Weaving Rewrite-Based Access Control Policies

Anderson Santana de Oliveira[*]
INRIA & LORIA
Nancy, France
santana@loria.fr

Eric Ke Wang
The University of Hong Kong,
INRIA & LORIA
Hong Kong
wangke@hku.hk

Claude Kirchner
INRIA & LORIA
Nancy, France
Claude.Kirchner@inria.fr

Hélène Kirchner
INRIA & LORIA
Nancy, France
Helene.Kirchner@inria.fr

ABSTRACT

Access control is a central issue among the overall security goals of information systems. Despite the existence of a vast literature on the subject, it is still very hard to assure the compliance of a large system to a given dynamic access control policy. Based on our previous work on formal islands, we provide in this paper a systematic methodology to weave dynamic, formally specified policies on existing applications using aspect-oriented programming.

To that end, access control policies are formalized using term rewriting systems, allowing us to have an agile, modular, and precise way to specify and to ensure their formal properties. These high-level descriptions are then weaved into the existing code, such that the resulting program implements a safe reference monitor for the specified policy.

For developers, this provides a systematic process to enforce dynamic policies in a modular and flexible way. The level of reuse is improved because policies are independently specified and checked, to be later weaved into various different applications. We implemented the approach on test cases with quite encouraging results.

Categories and Subject Descriptors

D.1 [**PROGRAMMING TECHNIQUES**]: Automatic Programming—*Program transformation*; D.2 [**SOFTWARE ENGINEERING**]: Software/Program Verification—*Formal methods*; D.2 [**SOFTWARE ENGINEERING**]: General—*Protection mechanisms*

General Terms

Security, Theory, Languages, Verification

[*]Supported by CAPES (BEX 2120-03/8)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FMSE'07, November 2, 2007, Fairfax, Virginia, USA.
Copyright 2007 ACM 978-1-59593-887-9/07/0011 ...$5.00.

Keywords

Access Control, Execution Monitoring, Term Rewriting, Strategic Rewriting, Aspect-Oriented Programming.

1. INTRODUCTION

Access control is one of the pillars of secure applications and environments. Its main objective is to unambiguously discern which authorized operations principals can perform on the resources of a system. This can be fairly well achieved when the security conditions do not change over time (see for example the first models of the access control matrix [21] and role-based access control [25]), but it becomes much more difficult when policies depend on their running environment. Some models for access control put forward the need for dynamic access control policies [17, 16, 9], but several problems remain to be solved.

One of them is to model access control policies in such a way that their properties can be formally stated and proved. For instance a policy should be non-ambiguous, it should cover all relevant cases, and when two policies are combined, they should be non-contradictory. In [11, 26] we proposed a formal model of policies, based on term rewriting techniques, which provides several advantages: first, the language allows us to handle a wide range of security policies, because we can easily describe the form of the access requests, and the set of possible authorization decisions, without restricting them to simply *permit* or *deny*. Moreover, policy application can be defined in a precise and expressive way, thanks to strategies that control rule application. This approach provides a clear semantics to access control policies and a framework to formulate the properties to be checked, thanks to properties of confluence, termination or completeness of rewrite systems. We also gave in this framework a semantics for policy composition relying on rewriting strategies, and took advantage of the theoretical and practical results available on modular properties of rewrite systems.

Another important problem is to enforce dynamic policies efficiently and effectively. Currently, the most widespread form of access control mechanism is the reference monitor, that watches the execution of a target program, and interferes with it when it is about to violate the security policy. Such monitors need to be easily analyzed and tested, they need to be complete, and they should not be bypassed.

In this paper, we contribute to solving the second problem

of dynamic policy enforcement while taking advantage of our rewrite-based approach to solving the first one. We design a generic scheme to *weave* rewrite-based security policies into target programs and construct reference monitors to enforce those polices. In addition, we design a methodology to relate the policy description to the program code, which makes explicit the correspondence between the policy environment and the data manipulated by a target program. At the same time, this allows for a complete separation between policies and the target programs, which provides reusable policy modules.

Our solution is implemented through the connection of two specific tools, Tom and AspectJ. Tom is an extension of Java with pattern-matching and rewrite strategies, which provides to the users the capability to describe rewrite systems embedded in Java programs. The second tool is a current implementation of Aspect-Oriented Programming (AOP) [19] for Java. Their combination allows us to inline access control rules as a program monitor, and thus to provide an efficient implementation of a given policy.

The structure of the paper is as follows. In Section 2, we address the design of access-control policies based on a term rewriting approach, leading to a formal description of a controller on which policy properties can be checked. Section 3 shows how aspects can be used to enforce rewrite-based policies on existing code and discuss their impact on the security of the generated program. Section 4 surveys related works, before we reach the conclusion in Section 5.

2. REWRITE-BASED POLICIES

2.1 Term rewriting techniques for policies

An access control policy determines which *actions* the *principals* of a system are (or not) allowed to perform over its *resources*. Several models (for a survey, see [9]) have been proposed to encode access control policies since Lampson [21]. He arranged these three elements (subjects, objects, and privileges) in an access control matrix, where each cell contains the exact privileges of the user in its row, over the resource in its column. These models can be more or less adapted for addressing the need for dynamic policies, which depend on their executing environments, in order to produce decisions about access requests.

