and Lam [15].

Their work addresses general concerns as ours, in particular, positive and negative authorizations and providing computable semantics. In our model, authorization is given a precise semantics independent of underlying policy requirements. This distinguishes our work from [15] where a formal notion of an authorization policy has different semantics for each set of authorization requirements.

The Policy Maker system described in the papers by Blaze, et al. [4], [5] focuses on construction of a practical algorithm for determining trust decisions. Policies and credentials encode a set of 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 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. Each condition is evaluated just one time. The order of condition evaluation is important.

In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in arbitrary order (and possibly many times) 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. The Policy Maker can be integrated in our model as a component for evaluation of the **trust constraints** conditions.

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 a 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.

The exploratory work by Moffet and Sloman [11] is aimed to understanding policy semantics. The two aspects of a policy are considered: motivation and actual ability to carry out actions.

3 Basic Conceptual Model

The conceptual model presents the high level organizing principles of the authorization model and defines the strategy chosen to realize the model.

3.1 Policy Elements

In this section we explore the notion of a policy and abstract it into a conceptual model. This section prepares us for going to the more detailed specification given in the next section. We start the design of the conceptual model with specification of the components that are to be modeled. At a conceptual level a policy is a compound entity, which regulates access to objects.

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. The notion of a negative access right 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[1]. Here are several of the more useful conditions [12].

- **access identity**
  Specifies an authenticated access identity (subject) on whose behalf request to access an object has been issued.
- **time**
  Time periods for which access is granted.
- **location**
  Location of the principal. Authorization is granted to the principals residing on specific hosts, domains, or networks.
- **payment**
  Specifies a currency and an amount that must be paid prior to accessing an object.

[1] However, if the access right is negative, the access is denied if all conditions are met.

- **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 an application level audit data in response to access requests.
- **notification**
  Enables automatic generation of notification messages in response to access requests. Specifies the notification method and a receiver.
- **trust constraints**
  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**
  Defines a set of attributes that must be possessed by subjects in order to get access to the object, e.g., user age.

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

In our model policies are treated as the first class citizens. Authentication, audit and accounting mechanisms are activated by explicit policy requirements, expressed through conditions. 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".

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

Unfortunately, side effects might complicate the model. Ignoring the side effects might cause problems when the side effects create a feedback loop, for example, 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).

Balancing the complexity this adds with the simplicity of the model is still an open issue, which requires further investigation. Initial ideas on handling the side effects are given in Section 4.2.

3.2 Basic Definitions and Assumptions

We present our conceptual model based on set and function formalism, algebra of sets and first order logic. The conceptual model specification is guided by conventional authorization notions and expected authorization requests.

An elementary policy statement consists of an object component, a positive or negative access right component and zero or more condition components. Thus,

to represent the components, we define sets of elements called objects O, positive rights R, negative rights $\overline{R}$ and conditions C. All existing policy statements are contained in the set P. In addition, we define a set of authorization requests Q.

All the sets, except for C[2] , are finite dynamic and unordered. 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. Negation is applied only to the elements of the set R to model negative rights. We do not define negative conditions. The empty set is denoted by $\emptyset$.

O is finite dynamic non-empty unordered set of object elements:

$$O = \{o_1, o_2, \ldots, o_n\} \ . \tag{1}$$

R is finite dynamic non-empty unordered set of access right elements:

$$R = \{r_1, r_2, \ldots, r_n\} \ . \tag{2}$$

$\overline{R}$ is finite dynamic non-empty unordered set of negative access right elements. Set $\overline{R}$ is constructed from the set R by applying negation to each element of the set R.

$$\overline{R} = \{\neg r_1, \neg r_2, \ldots, \neg r_n\} \ . \tag{3}$$

Note that $R \bigcap \overline{R} = \emptyset$.

