Rewriting-Based Access Control Policies

Anderson Santana de Oliveira[1],[2]

INRIA & LORIA
615, Rue du Jardin Botanique, 54600 Villers-lès-Nancy, France

Abstract

In this paper we propose a formalization of access control policies based on term rewriting. The state of the system to which policies are enforced is represented as an algebraic term, which allows us to model several aspects of the policy environment. Policies are implemented by sets of rewrite rules, whose evaluation produces authorization decisions. We discuss the relation between properties of term rewriting systems, such as confluence and termination, and their consequences on defining trusted access control policies.

Keywords: Access Control Policies, Term Rewriting Systems

1 Introduction

Term rewriting [1] is a well-established paradigm for specifying and prototyping systems. It has been proved useful in theorem proving, program transformation, and algebraic specification. In many practical situations, its straightforward formal background allowed to rapidly prototype and verify diverse kinds of systems. In the domain of computer security, term rewriting has been successfully applied to help reasoning about some of its aspects, notably in verification of security protocols [17,7].

Nevertheless, there are not many applications of term rewriting to security policies: up to our knowledge, few approaches have tried to introduce the use of rewriting into this domain[2,12]. The reader is referred to see a more extensive discussion on related work in section 7.

Access control concerns stating the *actions* which *principals (or subjects)* are allowed to execute in order to manipulate the *objects (or resources)* of a given system. The most widespread framework for describing this kind of protection is the *access control matrix*, a model adopted in the design of several operating systems. The rows of the matrix contain the subjects (active entities of the system, such as processes and users), the columns list the protected entities of the system (relevant

[1] santana@loria.fr

[2] Supported by Capes BEX.2120.03-8.

objects, such as files), and cells contain which access rights (*read, write, execute, etc.*) are assigned to each active entity with respect to the protected resources. A request from a subject to perform a certain action over an object will be granted only if there exists an entry for that action in the corresponding cell.

Some fundamental models for access control rely on modified variants of the access control matrix. For example, military security policies [4] add confidentiality levels to subjects and objects. Then the fixed security policy states that a subject cannot write to an object with inferior security level, and that it cannot read from objects with superior security levels - *no reads-up, no writes-down*. Even though the access control matrix is a support for a number of formalizations of access control policies, it is not appropriate to capture more dynamic policies, such as policies that depend on time, location, and many other possible attributes of the policy environment.

This paper presents a formalization of access control policies based on term rewriting. Policies are represented as sets of rewrite rules, whose evaluation produces authorization decisions, whilst requests and the environment where policies are enforced are represented as algebraic terms. Since we consider the policy environment as a "database of facts" under the form of a term, this formalization allows us to capture many dynamic aspects that are important for policy enforcement, e.g. diverse attributes of subjects and resources, referred as content-dependent conditions in the literature [10].

The main goals of this work are to provide a formal semantics for an expressive access control policy language, which is able to support dynamic policies; to provide a design, where it is possible to have maintainable enforcement mechanisms, and to characterize trusted policies by associating properties of the corresponding term rewriting systems that implement a security policy. For example, the absence of conflicts is an important property when both positive and negative authorizations are possible. It assures that for a certain access request no grant and denial are assigned at the same time.

Another goal of this formalization is to be able to facilitate policy enforcement. The architecture we propose here clearly separates policy and enforcement mechanism. Since policies are rewrite rules, a standard rewrite engine can do the job of applying the policy to requests and evaluating the results.

The paper is organized as follows: Section 2 recalls some useful definitions on term rewriting systems, Section 3 illustrates by an example the kind of access control policy we want to express, Section 4 presents what are the elements of the policy environment, Section 5 presents and discuss rewriting-based policies, Section 6 describes what kind of security mechanism is necessary to enforce these policies, Section 7 presents a discussion on related works, and Section 8 concludes and points out some future developments.

