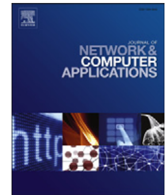
Journal of Network and Computer Applications 34 (2011) 1342–1352

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca



Applying a usage control model in an operating system kernel

Rafael Teigão, Carlos Maziero *, Altair Santin

Graduate Program in Computer Science, Pontifical Catholic University of Paraná State, Rua Imaculada Conceição 1155, 80.215-901 Curitiba, PR, Brazil

ARTICLE INFO

Article history:
Received 4 December 2009
Received in revised form
24 November 2010
Accepted 10 March 2011
Available online 16 March 2011

Keywords:
Access control
Usage control
Kernel services

ABSTRACT

Operating systems traditionally use access control mechanisms to manage access to system resources like files, network connections, and memory areas. However, classic access control models are not suitable for regulating access to the diversity of ways data is available and used today. Modern usage control models go beyond traditional access control, addressing its limitations related to attribute mutability and continuous usage permission validation. The recently proposed $\mathrm{UCON}_{\mathrm{ABC}}$ model establishes a predicate-based framework to satisfy the new access/usage control needs in computing systems. This paper defines a usage control model based on $\mathrm{UCON}_{\mathrm{ABC}}$ and describes a framework to implement it in an operating system kernel, on top of the existing DAC mechanism. A language for representing usage control entities and rules is also proposed, and some typical access/usage control scenarios are represented using it, to show its usefulness. Finally, a prototype of the proposed framework was built in an operating system kernel, to control the usage of local files. The prototype evaluation shows that the proposed model is feasible, straightforward, and may serve as a basis for more complex usage control frameworks.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

Classic access control models are not suitable for regulating access to the diversity of ways digital content is available and used today. For instance, *Digital Rights Management* (DRM) requires control that goes beyond the simple one-step access granting. This is also true for the manipulation of related data collected from several independent sources, such as medical information about patients in a hospital. Current electronic commerce of digital items brings with it the necessity of checking whether some requirements have been met, like accepting an end-user license agreement (EULA), or enforcing time restrictions in a commercial transaction. Although most of these controls are employed at the application level, they would be easier to deploy and harder to circumvent if more sophisticated access control mechanisms were made available by the underlying operating system.

The formal concept of *usage control* (UCON), presented by Park and Sandhu (2003), introduces the evaluation of attributes and requirements during the use of a resource (e.g., permission of a user to continue to watch a movie). It also considers the *mutability* of such attributes as a consequence of actions by users. Furthermore, the usage control concept includes the notion of dependency of the access policies on *external information*, like the time of day or the system load, which was not explicit in previous access control models. The $\mathrm{UCON}_{\mathrm{ABC}}$ model (Park and Sandhu, 2004) formalizes such concept.

This paper proposes a usage control model derived from the $\mathrm{UCON}_{\mathrm{ABC}}$ model. It considers the formal UCON specification defined in Zhang et al. (2004), adapting it to be implemented in an operating system context. From the proposed model, a language to describe usage control policies on system objects is defined, and its expressiveness is evaluated through a series of typical usage control scenarios. It also describes a prototype implementation for the proposed model, which was built in an operating system kernel to mediate operations on files. The prototype evaluation shows that the model is feasible and may serve as a basis for more complex usage control frameworks. This paper is an extended version of a previous work (Teigao et al., 2007), including a formal presentation of the usage control model, more details about its framework, the grammar specification, more usage examples, and the description/evaluation of an implementation prototype.

The paper is organized as follows. Section 2 describes the main features of usage control and the $\mathrm{UCON}_{\mathrm{ABC}}$ model. Section 3 explains the usage control model adopted in this paper. Section 4 presents the framework which implements the proposed model. Section 5 details the language proposed for representing usage control policies. Examples of the language representing common usage and access control scenarios are given in Section 6. Section 7 gives some details of the implemented prototype and its evaluation. Section 8 discusses related work. Finally, Section 9 gives some conclusions and presents future research directions.

* Corresponding author. Tel.: +55 41 3271 1669; fax: +55 41 3271 2121.
E-mail addresses: rafael.teigao@tjpr.jus.br (R. Teigão), maziero@ppgia.pucpr.br (C. Maziero), santin@ppgia.pucpr.br (A. Santin).

1084-8045/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2011.03.019

2. Usage control

Operating systems traditionally use standard access control models, which are used to define how processes can access files and inter-process communication channels like pipes, shared-memory areas, and semaphores. However, such models are frequently not flexible enough to represent specific usage situations, like processor, memory, and disk space usage, and time-dependent access/usage control. This problem motivated the proliferation of ad hoc solutions for specific control needs, like disk/memory/processor quotas, access restrictions based on network addresses, time, system load, and so on. Furthermore, applications and services that require a more sophisticated access/usage control policy should implement it by themselves. Thus, it would be useful to provide a means to generalize such controls in a more homogeneous model, and it should be available at the operating system kernel interface.

Although ad hoc usage control mechanisms are relatively old, formal models for usage control have been introduced recently (Park and Sandhu, 2003). A well-known usage control model, $UCON_{ABC}$ (Park and Sandhu, 2004), is based on the concepts of *authorization* (A), *obligations* (B), and *conditions* (C). In this model, subjects and objects are tagged with attributes that may be used in access decisions. Access and usage control decisions are based on policy rules stated over such user and object attributes, obligations, and conditions. The main decision elements offered by the $UCON_{ABC}$ model are:

- *Authorizations*: Functional predicates evaluated to decide if a user can exercise rights on an object. Such predicates encompass the traditional access control models, like DAC, MAC, and RBAC.
- *Obligations*: Actions that should be performed by a user prior or during the use of an object, to use it. For instance, a user should accept an end-user license agreement prior to running a new software.
- *Conditions*: Restrictions that consider system or environmental factors independent of users, objects or the decision mechanism. Conditions are considered external, independent information, like the time of the day or the processor load.

Predicates must be evaluated by a reference monitor (Sandhu and Samarati, 1994) to decide if a usage right of a subject over an object should be granted, maintained, or denied. A set of predicates must be evaluated before a subject accesses an object, namely *preA*, *preB*, and *preC*, respectively for authorization, obligation, and condition predicates. Similarly, the decision about maintaining a usage right is taken during the object usage, which defines *onA*, *onB*, and *onC* predicates. Finally, actions that should be performed when a usage finishes may be defined through *posA*, *posB*, and *posC* predicates.

The $UCON_{ABC}$ model also introduces attribute mutability, which brings with it the possibility to influence current or future actions based on usage history. For instance, it is possible to take usage decisions based on attributes that are modified at each access. A simple example of such scenario is a credit-based system: with each access to a given object, the user's credit decreases; when she runs out of credit, her access to it may be denied. It is important to notice that the value of conditions cannot be updated by the reference monitor, since conditions are external information and the reference monitor has no control over them.