C is dynamic unordered set of condition elements with a special condition element c^*, which represents an empty condition:

$$C = \{c^*, c_1, c_2, \ldots, c_n\} \ . \tag{4}$$

P is finite dynamic unordered set of compound policy elements:

$$P = \{p_1, p_2, \ldots, p_n\} \ . \tag{5}$$

Each element p of the set P represents a set of three elements:

$$p = \{o, r, c\} \, , \ o \in O, \quad r \in R \cup \overline{R}, \ c \in C \ . \tag{6}$$

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

Q is finite dynamic partially ordered set[3] of compound authorization request elements:

$$Q = \{q_1, q_2, \ldots, q_n\} \ . \tag{7}$$

Each element q of the set Q represents a set of three elements:

$$q = \{o, r, c\}, \ o \in O, \ r \in R, \ c \in C \ . \tag{8}$$

[2] The conditions can be represented by different entities, including numbers (see Section 4.2), so we can not state finiteness property.

[3] The reasoning behind the requirement of the partial ordering of the set Q is discussed in Section 4.2.

The elements correspond to the target object (o), requested access right (r) and a condition constant (c). The condition constant c represents information which is matched to the requirements specified in the condition of the relevant policy statement. In practice, this information can be represented by a set of credentials, e.g., authenticated user identity. For example, a policy statement "Anyone can read file A from 8am till 6pm" specifies a **time** condition. The request "read (r) file A (o) at 5pm (c)" specifies current time and is matched to the **time** condition in the policy statement.

To make our model practical, special provisions should be made for dealing with the following situations:

- incomplete data, not known at the authorization time. During network fragmentation some data may be inaccessible.
- policy requires a certain event to happen in the future. Statements about the future do not have truth values until the event described takes place.
- the function used to evaluate conditions does not terminate for the arguments supplied. Incorrect implementation, bad parameters.

In order to properly deal with these situations we will adopt a three-valued logic [9], [13].

Three-valued logic is classical boolean (true/false) logic extended with a third truth value - undefined.

We define an auxiliary set B, consisting of the three constants: true, represented by T, false, represented by F and U, meaning uncertainty.

$$B = \{T, F, U\} . \tag{9}$$

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

P	Q	P & Q	P ∨ Q
T	U	U	T
U	T	U	T
F	U	F	U
U	F	F	U
U	U	U	U

Table 1.

In addition, $\neg U = U$. Next we define functions to express an authorization process.

The *by_object* function takes a set of policy elements P and request q, which contains particular object $\widehat{o}$ as an argument and returns a subset $P' \subseteq P$ where this object appears.

$$P' = by_object(P, q),$$

$$\widehat{o} \in O,\ \widehat{o} \in q,\ q = \{\widehat{o}, r, c\},\ \ q \in Q,\ P' \subseteq P : \forall p' \in P' : p' = \{\hat{o}, r, c\} . \tag{10}$$

The *by_right* function takes a set of policy elements P and request q, which contains particular access right $\widehat{r}$ as an argument and returns a subset $P' \subseteq P$ where this right appears.

$$P' = by_right(P, \widehat{r}),\ \widehat{r} \in R,$$

$$P' \subseteq P : \forall p' \in P' : p' = \{o, \widehat{r}, c\}\ or\ p' = \{o, \neg\widehat{r}, c\}\ . \tag{11}$$

The *eval_cond* is a condition evaluation function.

$$b = eval_cond(\widehat{c}, \widetilde{c}),\ \widehat{c} \in C,\ \widetilde{c} \in C,\ b \in B\ . \tag{12}$$

The function M defines positive or negative modality of the policy element. If the access right, contained in the policy element is positive or negative, the modality is positive or negative, respectively.

$$M(p_i, q) = \begin{cases} eval_cond(\widehat{c}, \widetilde{c}), & \widehat{r} \in R \\ \neg eval_cond(\widehat{c}, \widetilde{c}), & \widehat{r} \in \overline{R}, \end{cases}$$

$$\widehat{c} \in p_i,\ \widetilde{c} \in q,\ \widehat{r} \in p_i,\ p_i \in P',\ q \in Q\ . \tag{13}$$

The M function has to be applied to all elements $P' \subseteq P$. The evaluated modality of each policy element will be taken with or without the negation $\neg$ according to its right. After all the modalities are evaluated, we will take their disjunction. These operations are performed by the *eval_conditions* function.