2 Preliminaries on Term Rewriting Systems

We recall some basic definitions on signatures and terms. A signature $\Sigma = \{\mathcal{S}, \mathcal{F}\}$ is a set of sorts $\mathcal{S}$, together with a set of function symbols, each one associated to a natural number by the arity function ($\text{ar} : \mathcal{F} \to \mathbb{N}$), which denotes the number of arguments. $\mathcal{F}_n$ is the subset of function symbols having n for arity, $\mathcal{F}_n = \{f \in \mathcal{F} \mid \mathrm{ar}(f) = n\}$. $\mathcal{T}(\Sigma, \mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term*, and $\mathcal{T}(\Sigma)$ is the set of all ground terms. A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\Sigma)$, written, when its domain is finite, $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. The result of applying a substitution σ to a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ is written $\sigma(t)$.

A rewrite rule is an ordered pair of terms denoted $l \rightarrow r$, where l, $r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A rewrite system or term rewriting system is a (finite or infinite) set of rewrite rules.

Given a rewrite system R, a term t rewrites to a term t', which is denoted $t \rightarrow_R t'$ if there exist a rule $t \rightarrow l$ of R, a position ω in t, a substitution σ, satisfying $t_{|\omega} = \sigma(l)$, such that $t' = t[\sigma(r)]$.

A subterm $t_{|\omega}$ where a rewriting step can be applied is called *redex*. A term that has no redex is said to be irreducible for R or in R-*normal form*.

A rewrite derivation is any sequence of rewriting steps $t_1 \rightarrow_R t_2 \rightarrow_R \ldots$ A rewrite derivability relation $\xrightarrow{*}_R$ is defined on terms: $t \xrightarrow{*}_R t'$ if there exists a rewriting derivation from t to t'. If the derivation contains at least one step, it is denoted by $\xrightarrow{+}_R$. A term rewriting systems is terminating if all rewrite derivations are finite. It is confluent if for all terms t, u ,v, $t \xrightarrow{*}_R u$ and $t \xrightarrow{*}_R s$ implies $u \xrightarrow{*}_R s$ and $v \xrightarrow{*}_R s$, for some s.

3 Motivating Example

The example that follows has been slightly adapted from the XACML specification [20], an initiative from the Oasis Consortium[3] to create a standard markup language for role-based access control. Let us suppose that a medical corporation adopts the policy below:

(i) A person, identified by his or her patient number, may read any record for which he or she is the designated patient.

(ii) A person may read any record for which he or she is the designated parent or guardian, and for which the patient is under 16 years of age.

(iii) A physician may write to any medical element for which he or she is the designated primary care physician.

(iv) An administrator shall not be permitted to read or write to medical elements of a patient record.

These rules illustrate the dependency of authorizations on the attributes of objects and subjects. This reflects the current needs in terms of flexibility in the declaration of access control for modern applications, being a representative of the kind of policies we are interested in.

The idea behind the statement of access control policies with high level of abstraction is that policy specification and enforcement can be separated from other

[3] http://www.oasis-open.org/

S_i	patient(”Bart Simpson”, 1, 14, guardian(”Homer Simpson”)) + record(patient(”Bart Simpson”, 1, 14, guardian(”Homer Simpson”)), physician(”Julius Hibbert”, 1), antibiotic, payment(visa)) + physician(”Julius Hibbert”, 1)
Request	req(physician(”Julius Hibbert”, 1), writeMedElements, record(patient(”Bart Simpson”, 1, 14, guardian(”Homer Simpson”)), physician(”Julius Hibbert”, 1) , antibiotic, payment(visa)))
S_{i+1}	patient(”Bart Simpson”, 1, 14, guardian(”Homer Simpson”)) + record(patient(”Bart Simpson”, 1, 14, guardian(”Homer Simpson”)), physician(”Julius Hibbert”, 1), antibiotic and aspirin, payment(visa)) + physician(”Julius Hibbert”, 1)

Figure 1. A system state transition after a request to write a medical element

functionalities of the application, thus avoiding bugs and increasing maintainability.