Dynamic policies consider the values of attributes of principals and resources, in addition to other conditions, such as time, location, etc, as part of the policy environment. The access control decisions generated by a policy may change according to the current state of a given application.

To illustrate our approach, whose full formal settings can be found in [11], we take as running example a conference management system described in [10]. We changed its original access control rules to be the following set:

1. During the submission phase, an author may submit a paper;
2. During the review phase, a reviewer r may submit a review for paper p if r is assigned to review p;
3. During the meeting phase, a reviewer r can read the scores for paper p if r has submitted a review for p;
4. Reviewers cannot submit reviews or read scores if they are conflicted with a paper;
5. Authors may never read scores.

Typically, the access rule "authors may submit papers during the submission phase" allows access only during a certain period of the execution of the system, while the same access request would be denied as soon as the paper submission deadline has expired.

It is quite natural to formalize such security policies using rule-based formalisms issued either from logic programming as surveyed in [7], or term rewrite systems like [5, 11, 26, 23]. Such rule-based formalisms are often close to natural language expressions and the formal background underlying these rule-based languages allows us to perform reasoning about the policies themselves.

For instance, the first natural language statement of the conference policy can be encoded as a rule as follows:

$$\begin{aligned} &aut(q(author(x), submitPaper, paper(x,z)),\\ &\qquad submission, cnd) \;\;\rightarrow\;\; permit \end{aligned}$$

The left-hand side of the rule represents an access request, which contains information about subject, action, and object, plus other conditions, like the current phase of the conference. In this case, this rule will match any requests where the subject is an author, the action he wants to perform is to submit his paper (the object concerned by the rule) and the current phase is submission, the variable *cnd* will match any extra condition which is not taken into account by this rule. Its right-hand side consists of an access decision, which ranges over the values *permit*, *deny*, and *notApplicable*.

This simple rule is evaluated by transforming the left-hand side expression into the expression on its right-hand side, which is the exact semantics provided by term rewrite systems. This formalism can be easily used for expressing policies, as shown in some recent works [26, 5].

In the rewrite rule above, x and y are variables, from a set of variables $\mathcal{X}$, while the other symbols are functions with some arity, used to build terms. In this paper we consider that terms are typed in the standard many-sorted discipline. If, given an incoming request (input term), we can find an assignment for the variables x and y such that we bind a paper to its author, and all other terms also match, then the request is rewritten into a permission. The "vocabulary" used to build well-sorted terms is called a signature, represented by the symbol $\mathcal{F}$, and the set of all terms over a signature $\mathcal{F}$ using the variables in $\mathcal{X}$ is denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$. When terms do not contain variables, we call them ground, and the set of all ground terms is denoted by $\mathcal{T}(\mathcal{F})$.

A rewrite system consists of a set of rules that are applied successively over the input terms until no rule can be applied anymore. In this case we say that the term is in normal form. This derivation process can be influenced by the use of rewrite strategies, which control rule application in several ways, in particular by associating priorities to rules.

EXAMPLE 1. *The Conference System access control policy is defined by:*

- *A signature $\mathcal{F}$ based on the sorts* Id, Subject, Object, Action, Phase, Condition, Request, and Decision*, where the sorts of the arguments appear inside paren-*

theses:

$$\begin{array}{rcl} paper(Id, Title) & : & Object \\ author(Id), reviewer(Id) & : & Subject \\ submitPaper, readScores & : & Action \\ submission, review, meeting & : & Phase \\ conflict(Subject, Object) & : & Condition \\ q(Subject, Action, Object) & : & Request \\ aut(Request, Phase, Condition) & : & Decision \\ permit, deny, notApplicable & : & Decision \end{array}$$

- *A set of requests answered by the policy which are any terms whose symbol in the top position is aut;*
- *The constants permit, deny, and notApplicable of sort Decision, which represent the possible access decisions for this policy;*
- *A set of rewrite rules over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ presented in Figure 1;*
- *A normalizing strategy, ζ, which computes the normal forms of input terms by the successive application of the rules in R, in this example following the rule order given in Figure 1 (i.e. we repeatedly try to apply r_1 before r_2 ... before r_9 until none is applicable).*

The main advantage of using rewrite systems to specify access control policies is that one can prove important properties about the underlying system which have a direct impact on the trust one can have in a particular policy. For example, the rewrite system presented in Figure 1 is *terminating*, which guarantees that every access request is evaluated in finite time. The system is *confluent* under the given strategy, which assures that there is only one decision for each access request. Other properties are discussed in some previous works [11, 26, 5] which apply term rewriting techniques to access control.

It is worth noticing that the rule r_9 in Figure 1 assures that every request not matched by the left-hand side expressions of the previous rules will be associated to the symbol *notApplicable*, what is particularly suitable for policy composition.