$$b = eval_conditiods(P', q) = M(p_1, q) \bigvee M(p_2, q) \bigvee ... \bigvee M(p_n, q),$$

$$p_i \in P',\ i = \overline{1, n},\ n\ is\ the\ cardinality\ of\ P',$$

$$P' \subseteq P,\ q \in Q,\ b \in B\ . \tag{14}$$

The resulting value b obeys to the $\bigvee$ operation for three-valued logic. That is, *eval_conditions* returns T if at least one modality gave the result T, F if all results were F, and U otherwise (i.e., at least one result was U, possible some F but none T).

The *authorization* is a composite function:

$$b\ =\ authorization(P, q)\ =$$

$$=\ eval_conditions(P''', q)\ \circ\ by_right(P', q)\ \circ\ by_object(P, q)\ =$$

$$=\ eval_conditions(P''', q)\ \circ\ by_object(P', q)\ \circ\ by_right(P, q)\ . \tag{15}$$

The *authorization* function takes the set of policies P and an authorization request q as arguments. It returns F, T or U meaning authorized, not authorized or uncertain. Three-valued logic at the conceptual level has to be mapped to the two-valued logic at the implementation level. In the end, the access must be either granted or denied.

3.3 Time Dependency

Time dependency appears in our conceptual model implicitly. At each instant only the 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 one can not send" [16]. Some policies may need to know precise time of the event occurrence, for example for audit purposes. This may require a time-stamping of certain occurrences and keeping record of them.

3.4 Changes in the Set Membership

Exercising access rights can result in creating new objects and defining new policies. In the conceptual schema this is represented as adding an element to the corresponding set. As we discussed in the previous section, changes in membership of the sets R and $\overline{R}$ depend entirely on the set O.

The deletion of an element from the sets O, R or $\overline{R}$ entails deletion of each element from P in which the deleted element appears. To simplify our model we require that rights can be applied only to the elements of set O. If we allow rights to be applied to the elements of P, we will have to consider a policy management model.

3.5 Policy Representation Issues

We do not allow use of the disjunction in representation of elements of the set P. The disjunctive form policies such as "Tom or Joe can read file A", "Tom can read either file A or B" and "Tom can either read or write file A" is modeled by using separate policy statements.

$$O = \{A, B\}\,, R = \{read, write\}, C = \{c^*, Tom, Joe\},$$

$$P = \{\{A, read, Tom\}, \{A, read, Joe\}, \{B, read, Tom\}, \{A, write, Tom\}\}\ .$$

However, disjunction of policy elements can be used in practice for optimization reasons. For example, in the implementation of an ACL we can combine several access rights which correspond to a particular access identity condition.

Let us consider the exclusive OR policy representation: "Tom can read files A or B, but not both". This policy is a variant of the Chinese wall policy [6], required in the operation of many financial services. The policy guards against the conflict of interest. A consultant can freely chose a company in order to offer an advice. However, once the company has been chosen, the consultant is mandatory denied access to the information about all other companies. This policy can

be implemented using an additional condition, let us call it *trigger_history*. This condition activates the history of execution.

$$P = \{\{A, read, Tom, trigger_history\}, \{B, read, Tom, trigger_history\}\} .$$

If Tom decides to read file A first, the history is checked, and since initially it is empty, the right is granted and the information about it is stored. If he tries to read file B after that, the request will be denied. A history information is maintained by the system. The history can be centralized or distributed. An example of implementation of the condition is briefly described in [18]. More detailed discussion of implementation of the history-dependent access control policies is given in [10].

In conventional access control models, a subject has been a separate notion. A subject is an entity on whose behalf a request to access an object has been issued. 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 and capability, have become practical implementations of the Lampson matrix [8].

In the ACL based systems, 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.

In the capability-based systems, policies are grouped by subjects. A capability lists sets of objects accessible by the subject along with the types of access rights.

These logical grouping can be represented in our model.