A possible execution of the medical system mentioned above is shown in Figure 1. At a given point of time, the system is in state s_i, where the values of attributes for objects and resources appears in the first line of the table. The “+” operator concatenates terms, it is better explained in section 5. The second line of the table illustrates the fact that the primary physician of a certain patient record, requires to prescribe a new medical element for this patient. According to the policy we just described, this request must be assigned grant access, therefore the system state changes to state s_{i+1} (last line of the table), containing the updated entry for the medical record.

We make a few assumptions about the application: a representation of the current state of the application under the form of a term is always available, as well as the user requests. In practice, this will require to modify the program in order to capture its state, and to intercept the control flow for monitoring intervention, every time a resource is to be accessed.

4 The Policy Environment

Regarding access control, one is interested in stating which *actions*, *principals (or subjects)* are allowed to execute in order to manipulate the *objects (or resources)* of a given system. The most widespread representation schema to this end is the *access control matrix* where lines list the subjects, columns contain the objects and cells keep information about the privileges (*read, right, execute*) assigned to an active entity over the passive ones.

This schema is enough to address most of the requirements for mandatory and

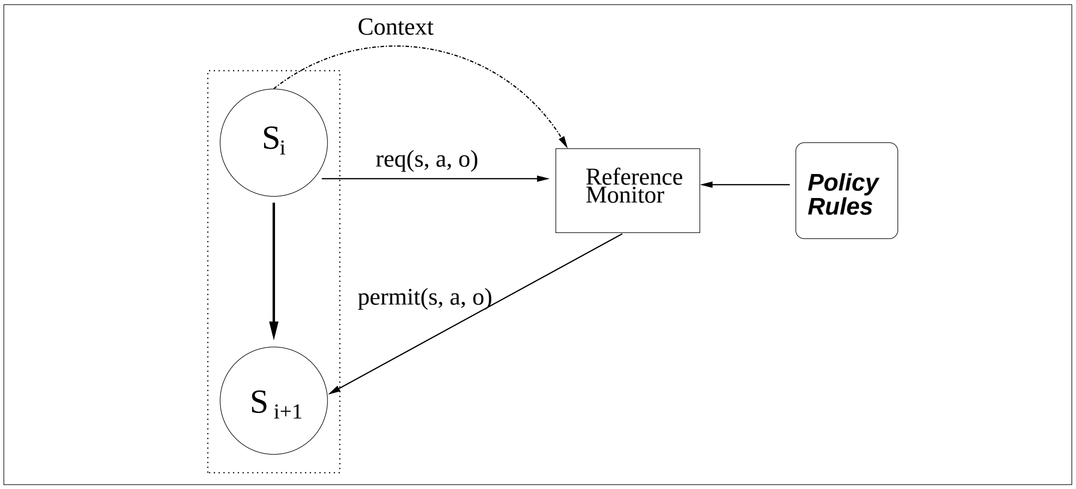

Figure 2. General representation schema

discretionary access control models, but it is not adequate to express highly abstract policies, like the one from our running example, presented in Section 3. For declaring this kind of policy, it is necessary to write sentences about the current values of attributes of subjects and resources, and not only their identities. We call *policy environment* the configuration of all elements relevant to access control.

We call *target system*, denoted by T, any arbitrary application which must respect a given security policy. A target is represented by a set of states and state transitions, which are triggered by access requests. To each state s_i of T we associate an algebraic term containing the facts that are true in s_i. Requests are also represented as terms (see Figure 1).

The idea we defend here, which was also recently exposed in some papers [3,21], is that access control is one aspect of the application, that can be specified, implemented and maintained independently. Furthermore, the mechanism applying a certain policy should be an external entity with respect to the application. An overview picture of policy enforcement is presented in Figure 2. The application context in state s_i together with the current request $req(s, a, o)$ are delivered to a reference monitor. The reference monitor evaluates every request according to the current policy. In the case the policy grants access for the request, the application proceeds. Otherwise, the execution of the application is aborted[4] .

We denote Σ_T the signature of the target system T, which provides the profiles of the constructors needed to build the representation of the system state. Consequently, the database of facts at each stage of the execution of T is a the set ground terms from $\mathcal{T}(\Sigma_T)$.