We introduce a general definition of an access control policy in the following. It abstracts the set of possible requests and decisions, and thus supports a wide range of policies:

DEFINITION 1 (SECURITY POLICY [11]). *An access control security policy, $\wp$, is a 5-tuple $(\mathcal{F}, D, R, Q, \zeta)$ where:*

1. *$\mathcal{F}$ is a signature;*
   To define the vocabulary of the data-structure.
2. *D is a non-empty set of closed terms: $D \subseteq \mathcal{T}(\mathcal{F})$;*
   To formalize the set of results allowed by the policy: in our example $\{permit, deny, notApplicable\}$.
3. *R is a set of rewrite rules over $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
   To provide the semantics of the policy.
4. *Q is a set of terms from $\mathcal{T}(\mathcal{F})$: $Q \subseteq \mathcal{T}(\mathcal{F})$;*
   To describe the form of the acceptable requests.
5. *ζ is a rewrite strategy for R;*
   To guide the rule application.

We argue that refinements of the security policy have to precede any deployment. Users must verify that a given policy satisfies the suitable properties (completeness, consistency and termination) before proceeding to its enforcement in a system. For termination, for instance, there exist a number of tools available, such as AProVE [14], which was actually used to verify our running example.

In the next subsection, we present how rewrite systems can be described in Tom.

2.2 Tom: implementing rewrite systems in Java

Tom [2][1] is a language extension which adds strategic rewriting capabilities to Java. A Tom program is the combination of a host program in Java with code fragments delimited by some special purpose constructs to define rewrite systems.

The constructs of the Tom language useful for our purposes are the following ones:

- `%match` corresponds to an extension of `switch/case` construct in functional programming languages, which allows discriminating among a list of patterns.
- `` ` `` (backquote construct) is used to build terms from Java values.
- `%strategy` groups rules to form the basic building blocks for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

In order to transform terms, it is necessary to state a relation between a Tom signature ($\mathcal{F}$ in the corresponding rewrite system) and Java objects. This relation is a mapping, either defined by hand or through an auxiliary tool, called Gom [24], which comes together with the Tom environment. Gom automatically generates the data structure implementation for a given term signature.

The program excerpt in Figure 2 illustrates some of the Tom constructs on a simple example. We encode part of the conference system policy for illustration purposes (the actual full version of the policy is found in Appendix A). The example is divided in two parts. The first part determines the signature for the terms. It is described in the code enclosed between the keywords `%gom` and curly braces (from lines 2 to 14). Since variables are not declared in Tom, we use empty parentheses after constant symbols to distinguish them from variables. The reader may see the symbols of a given sort, for example, the constant `review()` is of sort `Phase`. The function `aut` returns a value of sort `Decision`. In order to build a request, one can use the following expression q_1 = `` `q(author(1),submitPaper(),paper(1,"title")) ``, which contains only ground terms, and corresponds to the demand of an author whose id is `1`, to perform the action `submitPaper()`.

Rewrite rules are defined within the `%strategy` block. For the policy in Figure 2, we gave the rule set a name, `Rules`, which is instantiated in line 24 under an innermost strategy: subterms in more internal positions are reduced first. Rule application is illustrated in line 26, over the input term `q1`, whose result is of type `Decision`.

Figure 2 illustrates how Tom programs are embedded in Java. Following the paradigm of *formal islands* [3], such programs consist of a list of Tom constructs interleaved with

[1] http://tom.loria.fr

$$
\begin{array}{lll}
r_1: & aut(q(author(x), submitPaper, paper(x,z)), submission, cnd) & \rightarrow \; permit \\
r_2: & aut(q(author(x), submitPaper, paper(x,z)), phase, cnd) & \rightarrow \; deny \\
r_3: & aut(q(author(x), readScores, paper(x,y)), phase, cnd) & \rightarrow \; deny \\
r_4: & aut(q(reviewer(x), action, p), phase, conflict(x,p)) & \rightarrow \; deny \\
r_5: & aut(q(reviewer(x), submitReview, paper(y,z)), review, assigned(x, paper(y,z))) & \rightarrow \; permit \\
r_6: & aut(q(reviewer(x), submitReview, paper(y,z)), phase, assigned(x, paper(y,z))) & \rightarrow \; deny \\
r_7: & aut(q(reviewer(x), readScores, paper(y,z)), meeting, assigned(x, paper(y,z))) & \rightarrow \; permit \\
r_8: & aut(q(reviewer(x), action, paper(x,z)), phase, cnd) & \rightarrow \; aut(q(reviewer(x), action, \\
 & paper(x,z)), phase, conflict(x, paper(x,z))) & \\
r_9: & aut(a,b,c) & \rightarrow \; notApplicable
\end{array}
$$

Figure 1: Access rules for our running example. Rule r_8 is recursive, r_9 is the default case.

```
public class Policy {
%gom{
    ...
    Request = q(s:Subject, a:Action, o:Obj)
    Phase = submission()
           | meeting()
           | review()
    Action = submitPaper() | readScores()...
    Decision = permit()
      | deny()
      | aut(r: Request, p:Phase)
    ...
 }
  ...
 %strategy Rules() {
  ...
  aut(q(author(x),submitPaper(),paper(x,y)),
   submission(), cnd) -> {return `permit();}
  ...
  }
  ...
  public static boolean apply(...){
    ...
    Strategy policy =`Innermost(Rules());
    ...
    Decision d = policy.visit(q1);
    ...
  }
  ...
}
```

Figure 2: Structure of a policy in Tom

Java code. During the compilation process, all Tom constructs are translated into Java instructions, in a process that preserves the semantics of the Tom code, as proved in [20].