ACL An ACL consists of a set of ACL entries. An ACL entry is analogous to a policy element p, where all conditions are **access identity**.

Consider a policy: “Tom and Bob can read and write file A”. We can translate this policy into our policy model as:
“Tom (condition c_1) and Bob (condition c_2) can read (positive right r_1) and write (positive right r_2) file A (object o_1) “. We need four policy elements to represent this policy:

$$p_1 = \{o_1, r_1, c_1\},$$
$$p_2 = \{o_1, r_2, c_1\},$$
$$p_3 = \{o_1, r_1, c_2\},$$
$$p_4 = \{o_1, r_2, c_2\} .$$

This way of specification and storage of the policy is tedious and inefficient.

To represent an ACL, we adopt three modifications to the representation of a policy element p specified in(6):

1. An ACL is associated with each object, so the object is implicit and is omitted from the policy elements.
2. Conditions are listed first, then access rights. This order is closer to the traditional ACL specification.
3. We allow disjunction of either positive or negative access rights.

Now we need only two ACL entries to represent the policy:

$$p_1 = \{c_1, r_1 \vee r_2\},$$
$$p_2 = \{c_2, r_1 \vee r_2\} .$$

Furthermore, if we allow conditions to be aggregated into a single entry when the same set of access rights applies to all of them, we need only one policy statement to represent the policy: $p_1 = \{c_1 \vee c_2, r_1 \vee r_2\}$.

by_object function returns all policy statements associated with the given object. The returned set of policies P' conceptually represents an ACL associated with the object $\widehat{o}$.

Capability To demonstrate how capabilities can be represented, we define function *by_condition*, which takes the set of policies P and particular condition $\widehat{c}$ as arguments and returns a subset P', where this condition appears. Intuitively, this function returns all policy statements associated with the given condition.

$$P' = by_condition(P, \widehat{c}),\ \widehat{c} \in C,\ P' \subseteq P : \forall p' \in P' : p' = \{o, r, \widehat{c}\} .$$

Note that if the condition constant $\widehat{c}$ specifies particular access identity (subject), then the returned set of policies P' conceptually represents a capability possessed by the subject identified by the condition $\widehat{c}$. Next the set P' can be passed to the *authorization* function along with an authorization request for further evaluation.

Representation of a capability is quite similar to that of an ACL. A capability is associated with each subject, so the subject is implicit and is omitted from the policy element. Thus, each policy statement contains only elements, which represent objects and access rights.

More detailed discussion of the implementation of ACL and capability can be found in [14].

4 Extended Conceptual Model

The extended conceptual model expands upon basic conceptual model entities and interactions. The notion of a policy hierarchy is introduced. The design work at this level addresses condition side-effects issues.

4.1 Refinements

In this section we describe further refinements of our basic entities. 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 (6) we have considered only one condition in the policy statement. All existing conditions were aggregated into one set (4). 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 S condition types exist. We represent these different condition types by S disjoint sets:

$$C = \bigcup^{k=\overline{1,S}} C^k,\ C^i \bigcap^{i,\,j=\overline{1,S}\, i\neq j} C^j = \emptyset\ . \tag{16}$$

Now we define a totally ordered[4] set $\widetilde{C}$. Each element of this set is constructed from one element of the S disjunctive sets. Intuitively this means that each element of $\widetilde{C}$ consists of S condition elements of different types, some of the elements can be c^*.

$$\widetilde{C}\ =\ \{\widetilde{c}^1, \widetilde{c}^2, ..., \widetilde{c}^S\},\ \widetilde{c}^i \in C^i,\ i = \overline{1,S}\ . \tag{17}$$

We define S condition evaluation functions for each condition type. In our policy example we define two function for checking access identity and current time.

$$b\ =\ eval_i(\widetilde{c}^i, \widetilde{c}^i),\ \widetilde{c}^i \in C^i,\ \widetilde{c}^i \in C^i,\ b \in B,\ eval_i(c^*, c^*) = T,\ i = \overline{1,S}\ . \tag{18}$$

From (4.2) and (15) we observe that if at least one of the policy statements evaluates to T, the authorization will be granted. This behavior may not be always desirable. For example, we would want a policy assigned by the system administrator to take precedence over the one assigned by an individual user. This requires the means of specifying a hierarchical relationship among policy statements.