The $UCON_{ABC}$ model allows to represent uniformly several common ad hoc controls and encompasses a wide range of previous models, from conventional discretionary access control to various types of mandatory and role-based models. The next section presents a usage control model, derived from $UCON_{ABC}$, whose goal is to integrate usage control within an operating system kernel.

3. The proposed usage control model

Let us consider a system with a set of users $\mathbb{U}$ and a set of objects $\mathbb{O}$. Each object $o \in \mathbb{O}$ is associated with a set of rights $\mathbb{R}(o)$ like *read*, *write*, *remove*, and so on, which can be either fully or partially granted to users. The precise set of rights associated to an object depends, of course, on its nature.

Users access objects through *usage sessions*; a usage session captures the relationship between a user and an object during a certain period of time (Katt et al., 2008). A usage session starts when an access request from a user to an object is granted, and ends when the user releases the object, either explicitly or due to a usage policy decision. Formally, a usage session s is a triplet $s(u,o,\mathbb{G})$, in which u is a user, o is an object, and $\mathbb{G} \subseteq \mathbb{R}(o)$ is a set of rights on the object o granted to user u when the session is initiated. The set of all active usage sessions is defined as $\mathbb{S}$; being it a set, a given user can have at most one active session on each object: $|\{s(u,o,*) \in \mathbb{S}\}| \leq 1,\ \forall (u,o) \in \mathbb{U} \times \mathbb{O}$.

The usage session concept expresses only the usage relationship between a user and an object; it is orthogonal to any object sharing semantics. Furthermore, the precise semantics of a usage session depends on the kind of object being used. For instance, a file usage session starts when the file is opened and finishes when it is closed. Thus, all read and write operations on such file should be performed during a usage session.

Each user $u \in \mathbb{U}$ is associated with a (possibly empty) set of attributes $\mathbb{A}(u)$, whose initial values are given by $\mathbb{A}_0(u)$. Similarly, each object $o \in \mathbb{O}$ is associated with a (possibly empty) set of attributes $\mathbb{A}(o)$, with initial values $\mathbb{A}_0(o)$. Users' and objects' attributes are defined and manipulated according to the usage policy being enforced.

Additionally, users may have to fulfill *obligations* when accessing or using objects. An obligation $b \in \mathbb{B}$ is an external requirement that should be fulfilled by a user to start or to maintain a usage session. Obligations are seen here as information external to the decision system that are not directly produced by the user, but depend on her actions. For instance, the user may be asked to accept a license agreement before using a software, or should keep an advertisement window open while accessing a free service. The checking of obligations fulfillment and its feeding to the reference monitor are discussed in Section 4.

There is also a set of environmental conditions $\mathbb{C}$. Each condition $c \in \mathbb{C}$ represents a system/environment information accessible to the reference monitor, like the time of the day, system load, processor temperature, free disk space, and so on. The value of a condition depends uniquely on factors external to the users and to the reference monitor, and cannot be directly modified by them.

Finally, each object is associated with three finite lists of predicates (possibly empty) that are used in the reference monitor decisions. In each list, a single predicate may be either a *boolean expression* or an *attribute update*. A boolean expression may involve user attributes $\mathbb{A}(u)$, object attributes $\mathbb{A}(o)$, requested/granted rights $\mathbb{G}$, obligations $\mathbb{B}$, and conditions $\mathbb{C}$, and evaluates to either *true* or *false*. By contrast, an attribute update sets the current value of a single user or object attribute and always evaluates to *true*. The predicate lists associated with each object are:

- $Pre(o)$: list of predicates $[pre_0,\ pre_1,\ \ldots]$ to be evaluated for deciding whether a request for a usage session on object o

should be granted or denied. A user u can open a usage session on object o with a given set of granted rights $\mathbb{G}$ either if the list $Pre(o)$ is empty (meaning that there are no pre-conditions for using that object) or if all the predicates in the list are satisfied for u and $\mathbb{G}$:

$$\text{can_open}(u,o,\mathbb{G}) \leftarrow (Pre(o)=\emptyset) \vee \left(\bigwedge_{\forall i} pre_i(o) \right)$$

- $On(o)$: list of predicates $[on_0, on_1, \ldots]$ to be continuously evaluated during a usage session. A user u can continue to exercise her rights $\mathbb{G}$ on an object o either if the list $On(o)$ is empty or if all the predicates in the list are satisfied:

$$\text{can_use}(u,o,\mathbb{G}) \leftarrow (On(o)=\emptyset) \vee \left(\bigwedge_{\forall i} on_i(o) \right)$$

Obviously, a fully continuous usage evaluation is generally not possible in a real system, so it should be replaced by periodic checks. The granularity of this discretized evaluation is to be defined for each kind of object being used, in order to produce semantically meaningful results while not overloading the reference monitor.

- $Pos(o)$: list of predicates $[pos_0, pos_1, \ldots]$ to be evaluated when a usage session terminates, either because the user closes it or the reference monitor revokes the session due to a usage policy decision. Since the session closing has no influence on the session itself, only attribute update predicates are allowed here. All the predicates in the list $Pos(o)$ should be evaluated because the attribute updates they perform may have effects on other (concurrent or future) usage sessions:

$$\text{on_close}(u,o) \leftarrow \bigwedge_{\forall i} pos_i(o)$$

It should be noticed that predicates in lists *Pre*, *On*, and *Pos* are totally ordered, and should be evaluated respecting that order, using a short-circuit logic: once a single predicate evaluation returns *false*, the remaining predicates are not evaluated. In other words, each predicate list can be seen as a program, i.e., a sequence of statements to be evaluated in sequence. In addition, the evaluation strategy adopted in our model considers that a request should be granted if there is not a predicate that explicitly denies usage (i.e., its evaluation returns *false*). Therefore, an empty predicate list does not have the power to deny accesses and is not considered to express a complete UCON policy, according to the rules expressed by Zhang et al. (2005). This approach was also used in Woo and Lam (1992).

The usage control model proposed in this paper is derived from the formal UCON_{ABC} model as presented in Park and Sandhu (2004), Zhang et al. (2004), with adjustments (like the notion of *usage sessions*) to better fit in an operating systems context. Nevertheless, although the original UCON model is powerful enough to encompass traditional DAC mechanisms which are present in most operating systems, this work considers that the usage control model discussed here is not meant to replace the underlying OS controls, but to offer an additional decision level to them, more accurate, flexible, and consistent.