3. ENFORCING REWRITE-BASED ACCESS CONTROL

We are now ready to address the core problems of rewrite-based policy enforcement through construction of reference monitors.

A reference monitor is a mechanism for enforcing a security policy. It watches over the execution of an untrusted program and is able to interrupt its execution in the imminence of a policy violation. Examples of program monitors include firewalls, operating system kernels, and other components that intercept calls made by target applications. These monitors can be seen as an independent unit in the system architecture, but very often they are *inlined* in the application's code, at compile or load time. In this case, the untrusted code is transformed in such a way that the program's actions are forced to go through the monitor, which decides whether to abort its execution or not, thus avoiding the system from entering an insecure state.

The languages adopted in previous works to describe the policy enforced by a given reference monitor have a rather low level of abstraction, which is comparable to the language used to describe the target program itself [6, 12]. For example, it is possible to express policies that disallow system calls for opening files in the host computer, or to block access to network services after reading data, but it is hard to encode the dynamic behavior of a high-level policy, which takes an authorization decision based on the current conditions in its running environment.

Thus, the model of the policy environment becomes a crucial element for enforcing dynamic policies. In this work, it consists of a collection of values of interest to the policy (determined by the policy signature). Therefore the application's current state is structured as a term, which is modified along its execution. Whenever the application executes an action concerned by the policy, this term is evaluated in order to compute an access decision by applying the rewrite rules that define the policy.

The policy itself does not describe how the system state evolves, but whether the state transitions are allowed or not. This approach is called *execution monitoring*, and is illustrated in Figure 3, which is divided in two parts: the first presents the behavior of a target program alone, the second shows how the transitions are intercepted by a reference monitor enforcing a given policy, which can prevent a transition by aborting the execution of the target.

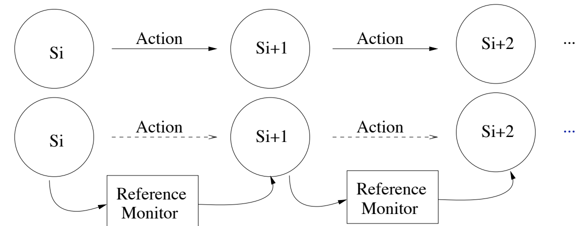

Figure 3: Reference monitoring: transitions correspond to actions

The approach we adopt here is more "collaborative" than the skeptical view taken in previous works [12, 22, 27], since we provide a methodology for the co-development of policy and program, with the choice of suitable mechanisms to enforce the policy. This contrasts with the viewpoint where

there is no confidence at all in the code being executed. Here, policy and program are designed in parallel, with the goal of helping developers to avoid bugs related to access control. The method used to inline a monitor enforcing the policy ensures that it cannot be bypassed.

In the rest of this section, we present in detail our approach for enforcing dynamic policies based on term rewriting, through the use aspect-oriented programming.

3.1 Aspect-Oriented Programming

The concept of Aspect Oriented Programming (AOP) was introduced in [19], as an approach to separate crosscutting concerns, or capabilities which are orthogonal to the system's main functionalities, in single units, called *aspects*. Aspects encapsulate behaviors that affect multiple classes (or units, modules, etc.) in reusable code fragments. Since aspects centralize the code for crosscutting issues, which would be rather spread along the code, in the hierarchical structure of the program, AOP drastically improves the potential reuse of these program artifacts, and decreases the maintenance costs.

In order to give a (classical) example of crosscutting concern, let us consider again a conference management system. In addition to its basic functionalities, such as submission of papers, assignment of papers to program committee members, and submission of reviews, there are other requirements such as logging, to register in a file all past activities performed by users in the system: the code for implementing the system log is scattered in several parts of the system.

The basic building blocks of an aspect-oriented programs are *pointcuts* and and *code advices*. Pointcuts define where a given crosscutting concern should be called. Pointcuts may be associated to a set of function names, for example, and can be expressed through patterns. We call *join points* the locations in the application code that match a given pointcut. Code advice is the actual code executed in join points. In a conference system, for example, a pointcut is any call to the method for submitting papers. The advice code must print to the system log the identity of the current user and some information about the submitted paper, with date and time.

The last essential step in AOP is to *weave* aspects into the main application code. Aspect compilation is a program transformation process that matches the pointcuts defined inside the aspects and inserts the corresponding code advice before, after or around a joinpoint. The expressivity of an aspect-oriented language relies on the pointcut specification, for example, on the kinds of patterns allowed by the aspect language, or on the way code advice can be weaved. To experiment with these concepts, in this paper, we have adopted AspectJ [18] which is becoming a very popular AOP implementation for Java.