In out running example, we have used order-sorted specifications to formalize the problem and the system using Maude [9].

Example 4.1 We use the signature below for the medical application described in Section 3, where a patient is represented by a term containing name, number, age, and a guardian, in this order. The keyword `ctor` indicates that the operator

[4] It would be possible to use exceptions to handle negative authorizations, but this issue is not explored in this paper.

it follows is a constructor.

```
fmod MEDICAL-SYSTEM-SIGNATURE is
  protecting STRING .
  protecting NAT .
    sort  Patient Physician Record Administrator Guardian
          MedicalElements OtherElements .
    op patient : String Nat Nat Guardian -> Patient [ctor] .
    op administrator : Nat -> Administrator [ctor] .
    op guardian : String -> Guardian [ctor] .
    op physician : String Nat -> Physician [ctor] .
    op record : Patient Physician MedicalElements
         OtherElements -> Record [ctor] .
endfm
```

In order to better express policies, it is necessary to indicate which sorts from Σ_T are subsorts of subject, action and object, as well as to introduce the available actions in T.

Example 4.2 The module below illustrates how we can determine subjects, objects and actions in the medical system presented in Section 3, through the use of subsorts. Actions are represented as constant symbols for simplicity.

```
fmod MEDICAL-SYSTEM-TERM-SIGNATURE is
         including MEDICAL-SYSTEM-SIGNATURE .
         including POLICY-SIGNATURE .
         subsort Physician < Subject  .
         subsort Patient < Subject  .
         subsort Guardian < Subject  .
         subsort Administrator < Subject .
         subsort Record < Object  .
         subsort MedicalElements < Object .
         subsort OtherElements < Object .
         op readRecord : -> Action  .
         op writeRecord : -> Action  .
         op readMedElements  : -> Action  .
         op writeMedElements  : -> Action  .
         op readOtherElements  : -> Action  .
         op writeOtherElements  : -> Action  .
endfm
```

Sorts and subsorts allows us to distinguish subjects and objects among the subterms appearing in the conjunction of ground terms (through the "+" operator) of the target's application current state. The order-sorted nature of the terms avoids several common mistakes , because type-checking is performed on the specifications.

5 Rewriting-Based Policies

In this section, we address the problem of specifying access control policies through term rewriting systems. We present an initial definition of security policy that characterizes policies syntactically. This definition includes potentially "unsafe" policies. After discussing the desired properties of an access control policies, we present a refined definition for "safe" policies.

A security policy is a statement of what is, and what is not, allowed [5]. When dealing with access control, a (formal) policy specification language will help to unambiguously define the rules that will govern the actions principals are allowed to execute over a set of resources. In order to deal with dynamic policies, a policy specification language has to provide means of encoding high level statements concerning the policy environment, into a function from access requests to authorization decisions.

In the following we present the signature for rewriting-based access control policies. It clearly establishes the functions that need to be defined by the rewrite system in order to compute authorizations. Requests are represented as ground terms containing the 3-tuple: subject, action and object, according to the constructor symbol below:

$$\mathit{req} : \mathit{Subject} \times \mathit{Action} \times \mathit{Object} \rightarrow \mathit{Request}$$

Positive and negative authorizations can be used in the same policy specification with the help of distinct constructors. The signatures for *permit* and *deny* contain subject, action and object, which gives more control on the permissions (or denials) generated from each request evaluation. The profiles for these constructors are shown below:

$$\{\mathit{deny}, \mathit{permit}\} : \mathit{Subject} \times \mathit{Action} \times \mathit{Object} \rightarrow \mathit{Authorization}$$

The goal of the policy designer is to provide a set of rules that will be used by the reference monitor to evaluate every incoming request. The resulting decision does not depend exclusively on the request, but also on the context of the target application at the time the request is made. In the formalism we introduce in this paper, policies consist in a set of rewrite rules defining the *auth* operator, whose signature is

$$\mathit{auth} : \mathit{Request} \times \mathit{Term} \rightarrow \mathit{Authorization}$$