The hierarchy of policies is modeled by assigning priorities. We do not attempt to give a full theoretical development of the method of assigning priorities here. The essential requirements is that one should be able to decompose the whole policy into totally ordered policy statements. To express policy priorities, we define set W. W is a finite totally ordered set of elements that can be compared (e.g., integers).

$$W = \{w_1, w_2, \ldots, w_n\}\ ,\ \ w_i < w_j,\ i, j = \overline{1,L},\ i < j,\ L\ is\ the\ cardinality\ of\ W\ .$$

We redefine element q, given in (8) in the following way:

$$q = \{o, r, \widetilde{c}^1, \widetilde{c}^2, ..., \widetilde{c}^S\},\ o \in O,\ r \in R,\ \widetilde{c}^i \in C^i,\ i = \overline{1,S},\ q \in Q\ . \tag{19}$$

We extend (6) in two ways: 1) each element p has an additional component w, which denotes priority of this element. 2) condition component is represented by a set of $\subseteq$ condition constants of different types.

$$p = \left\{o, r, \widetilde{C}', w\right\},\ o \in O,\ r \in R \bigcup \overline{R},\ \widetilde{C}' \subseteq \widetilde{C},\ w \in W,\ p \in P\ . \tag{20}$$

Figure 1 illustrates representation of a policy element p.

[4] The reasoning behind the requirement of the total ordering of the set $\widetilde{C}$ is discussed in Section 4.2.

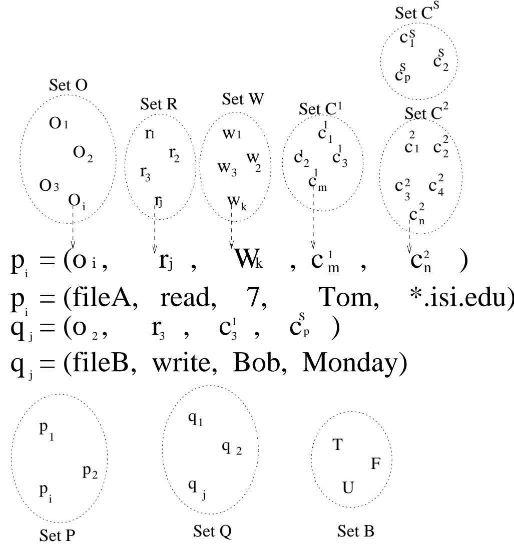

Figure 1.

The *by_priority* function takes a set of policies P as an argument and returns a subset P' with the maximum priority. The ordering in the set W determines the policy statement which is enforced if several policy statements are simultaneously satisfied. Note that if the set P' contains more then one element, the elements have equal priorities. In this case, if any of the policy statements is satisfied, authorization is granted.

$$P' = by_priority(P),$$

$$P' \subseteq P : \forall p' \in P',\ p' = \{o, r, \widetilde{C}, \widehat{w}\},\ \widehat{w} = \max(\forall w : p = \{o, r, \widetilde{C}, w\} \in P') \ . \tag{21}$$

We redefine *eval_cond* function given in (12) in the following way:

$$eval_cond \ = \ eval_1(\widehat{c}^1, \widetilde{c}^1)\&eval_2(\widehat{c}^2, \widetilde{c}^2)\&...\&eval_S(\widehat{c}^s, \widetilde{c}^s),$$

$$\widehat{c}^i \in C^i,\ \widetilde{c}^i \in C^i,\ i = \overline{1, S},$$

$$b = eval_cond(p),\ p \in P,\ b \in B\ . \tag{22}$$

The *eval_cond* function is a short hand notation for representation of conjunction of the results, obtained by applying $eval_i$ to corresponding condition constants from the policy element p. All conditions must be met simultaneously in order to satisfy the authorization request.