3.2 System Architecture

The general view of our approach is given in Figure 4. It highlights the correspondence between the formal description of the policy, through its signature, and the program code. The policy declares a certain number of actions (constants of sort *Action* in the policy signature) which are related to method calls in the source program. These determine sensible methods where requests for access must be made to a reference monitor. The reference monitor is the inlined representation of the policy rules, contained in a piece of code advice that is weaved into the target application code, generating a program that respects the policy.

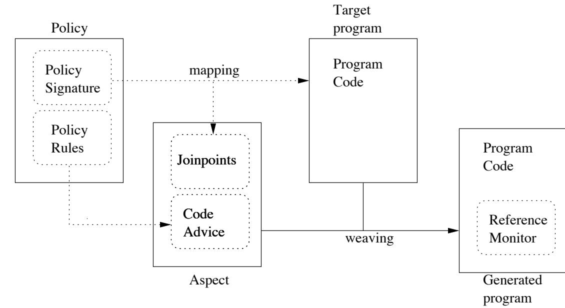

Figure 4: System architecture

In addition to requests, it is necessary to map the other constructors declared in the policy signature. This mapping tells how to interpret the current state of an application as a term. We use constructs provided by the aspect-oriented language to identify where the program objects of interest to the policy are created and/or modified (in AspectJ, these are pointcut designators of kind *field set*) in order to capture their values and build their equivalent term representations.

Therefore, the reliability of the policy enforcement depends on the ability of the policy developer to build a mapping that gives a correct representation from the program values to the policy objects.

3.3 Aspects for Access Control

Capturing the policy environment.

Guided by the construction of the mapping between the policy signature and the objects manipulated by the target application, the application developer must define a set of pointcut designators over these objects to capture changes in their values. The most practical way of controlling this is to declare some auxiliary variables within the aspect to keep the latest values of such objects. For example, let us consider the current user logged in the system. In our running example, we declared a global variable that holds this information from the moment the user is authenticated via his id and password. This is illustrated in the code fragment listed below, which is a piece of the access control aspect for the conference system.

```
aspect PolicyAspect {
...
  private int phase;
...
  after(int cp):
   set(int Conference.currentPhase)
   && args(cp){
     phase=cp;
  }
...
}
```

This code advice is executed every time after the value of the public variable `Conference.currentPhase` is set. The new value is then stored in the aspect variable `phase`, which will later be used to actually build the term representing the policy environment. This process has to be repeated for every piece of data that is part of the policy environment.

This simple example raises some important subtleties. The first is the difficulty in automating the task of capturing the values of variables in dynamic contexts. The task is facilitated by the capabilities provided by AspectJ, but they are not sufficient. For example, it would not be possible to access the value of `currentPhase` from the conference class if it was declared as private. A second difficulty is polymorphism. Sensible calls may present several signatures in Java. Therefore any program transformation approach would need very powerful static analysis to be able to detect automatically all the forms these sensible calls assume. Other security holes are associated to reflective capabilities recently introduced in Java. Provided that the code can transform itself at run-time, the enforcement process has to be able to identify whether it is harmful, according to the policy.

Identifying access requests.

In classical access control models, the notion of operation, or action carried out by the active entities of the system has always been placed as the origin of access requests. In practice, every function call (or method call in the object-oriented paradigm) would imply an access request. The role of the policy is to indicate which of these actions are relevant for access control. In our definition, an access request takes the form `q(s:Subject,a:Action,o:Obj)`. Therefore, for each constant symbol of sort *Action* in the policy signature, we must provide a set of pointcuts which associate the method calls to access requests that concern the policy. For instance, let us consider the program method that allows submitting a review for a paper in the program code, whose interface is shown in the code fragment below:

```
public void submitReview(
   int paperId, int reviewerId, int score)
```

It can be captured by the following pointcut designator in AspectJ:

```
pointcut submitReviewCut(int objId, int revId):
call(void Conference.submitReview(int,int,int))
     && args(objId,revId,*);
```

where we have declared some arguments to keep the data from the paper and reviewer identifiers, information which is used by the code advice. The full code for the access control aspect for the conference system is available in Appendix B.

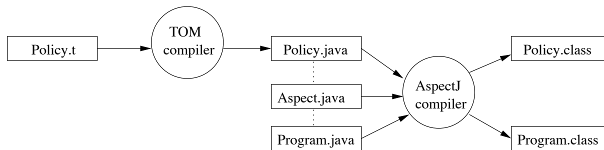

Figure 5: Weaving of access control rules using AspectJ and Tom

Weaving policy rules.

In Figure 5 we illustrate the interactions of the components and tools used in our weaving process. Since the Tom compiler generates Java code, there is no restriction to do the program transformations necessary for access control enforcement by using the facilities available in AspectJ. The conversion from Java objects into Tom terms is performed inside the policy class, implementing the complete mapping between the two representations described in Section 3.2.

