

Term Rewriting for Access Control

Steve Barker Maribel Fernández

June 2006

Access control is of fundamental importance in computer security. Formal specifications of access control models and policies make it possible to

- compare policies rigorously,

Access control is of fundamental importance in computer security. Formal specifications of access control models and policies make it possible to

- compare policies rigorously,
- understand the consequences of modifying policies, and

Access control is of fundamental importance in computer security. Formal specifications of access control models and policies make it possible to

- compare policies rigorously,
- understand the consequences of modifying policies, and
- **prove properties of policies.**

We propose to use *term rewriting* for the formalisation of access control models (in particular ACL and RBAC models and policies).

Also, we will use rewriting for access request evaluation.

Although the emphasis in the literature has been on the use of logic languages, there are several reasons to consider the use of term rewriting approaches:

Why using term rewriting to formalise access control models and policies?

- **Expressivity:** in the past, rewriting systems have been used to specify, in a uniform way, several computational paradigms, including functional, logic, imperative and concurrent ones. We will show that rewriting can also be used to define ACL (Access Control Lists) and RBAC (Role Based Access Control) policies in a uniform and formal way.

Why using term rewriting to formalise access control models and policies?

- Expressivity: in the past, rewriting systems have been used to specify, in a uniform way, several computational paradigms, including functional, logic, imperative and concurrent ones. We will show that rewriting can also be used to define ACL (Access Control Lists) and RBAC (Role Based Access Control) policies in a uniform and formal way.
- A well-developed theory: rewriting techniques can be used to prove properties of policies specified as rewriting systems. We will use them to prove consistency, correctness and completeness of access control policies.

Why using term rewriting to formalise access control models and policies?

- Expressivity: in the past, rewriting systems have been used to specify, in a uniform way, several computational paradigms, including functional, logic, imperative and concurrent ones. We will show that rewriting can also be used to define ACL (Access Control Lists) and RBAC (Role Based Access Control) policies in a uniform and formal way.
- A well-developed theory: rewriting techniques can be used to prove properties of policies specified as rewriting systems. We will use them to prove consistency, correctness and completeness of access control policies.
- Availability of tools such as ELAN, MAUDE and CiME to test, compare and experiment with evaluation strategies, to automate equational reasoning, and also for rapid prototyping of access policies.

The Access Matrix and Access Control Lists

The language of the access matrix [Lampson 74] includes a finite set \mathcal{U} of *users* (e.g., human users and software agents), a finite set \mathcal{O} of *objects* (e.g., files and directories), and a finite set \mathcal{A} of *access privileges* (e.g., read, write and execute privileges).

$u \in \mathcal{U}$ is authorised to exercise an access privilege $p \in \mathcal{P}$ on an object $o \in \mathcal{O}$ if and only if the access matrix includes an entry that specifies that u is assigned the p privilege on o .

An access matrix is usually implemented as an Access Control List (ACL).

Role-based Access Control

- A user u of a resource o may be assigned to a set of roles $\{r_1, \dots, r_n\}$
- access privileges on resources are also assigned to roles;
- a user u may exercise an access privilege p on a resource o if and only if u is assigned to a role r to which the privilege p on o is also assigned.

The most basic category of RBAC model, flat RBAC (or $RBAC_F$ for short), requires that these types of assignment are supported.

$RBAC_{H2A}$ extends $RBAC_F$ to include the notion of an RBAC role hierarchy.

$RBAC_{C3A}$ extends $RBAC_{H2A}$ by allowing constraints on policies to be represented.

$RBAC_{S4A}$ extends $RBAC_{C3A}$ by allowing administrator queries to be evaluated with respect to an RBAC policy specification.

Formalising Role-based Access Control

The semantics of user-role assignment is usually defined by a 2-place *ura* predicate, and permission-role assignment is defined by a 3-place predicate *pra*.

A *role hierarchy* is defined as a (partially) ordered (and finite) set of roles. The ordering relation is a role seniority relation, a 2-place predicate *senior_to*(r_i, r_j).

Example

Suppose that the users u_1 and u_2 are assigned to the roles r_2 and r_1 respectively, and that write (w) permission on object o_1 is assigned to r_1 and read (r) permission on o_1 is assigned to r_2 . Moreover, suppose that r_1 is directly senior to r_2 in an $RBAC_{H2A}$ role hierarchy.