4. The usage control framework

The usage control model presented in Section 3 consists basically of predicate lists to be sequentially evaluated before, during, or after usage sessions. Predicates in each list are boolean expressions or attribute updates, written using a simple language that will be presented in Section 5.

The usage control framework consists of a usage reference monitor, which takes usage decisions through the evaluation of predicate lists involving user and object attributes, obligations, and conditions, and a usage mediator, which receives access/usage requests from user processes, and invokes the usage reference monitor whenever needed. It should be noticed that the usage control framework does not replace the standard kernel DAC mechanism, but add another decision level to it: an access will be granted only if both the DAC controls and the usage controls allow it. Such structures are depicted in Fig. 1.

The initial values of user/object attributes are defined in *attribute files*. Each user may be associated with a user attribute file, and each object may have its corresponding object attribute file. Attribute values are normally read once and stored in memory, but the updated attribute values may be written back to the files, to provide attribute persistence between system reboots or faults. Each object may be also associated with three policy files as follows:

- the *pre policy* file contains the rules to be evaluated and the attribute update operations to be performed when a subject requests access to that object. Boolean predicates and attribute updates can be interleaved as needed to obtain the desired policy;
- the *on policy* file contains the rules to be verified and the attribute update operations to be performed whenever the object is used, i.e., after its usage session is open. The evaluation of this policy may be triggered periodically, or at any single event that characterizes a usage for that object;

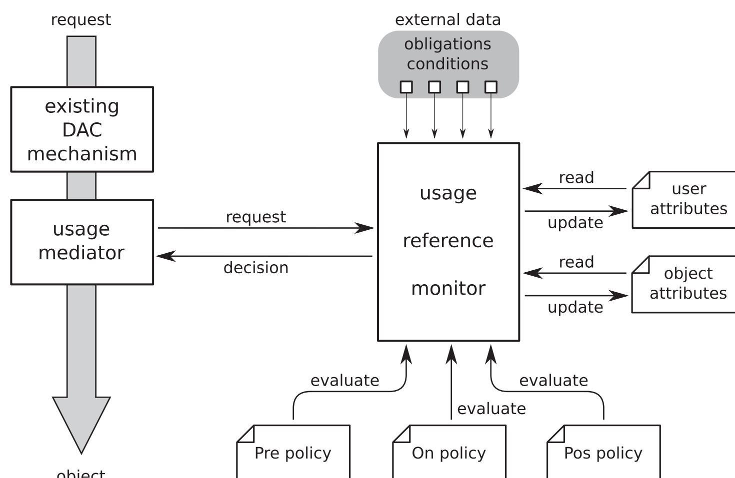


Fig. 1. The proposed usage control framework.

- the *pos police* file contains the attribute update operations to be performed when a usage session on the object is closed or revoked. There are no conditional rules to be evaluated here, only attribute updates, since there is no access/usage request being evaluated.

A missing or empty policy file means only that object does not have a *Pre*, *On*, or *Pos* usage policy associated with it. The same default behavior applies to user and object attribute files. Thus, the framework does not interfere with requests for which no usage policy is defined.

The mechanisms responsible for gathering obligation and condition values should be carefully designed, due to the nature of data they contain. Obligations and conditions concern to external information, which should be provided by external code, outside the reference monitor. A simple way to obtain such external data would be to associate obligations and conditions with external routines. Each time a given obligation or condition needs to be evaluated, its respective routine would be triggered, thus retrieving up-to-date values. However, in an OS context this solution is not adequate for obligations, since the policy evaluator/enforcer is a mechanism running in the operating system kernel. Triggering an external user-supplied code to run using the kernel privileges may adversely affect the system security.

As obligations usually refer to information coming from the user's context, a safer approach to retrieve such information is to define typed *data slots* representing them. Such data slots, which are readable by the reference monitor, are filled by privileged user-level processes, called here *obligation feeders*, which are responsible for verifying the fulfillment of each obligation and for feeding the respective obligation slots. Although this approach is less flexible than executing external routines to retrieve obligation data, as it constrains the amount and type of data fed to the reference monitor, and introduces some asynchrony between the production and the consumption of such data, it is considered safer.

On the other hand, in an operating system context, conditions usually refer to data available at kernel level, like the processor usage level, date/time, etc. Triggering routines for gathering such data does not imply a security risk because such routines can be considered part of the reference monitor framework. A basic set of conditions is described in Section 5, but it can be easily extended by adding new constructs to the grammar, relating condition keywords to the corresponding routines for gathering the required information. Figure 2 illustrates the use of data slots

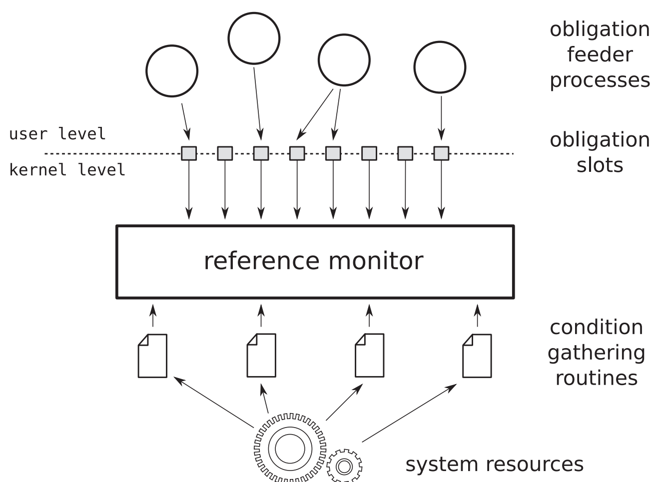


Fig. 2. Information flow for obligations and conditions.

to feed/retrieve obligation data, and routines for gathering condition information to be supplied to the reference monitor.

5. The usage control language

The usage control model discussed in Section 3 defines elements like user/object attributes, authorizations, obligations, and conditions. It also defines predicates for attribute updates and access/usage rules. This section proposes a language to allow a system administrator to associate attributes with users and objects, and to define rules that use such attributes to take usage decisions. The intended application context for this language is the access/usage control of local resources in a multiuser operating system.

5.1. Language requirements

Translating the predicate-based usage control model into a grammar for expressing practical usage control rules requires solving some problems, such as:

- *The language should be simple*, to be easily, clearly, and unambiguously understood and used by system administrators. Nevertheless, it should be expressive enough to represent most predicates involving user/object attributes, authorizations, obligations, and conditions.
- *The grammar should be flexible*; particularly, it should be possible to add new control rules without modifying the user's attributes, provided that all attributes required by the new rules are already defined.
- *Policy-object bindings*: since distinct system objects may have distinct usage needs, it should be possible to define object-specific usage rules, as well as system-wide rules.
- *The grammar parser should be fully implementable* and easy to integrate within an existing environment (like an operating system kernel).
- *The rule evaluation should be efficient*, otherwise the time required to take an access decision may hinder the user experience and degrade the system performance (i.e., it should not take longer than in traditional access control mechanisms to decide about an access request).
- *Safe evaluation of obligations and conditions*: obligations and conditions are, in general, strongly tied to data coming from external sources which are not related to the usage control monitor; the mechanisms used to generate/retrieve such data should be carefully designed to not compromise the system security.