In the code advice, the enforcement mechanism must apply the policy rules over the current request term. In order to evaluate the policy environment using a given rule set, we make a call to specific methods in the Java class generated by the Tom compilation process. This design decision avoids the application developers to be aware of the formal representation of the access control policy, which can be taken care of by the security engineer. Finally, the code advice for the `submitReview` action is presented below:

```
before(Paper p):submitPaperCut(p){
 paperId=p.getId();
 if(!Policy.apply(usr.getId(),paperId,
  usr.getRole(),phase,"submitPaper")){
  System.out.println("Access Denied.");
  System.exit(1);
 }
}
```

This advice code states that before any joinpoint determined by the submit paper pointcut designator, a call to the policy is executed whose arguments are the variables declared in the aspect, containing information from the current application context. When this call returns *false*, the reference monitor aborts the execution of the target application. The following code fragment shows how the values collected in the aspect are translated in the policy into their term representation (notice the use of Tom's backquote construct).

```
public static boolean apply(int sId,
  int oId, String role, int phase,
  boolean assign,  String action)
  throws Exception{
  ...
  switch(phase){
   case 0: {ph = `submission();break;}
   case 1: {ph = `review();break;}
   case 2: {ph = `meeting();break;}
  }
  ...
  Decision d = rules(`auth(q(s, a, o),
  ph, asg));
  if(d == `permit())
    return true;
  else
   if(d == `deny())
    return false;
     else throw new Exception();
}
```

3.4 Security and Proofs: towards better trust

Let us analyze the impact of the design decisions we made on the overall security and performance of the generated programs.

Negligence of a policy author with regard to the policy properties can lead to denial-of-service attacks. Let us consider the scenario where a malicious agent discovers that a policy evaluation loops for some kinds of requests. This will be the case if its underlying rewrite system is *weakly terminating*. Then, it is possible to block the application server by flooding it with such requests.

Another important property is completeness. It comes into play in two situations: first, it is necessary to know whether the rewrite system implementing a security policy is able to reduce all request terms into an access decision. This is a particular case of the *sufficient completeness* check in rewrite systems, since we require that normal forms contain

only constructors of a certain kind. However, this additional constraint does not increase the complexity of the problem.

The second situation, where completeness has to be taken into account, is related to the coverage of the application code where access control has to be enforced. Indeed, if for a specific kind of request, the corresponding pointcut designator is left unspecified, the target program will disclose protected data, whenever this missing call is executed. This kind of verification involves the use of polymorphic type systems (for Java, and other languages with this capability).

Other properties can be automated through the use of model-checking. Let us mention for instance the verification of information flow in rewrite-based policies [23]. Covert channels are very hard to detect, but a similar approach can be employed.

Other potential threats to the enforcement mechanism happen in the case policies or aspects are tampered with illegally. Therefore, it is necessary to protect the code with adopting a code signing scheme to verify the integrity and authenticity before running it.

3.5 Performance and Optimization

The main origins of the overhead caused by our enforcement scheme for access control policies are the code generated by Tom for the policy itself, and the overhead introduced by the weaving process made by the AspectJ compiler. Since our rewrite-based security policy language (*ie* full term rewriting) is Turing complete, the policy designer can "easily" add an arbitrary complexity to the existing code. However, our practice shows that policies are quite simple rewrite programs with a low computational complexity. Furthermore, the code generated by Tom is optimized [4]. AspectJ also disposes of some optimization tools [1]. We consider that the practical overhead is small, and that it is worth paying the price for better security mechanisms.

4. RELATED WORKS

Three categories of approaches to enforce security policies are classically identified [15]:

1. *Static analyzers* predict a program's potential harms prior to its execution, discarding those who have unacceptable behavior. An example, is the static type-checking performed by the Java virtual machine that rejects ill-formed bytecode.

2. *Execution monitors* (EM) continuously monitor a program's behavior as it runs, intervening when a coming policy violation is detected.

3. *Program-rewriters* transform untrusted code into self-monitoring code that performs security checks as it executes, thus the target programs are transformed into policy-satisfying ones.

The separations among these three classes are not sharp. Our approach best fits the last category, since we use the AOP technology and Tom as program and policy transformation tools. Actually, program-rewriters are able to combine the power of static analyzes and EM, by accepting untrusted code and well-formed polices as input, and automatically transforming them into policy-adherent and self-monitor code.

Using AOP for policy enforcement is not an original idea in itself. In [28] the authors use a design by contract approach, to separate access control from the other application features at the design level. They explain how to build models based on AOP to design mechanisms to enforce RBAC policies. Another application of AOP is enforcement of availability properties as aspects [8]. Availability requirements are specified in a formal model, combining deontic and temporal logics, and then transformed into aspects. In both approaches policies are written inside the aspect component, thus, in the case policies change, the aspects have to be changed accordingly.

In the second category, we may cite [27], which introduces a formal definition based on finite-state automata to express safety properties (nothing bad ever happens), proved enforceable by execution monitoring. This work was later extended [22] to consider other kinds of security properties.

In the class of program rewriters, let us mention [12, 13, 15, 6]. In all these works, security code is inserted into the executable target. They can assure different levels of trust on the transformation they perform, or on the policy specification.

The main difference between all these works and ours is the power of policies description and representation. In previous works, they are limited to an "event language" that the program monitor can intercept between state transitions, it is powerful but rather low level. In our work, we are able to describe more intelligible policies and relate them to the program code, although this process needs human intervention.