The *auth* function returns a decision term (*permit* or *deny*) derived from the normalization process issued from a request term, and from the information contained in the Term argument, which corresponds to a database of facts. The Term objects can be aggregated in a conjunction of ground terms by the use of the associative and commutative operator, "+". This eases the expression of the conditions where a certain access control rule must be applied. The full policy signature, Σ_P, is illustrated by the code that follows.

```
fmod POLICY–SIGNATURE is
  sort Object .
  sort Subject .
  sort Action .
  sort Term .
  sort Request .
  sort Authorization .
  subsort Subject < Term .
  subsort Object < Term .
  subsort Action < Term .
  op req : Subject Action Object –> Request [ctor] .
  op permit : Subject Action Object –> Authorization [ctor] .
  op deny : Subject Action Object –> Authorization [ctor] .
  op auth : Request Term –> Authorization .
  op _+_ : Term Term –> Term [assoc comm ] .
endfm
```

Definition 5.1 [Rewriting-Based Access Control Policy] A rewriting-based access control security policy, $\mathcal{P}$, is a term rewriting system over $\mathcal{T}(\Sigma, \mathcal{X})$, with $\Sigma = \Sigma_T \cup \Sigma_P$, where the top symbol of the left hand side of each rules is the *auth* function.

This definition states that a given set of rewrite rules is an access control policy if they transform terms rooted by *auth* symbol, which corresponds to the evaluation of an access request in a given environment. The syntactical restriction helps focusing on the actual problem of defining the permissions that should be granted in a given context. Although this definition seems restrictive, in practice rewriting-based policies can work in conjunction with an auxiliary set of rewrite rules to help defining the *auth* function. The following example illustrates how a policy can be declared.

Example 5.2 The following set of rewrite rules translates the natural language rules from Section 3 in our formalism. The rewrite rules appear in the same order as the plain English statements.

```
mod POLICY1 is
    protecting MEDICAL–SYSTEM–TERM–SIGNATURE .
    var p : Patient .          var ph : Physician .
    var g : Guardian .         var adm : Administrator .
    vars s1 s2 : String .      var t : Term .
    var n1 n2 : Nat .          var me : MedicalElements .
    var oe : OtherElements .

    rl [patReadRecord] :
    auth( req(p, readRecord,  record(p, ph, me, oe)),
          p + record(p, ph, me, oe) + t )
     =>  permit(p, readRecord, record(p, ph, me, oe)) .
    rl [guardReadRecord] :
    auth( req( g, readRecord,
                record(patient(s1, n1, n2, g), ph, me, oe )),
           patient(s1, n1, n2, g) +
           record(patient(s1, n1, n2, g), ph, me, oe) +  t)
     => permit(g, readRecord, record(patient(s1, n1, n2, g),
                                     ph, me, oe)) .

    rl [physWriteMedElem] :
    auth( req( ph, writeMedElements, record(p, ph, me, oe)),
           record(p,ph, me, oe) +  t)
    => permit(ph, writeMedElements, record(p, ph, me, oe)) .

    rl [admReadMedElem] :
    auth( req( adm, readMedElements,  record(p, ph, me, oe)),
           adm + record(p,ph, me, oe) + t)
    => deny(adm, readMedElements, record(p, ph, me, oe)) .

    rl [admWriteMedElem] :
    auth( req( adm,writeMedElements, record(p, ph, me, oe)),
           adm + record(p,ph, me, oe) +t  )
    => deny(adm, writeMedElements, record(p, ph, me, oe)) .
endm
```

There are several issues that can complicate policy enforcement. For example, the term rewriting systems implementing a policy can be non-terminating. That would block the target system when it makes certain access requests to the reference monitor, which would not be able to compute an authorization in finite time. We suggest that a refinement discipline for policy specification should be followed, such that verification steps for checking a number of important properties are performed before enforcing some policy. In the next section we discuss what are the desired properties for rewriting-based policies.