This $RBAC_{H2A}$ policy is represented by the relations:

$$ura(u_1, r_2), ura(u_2, r_1), pra(w, o_1, r_1), pra(r, o_1, r_2), ds(r_1, r_2).$$

User-role and permission-role assignments are related via the notion of an *authorisation*. An authorisation is a triple (u, a, o) that expresses that the user u has the a access privilege on the object o . In the previous example, the set of authorisations is:

$$\{(u_2, w, o_1), (u_2, r, o_1), (u_1, r, o_1)\}.$$

To extend $RBAC_{H2A}$ theories to $RBAC_{C3A}$ theories, *separation of duties constraints* must be supported, that is, a user cannot be assigned to a pair of mutually exclusive roles.

To extend $RBAC_{C3A}$ programs to $RBAC_{S4A}$ theories, *permission-role review* must be possible in addition to *user-role reviews*. That is, it must be possible for security administrators to pose queries on RBAC policy specifications to determine

- (i) the set of roles a user is assigned to, and
- (ii) the permissions that are assigned to roles.

Term rewriting systems can be seen as programming or specification languages, or as formulae manipulating systems that can be used in various applications such as operational-semantics specification, program optimisation or automated theorem proving.

They are defined by a syntax (set of terms) and a set of rewrite rules that are used to 'reduce' terms.

Term Rewriting

A *signature* \mathcal{F} is a finite set of *function symbols* together with their (fixed) arity. \mathcal{X} denotes a denumerable set of *variables*, and $T(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built up from \mathcal{F} and \mathcal{X} .

Terms are identified with finite labeled trees.

$\mathcal{V}(t)$ denotes the set of variables occurring in t .

Given a signature \mathcal{F} , a *term rewriting system* on \mathcal{F} is a set of rewrite rules $R = \{l_i \rightarrow r_i\}_{i \in I}$, where $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$, and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$.

t *rewrites* to u at position p with the rule $l \rightarrow r$ and the substitution σ , written $t \rightarrow_p^{l \rightarrow r} u$, or simply $t \rightarrow_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$, where $t|_p$ denotes the *subterm* of t at position p , and $t[u]_p$ is the result of replacing $t|_p$ with u at position p in t .

Irreducible terms are said to be in *normal form*.

Example

Signature for lists of natural numbers:

- *Z (with arity 0) and S (with arity 1, denoting the success or function) to build numbers;*
- *nil (with arity 0, to denote an empty list), cons (with arity 2, to construct non-empty lists), and append (also with arity 2, to represent the operation that concatenates two lists).*

We can specify list concatenation with the following rewrite rules:

$$\begin{aligned}\text{append}(\text{nil}, x) &\rightarrow x \\ \text{append}(\text{cons}(y, x), z) &\rightarrow \text{cons}(y, \text{append}(x, z))\end{aligned}$$

Then we have a reduction sequence:

$$\text{append}(\text{cons}(Z, \text{nil}), \text{cons}(S(Z), \text{nil})) \rightarrow^* \text{cons}(Z, \text{cons}(S(Z), \text{nil}))$$

Properties of Rewrite Rules

A term rewriting system R is:

- *confluent* if for all terms t, u, v : $t \rightarrow^* u$ and $t \rightarrow^* v$ implies $u \rightarrow^* s$ and $v \rightarrow^* s$, for some s ;
- *terminating* (or *strongly normalising*) if all reduction sequences are finite.

For example, the system defining `append` is confluent and strongly normalising.

Access Control Lists as a Rewrite System - An Example

Assume that user identifiers are natural numbers, and to make the example more concrete, assume that the objects are files and the access privileges are read (r), write (w) or execute (x). For simplicity, we will only consider one file (the generalisation to many files is straightforward).

The policy specifies that a user with an even identifier has rw rights whereas users with odd numbers have only r right and users whose identifier is a multiple of 4 have rw .

Requests will be expressed as $\text{access}(u, req)$ where u is a user-identifier and req is either r , w or x .

The requests will be evaluated using the rewrite system R_{ACL} given below; the result will be either grant or deny.