The simplicity, clearness, and efficiency requirements stated above lead to a LALR(1) grammar as a natural choice for expressing usage control rules. Parsers for such grammars are generally small, fast, memory-efficient, making them suitable to be embedded in an OS kernel. Furthermore, they can be automatically generated from the grammar specification (DeRemer and Pennello, 1982), reducing the risk of errors in the parser code itself.

5.2. Language structure

The language will be used to describe users' and objects' attributes, obligations, and conditions, and to build predicates on them. Such predicates will then constitute the $Pre(o)$, $On(o)$, and $Pos(o)$ lists. As mentioned in Section 3, such predicates may be boolean expressions or single attribute updates.

Table 1
Symbols for variables and values used in the language.

Symbol	Type	Description
$*name*	Integer, string	Represents a named variable, beginning with the $ symbol, followed by its unique name
digits	Integer	A constant integer value
alphanum chars	String	A constant string value
$int_1 \ldots int_n$	Integer	A constant set of integers
$str_1 \ldots str_n$	String	A constant set of strings
o$*slot*	Integer	Keyword to access obligation values (slots)
c&*name*	Integer	Keyword to access a value for the system resource *name*, used to express conditions

Table 2
Operators defined by the language.

Operator	Type	Functionality
=	Assignment	Assigns the value on the right to the variable on the left
== != < > <= >=	Comparison	Perform comparisons between left and right operands
& |	Logical	Logical operators AND and OR
size	Set operation	Returns the number of elements in a set
+ - * /	Arithmetic	Arithmetic operations between left and right numeric operands
+ *	Set operation	Union and intersection between left and right set operands
(*expression*)	Precedence	The inner-most expression should be analyzed before the non-parenthesized expression
#	–	Starts a comment

Attributes, obligations, and conditions are represented by variables which may have two different types: `integer` and `string`. An attribute variable is represented by a $ sign followed by its name (e.g., `$name`). When a variable is created, a value must be assigned to it. The value initially assigned to a symbol defines its type: if a symbol is first assigned an integer value, it will always be treated as an integer.

Obligations are represented as numbered slots, defined using the keyword `o$slot` followed by a slot number (e.g., `o$slot 37`). The obligation slot numbering is system-wide. Similarly, conditions are represented by keywords having the form "c$*name*". A standard set of conditions is defined for building usage rules associated with usual system resources, like processor, memory, and disk usage, date, time, and so on. In our current prototype (cf. Section 7), the following conditions are defined: `c$time` (current time), `c$cpu_used` (amount of CPU in use), `c$free_mem` (amount of free memory), `c$free_disk` (free disk space).

The language terminal symbols consist of variable names, keywords, constant values, and operators. Tables 1 and 2 present the terminal symbols: Table 1 shows those symbols related to variables and values, and Table 2 introduces the operators and their functionality (the operators are shown in increasing order of precedence). The most relevant grammar production rules for the usage control language are expressed in Fig. 3, using the EBNF notation.

Such grammar allows building complex boolean expressions, involving attributes, obligations, and conditions. Syntax and semantic errors (like reading from an undefined slot) are detected and treated during the rule parsing. To provide fail-safe defaults, an error in an assignment expression causes it to fail, whereas an error in a boolean expression causes it to be evaluated as false (thus leading to an access/usage denial). For convenience, long input lines can be "folded" in a multiple line representation, by breaking lines when appropriate and starting the continuation lines with a whitespace.

6. Usage control examples

A trivial example of access/usage control policy using the proposed language and its framework could be the following: for a given object, only its owner is authorized to access it. This requirement could be put as simply as follows:

1. first, each user's attribute file should contain an attribute to define her identity (it is assumed that each user has a distinct ID):
   `$userID = 4323`
2. then, the object's attribute file should define an attribute informing who is the object's owner:
   `$ownerID = 7503`
3. finally, a boolean predicate should be put in the object's *pre policy* file, to define the prerequisite to access the file:
   `$userID == $ownerID`

When a given user requests access to that file, the corresponding user and object attribute files are read (if not already in memory) and then its *Pre(o)* file is evaluated; access will only be granted if the user ID is defined and it is equal to the object's owner ID. Obviously, in this trivial example, the access would be denied because the IDs are distinct (4323 ≠ 7503).

In the following, more complex examples of policies defined using the language described here are presented. When comparing such examples with those presented in Park and Sandhu (2004) and Zhang et al. (2005), it can be seen that the proposed grammar can express the policies formally defined in the original UCON model proposal.

6.1. Discretionary access control with ACL

Discretionary access control (Lampson, 1974) with access control lists is a very simple and straightforward control mechanism, which only takes a few lines to be implemented using our language. The user's attributes should only contain her identification: `$userID = 5456`. The enforcer may inform the requested right as a string like "`read`" and "`write`". The object's attribute file should look like:

```
# set of IDs of users with read permission
$readers = 1549 4334 5456 8997

# set of IDs of users with write permission
$writers = 4456 5456 7896 8345
```

```
⟨input⟩       =  { ⟨line⟩ };
⟨line⟩        =  {( ⟨exp⟩ | ⟨comment⟩ ), "\n" };
⟨comment⟩     =  "#" , ? all visible characters ? ;
⟨exp⟩         =  ⟨ATT⟩ , "=" , ⟨values⟩
              |  ⟨exp⟩ , ( "==" | "<" | ">" | "!=" | "<=" | ">=" | "&" | "|" ), ⟨exp⟩
              |  "(" , ⟨exp⟩ , ")"
              |  "size" , ⟨white spaces⟩ , ⟨exp⟩
              |  ⟨values⟩ ;
⟨values⟩      =  ⟨attmath⟩ | ⟨condition⟩ | ⟨obligation⟩ ;
⟨attmath⟩     =  ( ⟨digit⟩ ,{ ⟨digit⟩ })| ⟨SET⟩ | ⟨ATT⟩ | ⟨attmath⟩ ,
                 ( "+" | "-" | "*" | "/" ), ⟨attmath⟩ ;
⟨condition⟩   =  "c$cpu_used" | "c$free_mem" | "c$free_disk" | "c$time" ;
⟨obligation⟩  =  "o$slot" , ⟨white spaces⟩ ,{ ⟨digit⟩ | ⟨ATT⟩ };
⟨ATT⟩         =  "$" , ⟨string⟩ ;
⟨SET⟩         =  { ? all visible non-reserved characters ? , ⟨white spaces⟩ ,
                 ? all visible non-reserved characters ? };
⟨string⟩      =  ⟨alphabetic character⟩ ,{ ⟨alphabetic character⟩ | ⟨digit⟩ };
```

Fig. 3. Grammar rules in EBNF notation.