The resulting value b obeys to the & operation for three-valued logic. That is, *eval_cond* returns T if all elements gave the result T, F if at least one result was F, and U otherwise (i.e. at least one result was U, possible some T but none F.

We redefine *authorization* function given in (15) in the following way:

$$b \;=\; authorization(P, q) \;=$$

$$= \; eval_conditions(P''', q) \circ by_priority(P'') \circ by_right(P', q) \circ by_object(P, q) \;=$$

$$= \; eval_conditions(P''', q) \circ by_priority(P'') \circ by_object(P', q) \circ by_right(P, q),$$

$$P''' \subseteq P'' \subseteq P' \subseteq P, \; q \in Q, \; b \in B \; . \tag{23}$$

Figure 2 illustrates the *authorization* function.

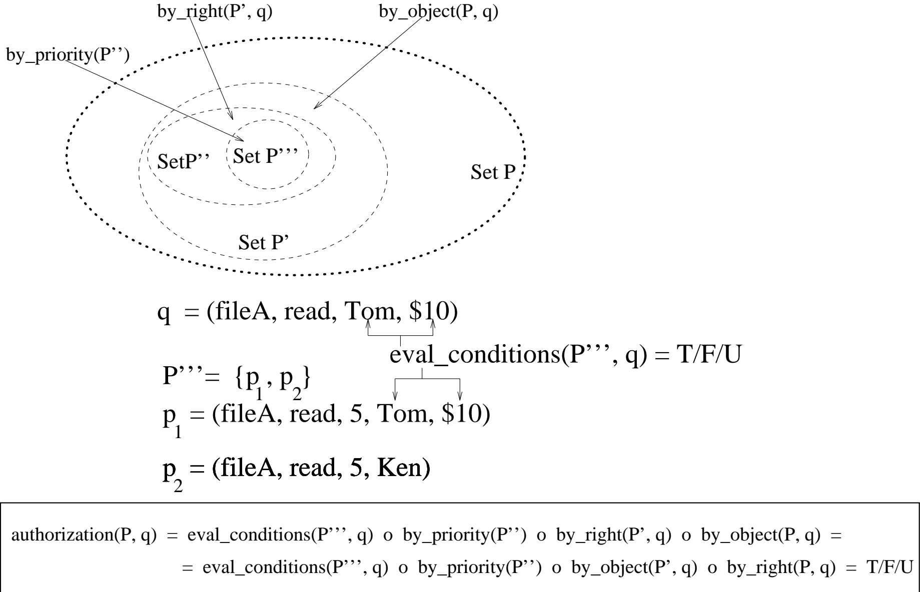

Figure 2.

4.2 Discussion of Condition Side-Effects

The total order property of the set $\widetilde{C}$ defined in (16) requires that policy elements that differ only by the order of condition elements are considered to be distinct. This property is important to deal with possible side effects caused by the condition evaluation. Consider a policy "Tom can read file A only if notification is sent (**notification** condition) and system threat condition is low (*threat_level_low* condition)". Assume that current system threat level is low. Assume that the notification about Tom reading file A triggers high system threat level. There are two ways to represent the policy in our model:

$$p_1 = \{A, read, Tom, threat_level_low, notification\},$$
$$p_2 = \{A, read, Tom, notification, threat_level_low\} \; .$$

The evaluation of p_1 results in access grant, however evaluation of p_2 results in denial.

In this section we will discuss determining the correct order of the condition elements in the policy statement p defined in (20).

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 is related to a condition evaluation process.

To simplify our presentation, we assume that dependent authorization requests do not overlap. The effects of the dependent requests are resolved by serialization, in which the 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 illustrates our representation.