Access Control Lists: Rewrite Rules

$$\text{access}(U, R) \rightarrow \text{acl}(\text{rem}(U, 2), R, U)$$
$$\text{acl}(1, r, U) \rightarrow \text{grant}$$
$$\text{acl}(1, w, U) \rightarrow \text{deny}$$
$$\text{acl}(1, x, U) \rightarrow \text{deny}$$
$$\text{acl}(0, r, U) \rightarrow \text{grant}$$
$$\text{acl}(0, w, U) \rightarrow \text{grant}$$
$$\text{acl}(0, x, U) \rightarrow f(\text{rem}(U, 4))$$
$$f(0) \rightarrow \text{grant}$$
$$f(1) \rightarrow \text{deny}$$
$$f(2) \rightarrow \text{deny}$$
$$f(3) \rightarrow \text{deny}$$

U, R are variables, $\text{rem}(n, m)$ computes the remainder of the division of n by m .

For example, a request from user 101 to write on the file is denied, whereas a request from user 20 to execute it is granted:

$$\begin{aligned} \text{access}(101, w) &\rightarrow_{R_{ACL}}^* \text{deny.} \\ \text{access}(20, x) &\rightarrow_{R_{ACL}}^* \text{grant.} \end{aligned}$$

R_{ACL} provides an executable specification of the policy.

Theorem

The rewrite system R_{ACL} is terminating and confluent.

Corollary

Every term has a unique normal form in R_{ACL} .

As a consequence, our specification of the access control policy is *consistent*.

Theorem (Consistency)

For any user u and request req , it is not possible to derive both grant and deny for a request $access(u, req)$.

We can give a characterisation of the normal forms:

Theorem

The normal form of a ground term of the form $\text{access}(u, \text{req})$ where u is a number and $\text{req} \in \{r, w, x\}$ is either grant or deny.

As a consequence, our specification of the access control policy is *total*:

Theorem (Totality)

Each access request $\text{access}(u, \text{req})$ from a valid user u to perform a valid operation req is either denied or granted.

Correctness and *Completeness* are also easy to check:

Theorem (Correctness and Completeness)

For any user u and request req :

- $\text{access}(u, req) \rightarrow^* \text{grant}$ if and only if u has the access privilege req on the file.
- $\text{access}(u, req) \rightarrow^* \text{deny}$ if and only if u does not have the access privilege req on the file.

Sketch of proof:

Since we have consistency and totality, it is sufficient to show:

$\text{access}(u, req) \rightarrow^* \text{grant}$ if and only if u has the access privilege req .

This is shown by inspection of the rewrite rules.

We model ura as a function (from users to lists of roles) instead of a predicate.

Modelling ura as a function makes it easy to obtain all of the roles that a specific user is assigned to (an essential requirement of $RBAC_F$).

$$\begin{array}{l} \text{roles}(u_1) \rightarrow [r_{11}, \dots, r_{1i}] \\ \vdots \\ \text{roles}(u_n) \rightarrow [r_{n1}, \dots, r_{nk}] \end{array}$$

To represent *pra* (assignment of privileges to roles), we have again two design choices: we could use a boolean function or a function *priv* from roles to lists of pairs $(a, o) \in (\mathcal{A} \times \mathcal{O})$.

As before, the second approach has advantages from a security administrator's point of view.

$$\begin{aligned} \text{priv}(r_1) &\rightarrow [(a_{11}, o_{11}), \dots, (a_{1i}, o_{1i})] \\ &\quad \vdots \\ \text{priv}(r_n) &\rightarrow [(a_{n1}, o_{n1}), \dots, (a_{nk}, o_{nk})] \end{aligned}$$

Example

The user-role and permission-role assignments described in the previous example may be expressed by the following rewrite rules:

$$\begin{aligned} \text{roles}(u_1) &\rightarrow [r_2] \\ \text{roles}(u_2) &\rightarrow [r_1] \\ \text{priv}(r_1) &\rightarrow [(w, o_1)] \\ \text{priv}(r_2) &\rightarrow [(r, o_1)] \end{aligned}$$

To evaluate access requests we use the rules:

$$\begin{aligned} \text{access}(U, A, O) &\rightarrow \text{check}(\text{member}((A, O), \text{privileges}(\text{roles}(U)))) \\ \text{check}(\text{True}) &\rightarrow \text{grant} \\ \text{check}(\text{False}) &\rightarrow \text{deny} \end{aligned}$$
$$\begin{aligned} \text{privileges}(\text{nil}) &\rightarrow \text{nil} \\ \text{privileges}(\text{cons}(R, L)) &\rightarrow \text{priv}(R) \cup \text{privileges}(L) \end{aligned}$$

For example, with the assignment shown in the previous example,
 $\text{access}(u_1, r, o_1) \rightarrow^* \text{grant}$.