5.1 Properties of security policies

Termination

The first interesting property is termination. This ensures that every request evaluation is finite, thus avoiding the target application execution to block indefinitely. Termination of term rewriting systems has been widely studied and there are many tools available that check termination of term rewriting systems such as CiMe, Approve, Cariboo, to mention a few[5].

Readers must be aware that policy combination may lead to problems, since

[5] Check for references and results on the Termination Competition home page: http://www.lri.fr/~marche/termination-competition/

termination is not a modular property [24]. This means that the union of the sets of rules of two terminating term rewriting systems may not produce a terminating one. Most of the positive results on the union of term rewriting systems assume the signature of the composed systems to be disjoint. A survey on modularity of various properties of term rewriting systems is found on [15].

Consistency

The combined use of positive and negative authorizations brings two main problems: incompleteness, when no authorization is specified for a certain request, and inconsistency, when for an access request there are both negative and positive authorizations. Classical approaches for policy specification adopt either the *closed policy* or *open policy* assumption, meaning that only positive or negative authorizations need to be specified, respectively. This has shown to be restrictive in practice. The current trend is to allow the user to discriminate between what is and what is not allowed [10].

Another way to deal with that problem, is to adopt *conflict resolution strategies*, that assign priorities to the conflicting cases. For example, one can say that *deny overrides* any other authorization computed for a certain request. Correspondingly, the *permit overrides* combinator always allow access in case of conflicting decisions. This kind of disambiguation is available in a number of policy specification languages, including XACML [20].

In the case of rewriting-based policies, conflicts are avoided if the corresponding rewriting system has the confluence property. This will ensure that a single response is derived from a given request and from the application current state. In contrast to termination, confluence has a better behavior with respect to the union of two confluent term rewriting systems. It is known that confluence is a modular property of rewrite systems with disjoint signatures [25].

Completeness

Completeness means that for each request corresponds an authorization decision. The usual way of assuring completeness is to assume that the open or closed policy operates as default. In term rewriting, a system is said to be called *sufficiently complete* if all ground terms can be reduced to a normal form that only contains constructors [6,13]. In the case of rewriting-based access control policies, this consists in checking whether the *auth* function is completely defined over the terms of policy environment, and if its normal forms are *permit* or *deny* constructors.

Alternatively, the user can make use of "meta" rules to determine default decisions in the case there are no redexes for a given request and environment. These rules are either of the form

$$auth(req(s_1, a_1, o_1), t) \rightarrow deny(s_1, a_1, o_1)$$

or

$$auth(req(s_1, a_1, o_1), t) \rightarrow permit(s_1, a_1, o_1)$$

for closed or open policy respectively. These rules must be used at the meta level, that is, the standard rewriting mechanism must take into account the fact that these rules must be chosen when no other rule applies.

Given this discussion on the desired properties of an access control security policies, we are ready to introduce the following definition:

Definition 5.3 [Trusted Security Policy] A trusted access control security policy is a *terminating* and *confluent* term rewriting system, $\mathcal{P}$, on the signature $\Sigma = \Sigma_T \cup \Sigma_P$, that *completely* defines the *auth* function.

The word trust was chosen to express that the system administrator can have much more confidence in a rewriting-based policy which has the properties of termination, confluence and sufficient completeness. This ensures that the policy unambiguously states authorizations.

6 Security Mechanisms

In this section we discuss how security mechanisms can implement rewriting-based policies, and how systems can be considered secure with respect to this formalization.

A *Security mechanism* ensures that a target system T does respect the policy being enforced during its whole execution. A state transition of T, $s_i \mapsto s_{i+1}$, corresponds to an access request from a subject to execute an action over a resource. The security mechanism must apply the rewrite rules provided by a policy $\mathcal{P}$, over the terms of T and the current request, $auth(req(s,a,o),t)$. In the case it evaluates to $permit(s,a,o)$, the computation of T can continue, if it evaluates to $deny(s,a,o)$ then the enforcement mechanism must abort the execution of T. This characterizes an *execution monitoring* security mechanism [23].

A given state t_i of a target's execution is considered valid if the information contained in that state is authentic, which means that the database of facts is not modified by an external malicious entity, and that this state was reached through a sequence of positive authorizations.