These attributes are groups of users' IDs that have read or write permission. The *pre* policies file could simply contain:

```
(
  # for reading, user ID should be in $readers set
  ($right == read) & (size ($userID *$readers) != 0)
) |(
  # for writing, user ID should be in $writers set
  ($right == write) & (size ($userID *$writers) != 0)
)
```

This policy basically states that if a user is requesting read or write permission, the user's ID must be present in the `$readers` or `$writers` list, respectively; otherwise, the access is denied.

6.2. Mandatory access control

A simplified version of the Bell La Padula mandatory access control policy (Bell and LaPadula, 1976) may be implemented by considering clearance as a user's attribute, and classification as an object's attribute, both integers. Thus, the user's attribute file contains her clearance level and her current security level:

```
$clearance = 5
$currlevel = $clearance
```

Similarly, the object's attribute file contains its current classification level:

```
$classification = 3
```

The *pre* policy file should then ensure that read access is granted only if the user's current level is higher or equal to the object's classification, and that write access is granted only if the user's current level is lower or equal to the object's classification:

```
(
  # don't read up
  ($right == read) & ($currlevel >=
  $classification)
) |(
  # don't write down
  ($right == write) & ($currlevel <=
  $classification)
)
```

This assumes that the enforcer fills the variable indicating the right required (`$right`), similarly to the previous example.

6.3. Usage control with obligation fulfillment

Let us consider a policy in which the user has to keep an advertisement window open while accessing an object. An external program, possibly the one controlling the window, will update an obligation slot indexed by the user's ID. The user's attribute file contains only this ID, for instance `$userID = 5899`. The object has only one associated file, containing its *on* policies, which are going to be checked every time the right over the object is to be exercised (e.g., when reading the next seconds from a music file):

```
# access to the slot indexed by UID
o$slot $userID == 1
```

The slot indexed by the user's ID is created by the system when the user requests access to the object. The external program that controls the advertisement window writes 1 in this slot once the window is opened, and changes it to 0 when it is closed.

6.4. Limit of the number of simultaneous users

Let us consider a policy in which the access to a given object should be limited to 10 simultaneous users between 8am and 6pm and to 20 users between 6pm and 8am. Users already accessing the object will not have their access revoked when the time shifts for a period admitting a smaller number of simultaneous accesses, but no new user will be accepted until the number of users falls under the limit. To implement this policy, the object's attributes file contains the number of simultaneous accesses accepted and the start and end times for the changes on the maximum number of users:

```
# number of current users
$users = 0
# max simultaneous users during the day
$max_day = 10
# max simultaneous users during the night
$max_night = 20
# day starts 8 o'clock (8am)
$day_start = 8
# day ends 18 o'clock (6pm)
$day_end = 18
```

The *pre* policies file controls the number of simultaneous users and updates the variable controlling this number:

```
(
  # it's day, check for max users during day
  ((c$time > $day_start) & (c$time < $day_end))
  &
  ($users < $max_day)
) |(
  # it's night, check for max users during night
  ((c$time < $day_start) & (c$time > $day_end))
  &
  ($users < $max_night)
)

# increments current users counter
$users = $users + 1
```

The last line states that, if access is granted to the user, then the number of users is updated. If the first rule fails, the parsing of the file stops and the update is not performed. There should also be a *pos* update file, for decreasing the number of current users when a user finishes accessing the object:

```
# decrements current users counter
$users = $users − 1
```

6.5. Limit on usage time

It is possible to define a policy to limit the amount of time a user can access an object. For achieving this, the user's attribute file should contain an attribute to track her total usage time and another attribute to record the time of her last action:

```
# total usage time (in hours)
$total_usage = 0
# time of last action
$last_action = 0
```

The object should also have an attribute to define the maximum usage time (6 h in this case):

```
# max number of hours per user
$max_usage = 6
```

The statements implementing such policy are divided into three files. The *pre* file should contain:

```
# stores the time this action was performed
$last_action = c$time
```

The *on* file should contain:

```
# $max_usage must not be hit
$max_usage > $total_usage

# increments $total_usage by the amount of
# time between this and the last action
$total_usage = $total_usage + c$time - $last_action
$last_action = c$time
```

Finally, the *pos* file should be:

```
# when the access ends, the control attributes
should be reset
$total_usage = 0
$last_action = 0
```

By setting to zero the user's attributes, the policy allows her to release the object and request it again. This policy could be modified to leave the total usage time recorded, requiring an administrative action to allow the user to access the object again, after attaining her maximum usage time.

6.6. Basic RBAC

In role-based access control (Sandhu et al., 2000), a user can only activate roles she is authorized to. Two user attributes are necessary to represent this: one for the user's authorized roles and other for the user's active roles. Thus, the user attribute file should contain:

```
# roles authorized for this user
$roles = director manager teller


# currently active roles
$active_roles = manager teller
```

These two variables provide many-to-many user-role assignment (`$roles` is a group of roles, and each role can be assigned to other users), support for user-role assignment review (one can ask for the contents of `$role`) and a user can activate multiple roles simultaneously (`$active_roles` is also a group of roles). The actual access control is implemented as a combination of object's attributes and policies. Just one object's attribute is required:

```
$required_roles = teller manager
```

The *pre* policies should take care that at least one of the required roles should be present on the user's roles list, and update the active roles to add this role:

```
# roles required to access the object and roles
  available
# to the user must have a non-null intersection:
  size ($required_roles *$roles) != 0


# mark the role as active
$active_role = $active_role + ($required_roles
 *$roles)
```

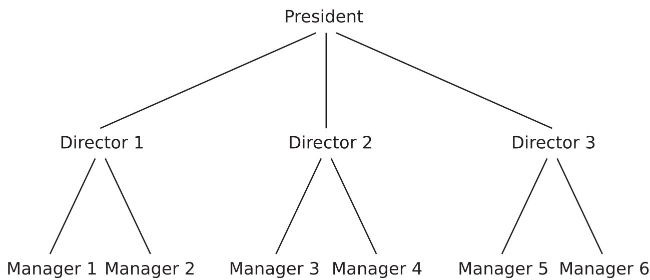


Fig. 4. A typical RBAC hierarchy.