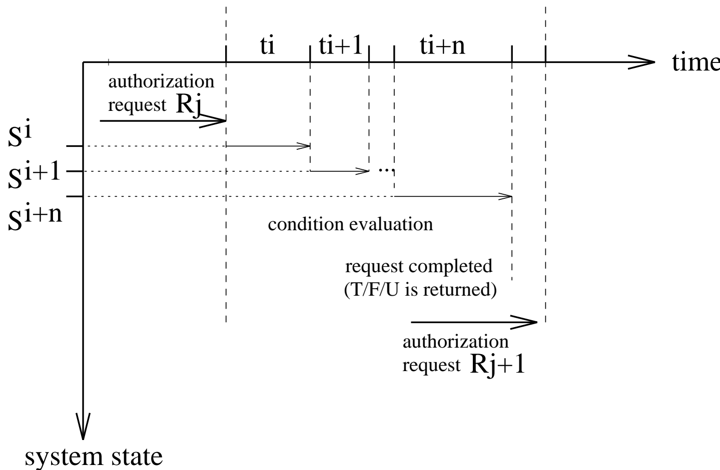

Figure 3.

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.

The general idea underlying our approach is that the system state can be formalized by a sequence of system states S^1, S^2, ..., S^k. Each system state S^i is labeled by the time interval i.

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 information about the real world which is representable in a computer system, for example: bank account balance, temperature, user identity.

Here any system state S^i is all the information that has been deduced up to the time interval i. The information is represented by a set of system variables. The information is partial, since some system variables can be undefined at some time intervals.

At each time interval i there is a transition $S^i \rightarrow S^{i+1}$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. All side effects of condition evaluation are recorded in the corresponding system variables.

Classification of Conditions In this section we present a taxonomy of conditions. We say that a condition **writes system state**, if the condition evaluation function changes values of some system variables.

The fact that evaluation of condition c changes value of the system variable j is represented by the notation $c(S^i) \rightarrow S_j^{i+1}$.

We say that a condition **reads system state** if the condition evaluation function requires reading of particular system variables.

The fact that evaluation of condition c requires the value of the system variable j is represented by the notation $c(S_j^i) \rightarrow S^{i+1}$.

We say that a condition $\widehat{c}$ **depends on condition** $\widetilde{c}$, if condition $\widehat{c}$ requires reading of some system variables, which are written by the condition $\widetilde{c}$.

The fact that condition $\widehat{c}$ requires the value of the system variable j, which is written by the condition $\widetilde{c}$ is represented by the notation: $\widetilde{c}(S_j^i) \hookrightarrow \widehat{c}(S_j^{i+1})$.

Conditions are classified by the read/write system state property:

- **read conditions** read system state but do not write system state, for example **time**, **location** and **system load**.
- **write conditions** write system state and may read system state, for example, **payment**. Payment requires checking for the presence of required amount (read system variable k) and reducing the balance by the requested amount (write system variable k). This is represented as: $c(S_k^i) \rightarrow S_k^{i+1}$.

Note that **write conditions** must be evaluated before the **read conditions** that are dependent on them.

Designing the condition ordering algorithm that satisfies the ordering requirements falls into the realm of scheduling of processes with precedence constraints and is outside of the scope of this paper.

Condition Representation and Evaluation Read conditions such as **access identity** and **location** appearing in the authorization request, specify a set of constants which must be matched against a corresponding set of constants found

in the policy elements. These conditions are represented by a set C^1. This set is constructed from a set of all condition constants passed in authorization requests q defined in (19), a set of all condition constants contained in policy elements p defined in (20) and a set of operations M. Condition evaluation function for this type of conditions returns T if applying operation m, $(m \in M)$ to the condition constants evaluates to T, otherwise it returns F.

For example, a set of operations M may contain $(\subseteq)$. If $m = \subseteq$, condition evaluation function returns T if $\widehat{c}^1 \subseteq \widetilde{c}^1, (\widehat{c}^1 \in C^1, \widetilde{c}^1 \in C^1)$, otherwise it returns F.

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
$M = \{ = , \neq , < , > , \leq, \geq \}$.

Write conditions, such as **notification** and **audit** specify the name of a system variable, whose value must be changed, and the new value. Condition evaluation function for these conditions returns T if the updating of the system variable succeeded[5], and F otherwise.

Unfortunately not all conditions can be represented in this way. In practice, conditions can be application-specific and complex. The problem is how an informal specification of the condition can be transformed into a precise formal mathematical structure, within which we can actually prove things about the properties, such as computability and polynomial-time decidability.