Definition 6.1 [Secure System] A target system T is said secure w.r.t. a policy $\mathcal{P}$, if it starts from a valid state t_0, and for every transition state $t_i \mapsto t_{i+1}$ a new valid state is produced.

This definition is close to classical automata-based approaches of secure systems, from Goguen and Meseguer [14], and more recently, Scheneider [23], where assertions are stated about the possible execution paths of the target system.

Another advantage of this approach is that the security mechanism can be reused to implement access control for different policy sets, since the separation between policy and mechanism is clear.

7 Related Work

The works more closely related with the one described in this paper are recent initiatives that introduce term rewriting to the specification of security policies.

In [12], term rewriting is used to control the confidentiality level of data, by describing downgrading functions, in a concurrent programming setting, whose formal model is based on a variant of process calculus.

In [2], authors model access control lists and role based access control (RBAC) as term rewrite systems. They characterize consistency, totality, and completeness of policies w.r.t the properties of the rewriting systems defining them, thus sharing some of the goals of this paper. However, we focus on the dynamic aspects of the policy environment to specify authorizations, and how access control rules can be integrated in program development. In general, the main advantage of the approach we present here in comparison with access control lists and RBAC is that it is possible to express more constraints over the policy environment in a flexible way.

Most of the formal approaches based on rules to specify security policies rely on some dialect of a "logic language". We mention [16,11,18], to cite a few. Rewriting has an associated logic [9] and calculus [8], which makes it an appropriate framework for modeling various kinds of systems. The fact that rewriting used pattern matching as its core mechanism makes available a number of important theoretical results that are useful for analyzing security policies, as well.

For example, consider the consistency problem under policy composition. We know that in the case of access control policies based in logics, this property can be preserved by restricting the form of the rules, as discussed in [16]. In the other hand, the rewriting-based approaches can profit from existing results concerning the modularity of the confluence property. Under the same perspective, some necessary conditions under which the termination of term rewriting systems is preserved, are known [22,19]. This makes rewriting an interesting approach for defining policies in a declarative manner.

With respect to enforcement mechanisms, the work presented here follows the line of program monitoring, and considers that the code implementing the application is untrusted. As an example, we mention the Polymer system [3], which enforces access control on Java programs. Rewriting-based policies can be "inlined" as a program monitor for any target application. This can be achieved by relating the policy signature with the signature of the target program, in order to intercept its sensitive function calls. One technique that allows such manipulation is program transformation, that aims to generate new code, preserving the program semantics, a field where term rewriting is widely employed.

8 Conclusions and Future Work

We have discussed in this paper a formalization for access control policies using term rewriting. The resulting language allows us to express access control rules which capture dynamic conditions of the policy environment, providing a flexible way to encode authorizations. Rewrite specifications of access control written according this proposal can be employed to rapidly prototype policies, since several efficient implementations of term rewriting systems are available. Reasoning about some key properties of access control policies is also made possible, thanks to the correspondence we have established between the properties of policies, like consistency, and those of term rewriting systems.

As future work, we shall investigate how rewriting strategies can be useful in policy composition and conflict resolution. An implementation for enforcing rewriting-based policies will also be built, in order to validate this model, by using the monitor inlining approach, as discussed in the previous section.

9 Acknowledgments

I would like to thank my advisors Claude and Hélène Kirchner for the fruitful discussions on this subject, and also Judson Santiago and Horatiu Cirstea for reading previous versions of this paper. I would like to thank the referees for their valuable comments and suggestions.

References

[1] Baader, F. and T. Nipkow, “Term rewriting and all that,” Cambridge University Press, New York, NY, USA, 1998.

[2] Barker, S. and M. Fernández, *Term rewriting for access control.*, in: E. Damiani and P. Liu, editors, *DBSec*, Lecture Notes in Computer Science **4127** (2006), pp. 179–193.

[3] Bauer, L., J. Ligatti and D. Walker, *Composing security policies with polymer.*, in: V. Sarkar and M. W. Hall, editors, *PLDI* (2005), pp. 305–314.