6.7. Hierarchical RBAC

Our language gives support to limited partial ordering or hierarchy (RBAC level 2 Sandhu et al., 2000). Since this language does not support complex data structures, such as trees or linked lists, the hierarchies have to be built using sets. To illustrate this, let us take as an example the hierarchy shown in Fig. 4.

To describe this hierarchy, a fourth file containing system-wide values could be used to hold the relationships between roles. However, for the sake of simplicity, in this example the hierarchy is defined using object's attributes. The rules to build the hierarchy take advantage of the ability to expand variables to their contents when assigning values to a new variable:

```
$Director_1 = Manager_1 Manager_2 Director_1
$Director_2 = Manager_3 Manager_4 Director_2
$Director_3 = Manager_5 Manager_6 Director_3
$President = $Director_1 $Director_2 $Director_3
 President
```

Each director has her role and the roles of the managers below her. The president has her roles created by expanding every role held by all the directors and by adding her own. The access control rules are similar to those presented in the previous example.

6.8. Constrained RBAC

The constrained RBAC model (Sandhu et al., 2000) includes separation of duty (SoD) support. Such behavior can be achieved here basically through a modification of the *pre* policy file and the definition of a *on* policy file. The *pre* policy should ensure that two conflicting roles cannot be activated at the same time. This can be easily done by ensuring that only one role is currently active (the size of the intersection between the set of active roles and the set of required roles is one).

```
size ($active_role * $required_roles) == 1
```

This example assumes that the roles listed on `$required_roles` are mutually exclusive. If that is not the case, it is also possible to add an attribute to the object's attribute file, listing the conflicting roles. The following *on* policy guarantees that, if a conflicting role is activated by an access to another object (in which these roles are non-conflicting ones) or if the required role is no longer active, the usage session will be revoked:

```
size($active_role *$required_roles) == 1
```

This rule is identical to the rule defined in the *pre* file; since it should be continuously verified during the object usage session, it should be placed in the *On(o)* policy file.

7. Evaluation prototype

The usage control model and framework proposed in this paper were partially implemented as a proof-of-concept prototype, incorporated into an operating system kernel. The prototype allows a system administrator to define usage policies for file access operations using our grammar rules. This section describes the prototype architecture and the experiments performed to evaluate it.

7.1. Prototype architecture

The prototype is composed of a rule parser, a usage reference monitor and a usage mediator. The parser translates the rules expressed by the grammar into an internal representation to be used by the reference monitor. The usage mediator is responsible for intercepting system calls related to file access, after they are accepted by the standard DAC mechanism. Figure 5 gives an overview of the prototype. The existing DAC structure is represented by the gray box, which encapsulates the DAC reference monitor and its enforcer.

The usage mediator is the entry point of the system; it is activated during the execution of some relevant system calls. It analyzes the right requested and consults the usage reference monitor for a decision. The monitor will then evaluate each rule and the set of rights, in order to decide granting or denying the request. The process of evaluating a policy consists of setting the semantic values to the rules expressed through the grammar (associating requirements to objects), and deciding if the subject attributes meet the object's required rights. If so, the request is allowed, otherwise it is denied. If any rule fails, the reference monitor returns `false` to the usage mediator, which takes the appropriate actions to refuse the access and/or to revoke active rights, if any.

The mediator intercepts system calls used in file operations, such as `open`, `close`, `read`, and `write`. The *pre* rules evaluation and attributes updating are performed during `open` requests, while ongoing actions are performed in the `read` and `write` calls. *Pos* updates are realized when the `close` system call is invoked, or when a previous rule evaluation results in access denial. As the file usage is achieved through the `read` and `write` operations,[1] mutable attributes can be checked before such operations are actually performed. This approach is consistent with the model presented in Park and Sandhu (2004). Table 3 lists all the system calls intercepted by the current mediator implementation.

It is important to observe that the usage control mechanism does not *replace* the default access control mechanism used by the kernel (i.e., UNIX file permissions), but only complements it. Thus, if a usage control policy denies an access, the access is definitely denied; if it approves the access, the corresponding file permissions are considered. If a file has no usage policies associated to it, then only its file permissions are checked. This strategy simplified the implementation, as only usage policies for some files of interest need to be defined.

7.2. Implementation details

The prototype was implemented in the OpenBSD *4.1* UNIX kernel, using C and the compiler generator *Bison*. This operating system was chosen due to its small and well documented kernel source code. The mediator is implemented through hooks in the kernel function `syscall`, responsible for system call dispatching in the kernel. When this function is activated, the mediator

[1] For the sake of simplicity, other file-related operations, like `lseek`, `mmap`, and others, are not considered here.

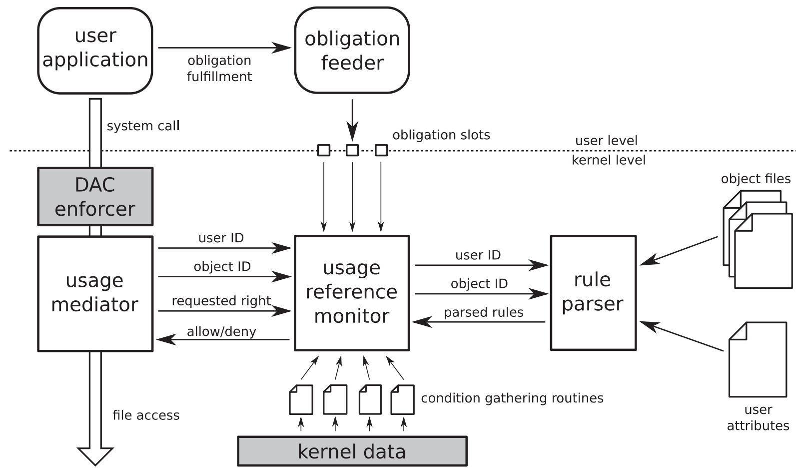


Fig. 5. Overview of the prototype architecture.

Table 3
File-related system calls intercepted by the current mediator implementation.

Syscall	Description	Policy
`SYS_open`	Open a file for reading or writing	*pre*
`SYS_close`	Close a file referred by a descriptor	*pos*
`SYS_read`	Read from an open file	*on*
`SYS_write`	Write to an open file	*on*
`SYS_recvmesg`	Receive a message from a socket	*on*
`SYS_recvfrom`	Receive a message from a socket	*on*
`SYS_sendmsg`	Send a message through a socket	*on*
`SYS_sendto`	Send a message through a socket	*on*
`SYS_accept`	Accept a connection from a socket	*pre*
`SYS_socket`	Create a communication endpoint	*pre*
`SYS_connect`	Establish a connection on a socket	*pre*
`SYS_listen`	Wait for a connection from a socket	*pre*
`SYS_execve`	Execute a file	*pre*

collects information about the request, submits it to the reference monitor and waits for a decision. If the access is granted, the `syscall` function continues its execution. However, if the access is denied by the reference monitor, the `syscall` function jumps to error handling and returns the `EACCES` (*Access error*) status to its caller process.