5. CONCLUSION

The need for dynamic policies is predominant in today's information systems. The understanding of how to manage dynamic requirements for policies has augmented significantly with the introduction of several rule-based formalisms for access control [10, 7, 16, 9]. This is due to the fact that rule-based languages can express such kind of policies reasonably well, and they furnish good theoretical background to perform static analysis of possible policy vulnerabilities. In the present work we adopted a recent approach that uses rewrite systems to model dynamic policies [11, 26]. In our model, the correspondences between the properties of the rewrite system and the policy it implements are straightforward, thus we are able to directly apply a rich corpus of existing proof techniques and tools.

On the other hand, there are several problems linked to the enforcement of dynamic policies. The classical approaches of building access control mechanisms by inlining reference monitors into the target application's code does not give much insight on how to capture the policy environment, for instance.

The main contribution of this paper is to introduce a methodology for constructing inlined reference monitors for dynamic access control policies. We showed that aspect-oriented programming is a suitable tool to capture the policy environment and to transform the untrusted code such that it satisfies a given policy. Our methodology is general enough to be applied to various pairs of policy specification and programming languages. Here we presented rewrite-based policies weaved in Java programs, by using Tom and AspectJ. We illustrated the usefulness of the approach on a realistic example where policy and program are completely

separated.

Future works arise from the two sides of our approach. First to provide more fine-tuned analysis tools for policies, such as policy coverage with respect to the target program and administrator queries. Second to perform a systematic analysis of the application to which the security policy should be weaved to, in order to mechanize the weaving process and in particular the construction of the cut and join points.

6. ACKNOWLEDGMENTS

Many thanks the Tom team and in particular to Pierre-Etienne Moreau, for their outstanding work in developing the system, as well as for always fruitful discussions and interactions. We also thank the anonymous reviewers for they valuable comments.

7. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising aspectj. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[2] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[3] E. Balland, C. Kirchner, and P.-E. Moreau. Formal Islands. In M. Johnson and V. Vene, editors, *AMAST, Kuressaare (Estonia)*, volume 4019 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, July 2006.

[4] E. Balland and P.-E. Moreau. Optimizing pattern matching compilation by program transformation. In J.-M. Favre, R. Heckel, and T. Mens, editors, *3rd Workshop on Software Evolution through Transformations (SeTra'06)*. Electronic Communications of EASST, 2006. To appear.

[5] S. Barker and M. Fernández. Term rewriting for access control. In E. Damiani and P. Liu, editors, *DBSec*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

[7] P. A. Bonatti, N. Shahmehri, C. Duma, D. Olmedilla, W. Nejdl, M. Baldoni, C. Baroglio, A. Martelli, V. Patti, P. Coraggio, G. Antoniou, J. Peer, and N. E. Fuchs. Rule-based policy specification: State of the art and future work. Deliverable I2/D1, REWERSE, 2004.

[8] F. Cuppens, N. Cuppens-Boulahia, and T. Ramard. Availability enforcement by obligations and aspects identification. In *ARES*, pages 229–239. IEEE Computer Society, 2006.

[9] S. D. C. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In S. Bhalla, editor, *DNIS*, volume 3433 of *Lecture Notes in Computer Science*, pages 225–237. Springer, 2005.

[10] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[11] D. J. Dougherty, C. Kirchner, H. Kirchner, and A. Santana de Oliveira. Modular access control via strategic rewriting. *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

[12] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[13] D. Evans and A. Twyman, editors. *Flexible Policy-Directed Code Safety, IEEE Symposium on Security and Privacy, 1999*. IEEE Computer Society, 1999.

[14] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In V. van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[15] K. Hamlen. *Security Policy Enforcement By automated Program-Rewriting*. Phd thesis, Cornell University, 2006.

[16] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[17] A. Kalam, R. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization based access control. *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131, 2003.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[20] C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In P. Barahona and A. P. Felty, editors, *PPDP*, pages 187–197. ACM, 2005.

[21] B. Lampson. Protection. ACM Operating Systems Review. *Vol*, 8:18–24, 1974.

[22] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In S. D. C. di Vimercati, P. F. Syverson, and D. Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 2005.

[23] C. Morisset and A. Santana de Oliveira. Automated detection of information leakage in access control. In M. Nesi and R. Treinen, editors, *Preliminary Proceedings of the 2nd International Workshop on Security and Rewriting Techniques (SecReT'07)*, Paris, France, July 2007.

[24] A. Reilles. Canonical abstract syntax trees. *Electr. Notes Theor. Comput. Sci.*, 176:165–179, 2007.

[25] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[26] A. Santana de Oliveira. Rewriting-based access control policies. *Electr. Notes Theor. Comput. Sci.*, 171:59–72, 2007.

[27] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[28] E. Song, R. Reddy, R. B. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control and application features. In E. Ferrari and G.-J. Ahn, editors, *SACMAT*, pages 120–129. ACM, 2005.