[4] Bell, E. D. and L. J. LaPadula, *Secure computer systems: Mathematical foundations*, Technical Report Mitre Report ESD-TR-73-278 (Vol. I-III), Mitre Corporation (1974).

[5] Bishop, M., “Introduction to Computer Security,” Addison-Wesley Professional, 2004.

[6] Bouhoula, A. and F. Jacquemard, *Automatic verification of sufficient completeness for conditional constrained term rewriting systems*, Technical Report RR-5863, INRIA (2006).
URL http://hal.inria.fr/inria-00070163/en/

[7] Cirstea, H., *Specifying authentication protocols using rewriting and strategies.*, in: I. V. Ramakrishnan, editor, *PADL*, Lecture Notes in Computer Science **1990** (2001), pp. 138–152.

[8] Cirstea, H. and C. Kirchner, *The rewriting calculus — Part I* and *II*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), pp. 427–498.

[9] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: specification and programming in rewriting logic.*, Theor. Comput. Sci. **285** (2002), pp. 187–243.

[10] di Vimercati, S. D. C., P. Samarati and S. Jajodia, *Policies, models, and languages for access control.*, in: S. Bhalla, editor, *DNIS*, Lecture Notes in Computer Science **3433** (2005), pp. 225–237.

[11] Dougherty, D. J., K. Fisler and S. Krishnamurthi, *Specifying and reasoning about dynamic access-control policies.*, in: U. Furbach and N. Shankar, editors, *IJCAR*, Lecture Notes in Computer Science **4130** (2006), pp. 632–646.

[12] Echahed, R. and F. Prost, *Security policy in a declarative style.*, in: P. Barahona and A. P. Felty, editors, *PPDP* (2005), pp. 153–163.
URL http://doi.acm.org/10.1145/1069774.1069789

[13] Gnaedig, I. and H. Kirchner, *Computing constructor forms with non terminating rewrite programs.*, in: A. Bossi and M. J. Maher, editors, *PPDP* (2006), pp. 121–132.

[14] Goguen, J. A. and J. Meseguer, *Security policies and security models.*, in: *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[15] Gramlich, B., *On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems.*, Theor. Comput. Sci. **165** (1996), pp. 97–131.

[16] Halpern, J. Y. and V. Weissman, *Using first-order logic to reason about policies.*, in: *CSFW* (2003), pp. 187–201.

[17] Jacquemard, F., M. Rusinowitch and L. Vigneron, *Compiling and verifying security protocols.*, in: M. Parigot and A. Voronkov, editors, *LPAR*, Lecture Notes in Computer Science **1955** (2000), pp. 131–160.

[18] Jajodia, S., P. Samarati, M. L. Sapino and V. S. Subrahmanian, *Flexible support for multiple access control policies.*, ACM Trans. Database Syst. **26** (2001), pp. 214–260.

[19] Middeldorp, A., *A sufficient condition for the termination of the direct sum of term rewriting systems*, in: *LICS* (1989), pp. 396–401.

[20] Moses, T., *Extensible access control markup language (xacml) version 2.0*, Technical report, OASIS (2005).

[21] Pavlich-Mariscal, J. A., L. Michel and S. A. Demurjian, *A formal enforcement framework for role-based access control using aspect-oriented programming.*, in: L. C. Briand and C. Williams, editors, *MoDELS*, Lecture Notes in Computer Science **3713** (2005), pp. 537–552.

[22] Rusinowitch, M., *On termination of the direct sum of term-rewriting systems.*, Inf. Process. Lett. **26** (1987), pp. 65–70.

[23] Schneider, F. B., *Enforceable security policies.*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.

[24] Toyama, Y., *Counterexamples to termination for the direct sum of term rewriting systems.*, Inf. Process. Lett. **25** (1987), pp. 141–143.

[25] Toyama, Y., *On the church-rosser property for the direct sum of term rewriting systems.*, J. ACM **34** (1987), pp. 128–143.