The `gucon_evaluate` hook function, called at the beginning of the `syscall` code, gets as parameters the system call data structure and the user requesting it. This function locates the *inode* and the *device number* of the requested file access, and retrieves the corresponding policy in the object directory: `/var/gucon/obj/`*devnumber*`/`*inode*. The object attributes are in the same directory as the policies; the user attributes are stored in the `/var/gucon/usr/`*userID* file. The object directory also contains the files used as obligation *slots*.

7.3. Experiments

The scenario defined to evaluate the prototype consisted in playing a standard MP3 music file. The file is 5 MB large and was played using `mpg123`, a very simple music player with a command line interface. The usage policy defined for this file is:

- at most 10 simultaneous users can play the file;
- the file only can be played if the current processor usage is under 30%;
- an externally defined obligation, simulated by having the value 1 on the obligation slot 1, should be continuously met.

These usage policy statements lead to the definition of the following file attributes, user attributes, and *pre*/*on*/*pos* policies. The file attributes are:

```
# maximum number of simultaneous users
$maxusers = 10
# current number of simultaneous users
$currusers = 0
# maximum processor usage (in %)
$maxcpu = 30
# (arbitrary) value for the obligation
$slotvalue = 1
# authorized user groups
$groups= USERS ADMINS
```

The user attributes are:

```
# group to which the user belongs
$user_group = USERS
```

The *pre* policy file contents are:

```
# is this user group authorized?
size ($groups *$user_group) > = 1
# too many users?
$currusers < = $maxusers
# increment user count
$currusers = $currusers + 1
```

The *on* policy file contents are:

```
# is the cpu load acceptable?
$maxcpu < = c$cpu_used
# is the obligation slot value met?
o$slot 1 = = $slotvalue
```

The *pos* policy file contents are:

```
# decrement user count
$currusers = $currusers − 1
```

The experiments described hereafter were performed in a PC equipped with an AMD Athlon 64 Processor 3200+ (2.0 GHz), with 512 MB RAM. The first experiment consists of checking whether the defined policies were being correctly evaluated and enforced. Initially, we measured the processor usage according to the external obligation rules defined in the *on* policy. When

pushing the processor load above 30% or when setting the `o$slot 1` value to values other than `1`, the `mpg123` player quickly stopped playing the music, due to an `EACCES` error received from the kernel. The meaning of the obligation represented by `o$slot 1` was not defined because it is not relevant for the experiment. It could be, for instance, the previous acceptance of an EULA contract, or keeping an advertisement window open; such requirements would be checked by a user-level obligation feeder process, responsible for defining the value of the "`o$slot 1`" obligation slot.

We also evaluated the system behavior under several concurrent accesses, varying from 1 to 15 users. For each user that opened or closed the audio file, the corresponding *pre* and *pos* policies correctly updated the object attributes. Also, when the `$maxusers` threshold was attained, no new users could open the file until the current number of users went under that threshold.

Before evaluating the impact of policy evaluation in system performance, we inspected how many times each relevant system call was called during the player execution. Whereas the `open` and `close` system calls were invoked only 4 times each, the `read` system call was invoked 18 922 times. This leads us to conclude that only the *on* policy evaluation has a relevant impact in performance, at least in this scenario.

Finally, we also evaluated the impact of the *on* policy evaluation in the system performance. We created a simple policy consisting of the following rule, repeated *N* times, with *N* varying from 0 to 20:

```
$variable == 1
```

Figure 6 presents the time spent in the kernel (*systime*) during the execution of the MP3 player software, according to the number of rules in the *on* policy (the time values presented are the average for 100 executions; coefficient of variation was 10.8% for $N=0$ and 4.2% for $N=20$). We observed that each new rule in the policy linearly increased the *systime* by 34 ms, which means 1.8 μs per `read` invocation. Obviously, longer and/or more complex rules will increase this time.

7.4. Results discussion

The experiments performed show that the proposed model and its implementation can be used in the context of a real operating system to provide the advantages offered by the UCON_{ABC} model, even considering some degradation in the system's performance. Although the rule evaluation performance depends strongly on the number of statements and their complexity, it could be significantly

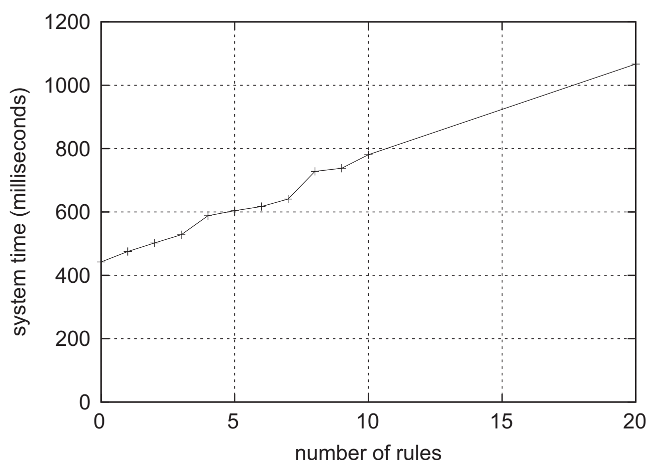


Fig. 6. System time spent evaluating usage control rules.

improved if rule caching techniques were used, to avoid reading them continuously from the disk. Other possible improvement would be to define a binary representation for storing the rules, to reduce the parser size and to improve its efficiency. Such optimizations are planned as future work.

Even considering the system performance degradation, it seems clear that the concepts used to represent attributes, obligations, and conditions satisfy the UCON_{ABC} model description, specially the concept of *slots* to represent obligations. Also, the application of this model to control file usage in an operating system showed that it can be a viable substitute to current ad hoc controls.

As a side result, it was observed that most application programs used during our tests (among them a text processor and an MP3 player) behaved correctly concerning usage denial. When an access request is denied, the corresponding system call returns the `EACCESS` error status, and the applications behave accordingly. Such behavior makes it easier to implement more complex access/usage control models like the one presented here.