Properties of the RBAC Policy

Theorem

The rewrite system R_{RBAC} is terminating and confluent.

Corollary

Every term has a unique normal form in R_{RBAC} .

As a consequence of the unicity of normal forms, our specification of the RBAC policy R_{RBAC} is *consistent*.

Theorem (Consistency)

For any $u \in \mathcal{U}$, $a \in \mathcal{A}$, $o \in \mathcal{O}$: it is not possible to derive, from R_{RBAC} , both grant and deny for a request $\text{access}(u, a, o)$.

We can give a characterisation of the normal forms:

Theorem

The normal form of a ground term of the form $\text{access}(u, a, o)$ where $u \in \mathcal{U}$, $a \in \mathcal{A}$ and $o \in \mathcal{O}$ is either grant or deny.

As a consequence, our specification of the access control policy is *total*.

Theorem (Totality)

Each access request $\text{access}(u, a, o)$ from a valid user u to perform a valid action a on the object o is either granted or denied.

Correctness and *Completeness* are also easy to check:

Theorem (Correctness and Completeness)

For any $u \in \mathcal{U}$, $a \in \mathcal{A}$, $o \in \mathcal{O}$:

- $\text{access}(u, a, o) \rightarrow^* \text{grant}$ if and only if u has the access privilege a on o .
- $\text{access}(u, a, o) \rightarrow^* \text{deny}$ if and only if u does not have the access privilege a on o .

It is important to note that the proofs above do not have to be generated by a security administrator; rather, the proofs demonstrate that an RBAC policy R_{RBAC} satisfies the properties described above. A security administrator can simply base an RBAC policy on the term rewrite system that we have defined and can be sure that the properties of R_{RBAC} hold.

RBAC with a Hierarchy of Roles: $RBAC_{H2A}$

To accommodate a notion of seniority of roles (where a role inherits the privileges of its subordinate roles) we just add rules of the form

$$\text{dsub}(r_i) \rightarrow [r_1, \dots, r_j]$$

when r_1, \dots, r_j are direct subordinate roles of r_i (hence r_i is directly senior to $r_1 \dots r_n$).

Then, we redefine the privileges of a role using dp to compute direct privileges (which corresponds to the previously defined priv) and privileges (defined above):

$$\text{priv}(r) \rightarrow \text{dp}(r) \cup \text{privileges}(\text{dsub}(r))$$

Remark: no need to change the definition of access, and no restrictions on the form of the hierarchy (apart from acyclicity, which is a natural requirement).

For RBAC policies beyond $RBAC_{H2A}$ separation of duties constraints must be supported and it must be possible for security administrators to review policy specifications.

Separation of Duties

Separation of Duties (roles assigned to a user cannot be mutually exclusive) can be ensured by erasing conflicting roles assigned to a user.

This is obtained by evaluation of $\text{clean}(\text{roles}(u))$ in a rewrite system containing the rules:

$$\begin{aligned}\text{clean}(\text{nil}) &\rightarrow \text{nil} \\ \text{clean}(\text{cons}(R, L)) &\rightarrow \text{cons}(R, \text{clean}(\text{eraseclash}(R, L))) \\ \text{eraseclash}(R, \text{nil}) &\rightarrow \text{nil} \\ \text{eraseclash}(R, \text{cons}(R', L)) &\rightarrow \text{eraseclash}(L) \quad (R, R' \text{ clash}) \\ \text{eraseclash}(R, \text{cons}(R', L)) &\rightarrow \text{cons}(R', \text{eraseclash}(R, L)) \quad (\text{no clash})\end{aligned}$$

To check that every user has been assigned a role, an administrator could simply evaluate the term $\text{RolesDefined?}(u)$.

$\text{RolesDefined?}(u)$	\rightarrow	$\text{review}(\text{roles}(u))$
$\text{review}(\text{nil})$	\rightarrow	“error: user without a role”
$\text{review}(\text{cons}(r, lr))$	\rightarrow	“OK”

Using term rewriting systems, different access control models can be flexibly defined in a uniform way. Access requests are evaluated by reduction, and (static) properties of policies are proved using standard rewriting techniques.

In future work, we intend to consider the use of term rewriting for the specification of access control models other than ACL and RBAC: in particular, usage control models, and access control models that may be used in a distributed computing environment.

Questions ?