5 Conclusions and Future Work

In this paper we presented a conceptual model for authorization in distributed systems. We introduced precise semantics for policy representation and evaluation. The semantics is defined independently from underling security mechanisms and is separate from implementation. The flexibility of the model makes it possible to represent existing access control mechanisms.

We believe that the model provides an effective way to understand and employ authorization policies in distributed systems.

We have begun to investigate the side-effects of the condition evaluation. Through the use of the side effects, in our current work we consider integrating intrusion and misuse detection systems with applications using our model.

We hope that this model will lead to other insights about authorization policies. We are looking for possible ways to restrict condition expressiveness to guarantee policy computability and polynomial-time decidability.

[5] Updating the system variable can fail due to various reasons, for example we might be unable to append audit information to the audit log because the disc space has been exceeded.

6 Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-0595, Security Infrastructure for Large Distributed Systems (SILDS) Project under agreement number DABT63-94-C-0034 and by Xerox Corporation, XAUTH Project under agreement number HE1254-97. The U.S. Government is authorized to reproduce and distribute repreints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained in this document are those of the authors 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, the U.S. Government, or Xerox Corporation.

References

1. Abadi, M., Burrows, M., Lampson, B. and Plotkin, G.: A calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems, Vol. 15, No 4* (September 1993) 706–734
2. Gail-Joon Ahn and Sandhu, R.: The RSL99 Language for Role-Based Separation of Duty Constraints. *ACM Workshop on Role-Based Access Control* (1999) 43–54
3. Bertino, E. and Jajodia, S.: Supporting Multiple Access Control Policies in Database Systems. *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (1996)
4. Blaze, M., Feigenbaum, J. and Lacy, J.: Decentralized Trust Management. *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Press, Los Angeles (1996) 164–173
5. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance Checking in the Policy Maker Trust Management System. *In Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science*, Vol. 1465 254–274
6. Brewer, D.F.C. and Nash, M.J.: The Chinese Wall Security Policy. *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages (1989) 206–214
7. Jajodia, S., Samarati, P. and Subrahmanian, V.S.: A logical Language for Expressing Authorizations. *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997)
8. Lampson, B.: Protection. *ACM Operation System review 8(1)* (January 1974) 18–24
9. Lukasiewicz, J.: On Three-Valued Logic. 1920. *Ruch Filozoficzny* 1920, 5, pp.170-1. *English translation in Borkowski, L. (ed.) Jan Lukasiewicz: Selected Works.* Amsterdam: North Holland (1970)
10. Massimo, A., Cazzola, W., Fernandez, E.B.: A History-Dependent Access Control Mechanism Using Reflection *Proceedings of 5th ECOOP Workshop on Mobile Object Systems (EWMOS'99)*, (June 1999)
11. Moffet, J.D. and Sloman, M.S.: The representation of Policies as System objects. *Proceedings of the ACM Conference on Organizational Computing Systems*, Atlanta, GA (November 1991) 171–184

12. Neuman, B.C.: Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh* (May 1993)
13. Prior, A.N.: Three-Valued Logic and Future Contingents. *Philosophical Quarterly*. Vol. 3 (1953) 17–26
14. Ryutov, T.V. and Neuman, B.C.: Representation and Evaluation of Security policies for Distributed system Services. *In Proceedings of the DARPA Information Survivability Conference and Exposition*. Hilton Head, South Carolina (January 2000)
15. Woo, T.Y.C. and Lam, S.S.: Authorization in distributed systems: a new approach. *Journal of Computer Security, 2* (1993) 107–136
16. Schneider, F.B.: Enforceable security policies. *Technical report TR98 1664*, Cornell University (January 1998)
17. Shen, W. and Dewan, P.: Access Control for Collaborative Environments. *Proceedings of CSCW* (November, 1992) 51–58
18. Simon, R.T. and Zurko, M.E.: Separation of Duty in Role-Based Environments *Computer Security Foundations Workshop* (June 1997)