8. Related work

There are several languages proposed to address different security and privacy needs. The *Enterprise Privacy Authorization Language* (EPAL) (May, 2004) tries to unify the rules that control how privacy sensitive information should be handled across systems, by creating a universal mechanism for describing required privacy policies. In Wedde and Lischka (2004), the authors propose a modular authorization language to support distributed authorization between cooperating administrative teams, based mostly on RBAC. Woo and Lam (1992) also tackle distributed authorization by introducing a language to encode authorization requirements, which they call a *Policy Base*; in another paper (Woo and Lam, 1998) they further introduce the formal syntax and semantics for a language called *Generalized Access Control List* (GACL) for representing authorization policies based upon ACL. Since GACL is limited to ACL-based mechanisms, Ryutov and Neuman (2000) present a policy language that allows representing several control models (such as ACL, capabilities, lattice-based and RBAC) and a generic authorization and access-control API (GAA API) to facilitate integration of authentication and authorization.

Just like the UCON_{ABC} model is distinct from traditional models by supporting modern needs of continuous usage control and privacy, such as credit-based usage and DRM, the language represented by our grammar differs from the above presented languages by including means to express a more abstract view of control.

There are several works applying the UCON_{ABC} model in specific domains. For instance, Hilty et al. (2007) present a policy language for distributed usage control, with particular interest in DRM interoperability aspects. Distributed systems are also the focus area of Pretschner et al. (2008), which formally models mechanisms to enforce usage control in a distributed setting. To our knowledge, only the works (Xu et al., 2007; Zhang et al., 2008) used the UCON_{ABC} model in the context of an operating system.

In the first paper (Xu et al., 2007), the authors applied usage control and virtual machines to ensure the integrity of an operating system kernel against *rootkit* attacks. The second paper (Zhang et al., 2008) proposes a platform architecture and mechanisms to enforce usage control in heterogeneous distributed environments. Instead of defining a language to specify usage control rules, their approach uses the SELinux framework (Loscocco and Smalley, 2001) as a reference monitor; usage control policies are expressed using XACML, which are automatically translated into SELinux conditional

policies. The standard *policy booleans* facility of SELinux is used to inform the reference monitor about subject/object integrity and environmental conditions. User-level *daemons* are responsible for setting/clearing SELinux policy booleans, which are read by the kernel-level reference monitor, similarly to the obligation slots defined in our approach (Teigao et al., 2007). Such approach is similar to ours, but we understand that using only the boolean flags provided by SELinux to inform the reference monitor is limiting and may lead to overly complex usage control policies.

9. Conclusion

This paper presented a usage control model suited for implementation in an operating system kernel, inspired from the $UCON_{ABC}$ model. It also proposes a framework for usage control and an LALR(1) grammar for specifying usage control policies. The proposed grammar is simple to understand but expressive enough to describe a wide range of policies, like DAC, MAC, RBAC, UCON, and DRM-like policies, as shown in Section 6.

We integrated the proposed framework into a real operating system, to show its feasibility. Although the prototype is functional, there are some open issues to be solved, such as efficiently relating the rules and attribute files to objects and users, and creating and managing obligation slots, which could be overlooked in our proof-of-concept implementation, but are essential for a production system. Also, the solution found to provide condition values is not very flexible, as it depends on data gathering kernel-level routines. Finally, our current model lacks administrative policies to make clear who can modify the user and object attribute files and the corresponding policy files.

References

Bell D, LaPadula L. Secure computer systems: unified exposition and Multics interpretation. Technical Report MTR-2997 Rev. 1, MITRE Corporation; 1976.

DeRemer F, Pennello T. Efficient computation of LALR(1) look-ahead sets. ACM Transactions on Programming Languages and Systems 1982;4(4):615–49.

Hilty M, Pretschner A, Basin D, Schaefer C, Walter T. A policy language for distributed usage control. In: Biskup J, López J, editors. Computer security ESORICS 2007. Lecture notes in computer science, vol. 4734. Springer; 2007. p. 531–46.

Katt B, Zhang X, Breu R, Hafner M, Seifert J-P. A general obligation model and continuity-enhanced policy enforcement engine for usage control. In: ACM symposium on access control models and technologies; 2008. p. 123–32.

Lampson B. Protection. SIGOPS Operating System Review 1974;8(1):18–24.

Loscocco P, Smalley S. Integrating flexible support for security policies into the Linux operating system. In: Usenix Annual Technical Conference; 2001. p. 29–42.

May MJ. Privacy system encoded using EPAL 1.2. Technical Report, University of Pennsylvania; August 2004.

Park J, Sandhu R. Usage control: a vision for next generation access control. In: Gorodetsky V, Popyack LJ, Skormin VA, editors. International workshop on mathematical methods, models, and architectures for computer network security, Lecture Notes in Computer Science, vol. 2776. Springer; 2003. p. 17–31.

Park J, Sandhu R. The $UCON_{ABC}$ usage control model. ACM Transactions on Information and System Security 2004;7(1):128–74.

Pretschner A, Hilty M, Basin D, Schaefer C, Walter T. Mechanisms for usage control. In: ACM symposium on information, computer and communications security. ASIACCS '08; 2008. p. 240–4.

Ryutov T, Neuman C. Representation and evaluation of security policies for distributed system services. In: DARPA information survivability conference and exposition. Healton Head, South Carolina, January, 2000. p. 1172.

Sandhu R, Ferraiolo D, Kuhn R. The NIST model for role-based access control: towards a unified standard. In: ACM workshop on role-based access control. New York, NY, USA: ACM Press; 2000. p. 47–63.

Sandhu R, Samarati P. Access control: principles and practice. IEEE Communications Magazine 1994;32(9):40–8.

Teigao R, Maziero C, Santin A. A grammar for specifying usage control policies. In: IEEE international conference on communications; 2007. p. 1379–84.

Wedde HF, Lischka M. Modular authorization and administration. ACM Transactions on Information and System Security 2004;7(3):363–91.

Woo TYC, Lam SS. Authorization in distributed systems: a formal approach. In: IEEE symposium on research in security and privacy; 1992. p. 33–51.

Woo TYC, Lam SS. Designing a distributed authorization service. In: IEEE international conference on computer communications; 1998. p. 419–29.

Xu M, Jiang X, Sandhu R, Zhang X. Towards a VMM-based usage control framework for OS lernel integrity protection. In: ACM symposium on access control models and technologies; 2007. p. 71–80.

Zhang X, Parisi-Presicce F, Sandhu R, Park J. Formal model and policy specification of usage control. ACM Transactions on Information and System Security 2005;8(November):351–87.

Zhang X, Park J, Parisi-Presicce F, Sandhu R. A logical specification for usage control. In: ACM symposium on access control models and technologies; 2004. p. 1–10.

Zhang X, Seifert J-P, Sandhu R. Security enforcement model for distributed usage control. In: IEEE international conference on sensor networks ubiquitous and trustworthy computing; 2008. p. 10–8.