APPENDIX

A. THE CONFERENCE POLICY IN TOM

We list here the conference policy as programmed in Tom. The first part of the code, contained in the Gom definition, describes the signature of the rewrite system. The reader may identify some additional sorts w.r.t Definition 1, which are used to describe the policy environment. The second part of the code contains the rewrite rules defining the *aut* function, which relies on the pattern matching provided by Tom.

```
package conference;
import conference.policy.conference.types.*;
import tom.library.sl.*;
import jjtraveler.reflective.VisitableVisitor;
import jjtraveler.Visitable;
import jjtraveler.VisitFailure;

public class Policy{

 %include{sl.tom}

 %gom{
   module Conference

    imports String int

    abstract syntax
      Decision = permit()
      | deny()
      | notApplicable()
      | aut(r: Request, p:Phase, cnd:Condition)

      Request = q(s:Subject, a:Action, o: Obj)

      Phase = submission() | meeting() | review()

      Action = submitPaper()
      | readScores()
      | submitReview()
      | assignPaper()
      | addReviewer()

      Subject = author(id:int)
      | reviewer(id:int)
      | chair(id:int)

      Obj = paper(id:int, title:String)

      Condition = assigned(id:int, o:Obj)
      | conflict(id:int, o:Obj)
 }

 %strategy Rules() extends Fail(){
   visit Decision{

     aut( q(author(x), submitPaper(), paper(x, z)),
          submission(), cnd) -> {return `permit();}

     aut( q(author(x), submitPaper(), paper(x, z)),
          phase, cnd) -> {return `deny();}

     aut(q(author(x), readScores(), paper(x,z)),
         phase, cnd)  -> {return `deny();}

     aut(q(reviewer(x),action, p), phase,
         conflict(x,p)) -> { return `deny(); }

     aut(q(reviewer(x), submitReview(), paper(y,z)),
         review(), assigned(x, paper(y,z)))
     -> {return `permit();}

     aut(q(reviewer(x), submitReview(), paper(y,z)),
         phase, assigned(x, paper(y,z)))
     -> {return `deny();}

     aut(q(reviewer(x), readScores(), paper(y,z)),
         meeting(), assigned(x, paper(y,z)))
     -> {return `permit();}

     aut(q(reviewer(x), action, paper(x,z)),
     phase , cnd ) -> { return
       `aut(q(reviewer(x), action, paper(x,z)),
            phase, conflict(x, paper(x,z))); }

     aut(_,_,_) -> {return `notApplicable();}
   }
 }

 public static boolean apply(int sId, int oId,
     String role, int phase, boolean assign,
     String action){

     Subject s;
     Action a = `readScores();
     Obj o = `paper(oId,"");
     Phase p = `submission();
     Condition cnd = `conflict(sId,o);

     if (role.equals("Author")){
         s = `author(sId);
     }else{
         if (role.equals("Reviewer")){
             s=`reviewer(sId);
         }
         else{
             s=`chair(sId);
         }
     }

     if (assign) cnd = `assigned(sId, o);

     switch(phase){
     case 0: {p = `submission();break; }
     case 1: {p = `review();break;}
     case 2: {p = `meeting();break;}
     }

     if (action.equals("submitPaper")){
         a = `submitPaper();
     }
     else{
         if(action.equals("readScores")){
             a = `readScores();
         }else{
             if (action.equals("submitReview")){
                 a=`submitReview();
             }
         }
     }

     Strategy policy = `Innermost(Rules());
```

```
        try{
          Decision d =
          (Decision) policy.visit(
                       `aut(q(s, a, o), p, cnd));
          if(d == `permit())
           return true;
        else
           return false;
        }catch(Exception e){
           System.err.println(e.getMessage());
        }
        return false;
    }
  }
```

B. ASPECTJ CODE FOR THE CONFERENCE SYSTEM

The following code brings the AspectJ implementation for weaving the conference policy of Example 1 into the conference management system.

```
package conference;
import java.*;
import org.aspectj.lang.*;

aspect PolicyAspect {

  private int phase;
  private User  usr;
  private int  paperId;

  pointcut readScoresCut():
    call(int Paper.readScores(int));

  pointcut submitPaperCut(Paper pa):
    call(void Conference.submitPaper(Paper))
   && args(pa);

  pointcut submitReviewCut(int objId,int revId):
    call(void Conference.submitReview(int,int,int))
   && args(objId,revId,*);

  pointcut loginCut():
    call(void Conference.login(String, int,
                               String ));

  pointcut readScoreCut(): call(
    int Paper.readScore(int));

  after(User user): set(
    User Conference.currentUser)  && args(user){
    usr=user;
  }

  after(int cp): set(int Conference.currentPhase)
    && args(cp){
    phase=cp;
  }

  before(Paper p):submitPaperCut(p){
    paperId=p.getId();
    if(!Policy.apply(usr.getId(),paperId,
        usr.getRole(),phase, true,"submitPaper")){
        System.out.println("Access␣Denied.");
        System.exit(1);
    }
  }

  before(int oi,int ri):submitReviewCut(oi,ri){
    paperId=oi;
    if(!Policy.apply(usr.getId(),paperId,
        usr.getRole(),phase, true,"submitReview")){
        System.out.println("Access␣Denied.");
        System.exit(1);
      }
    }

   before():readScoresCut(){
     if(!Policy.apply(usr.getId(),paperId,
          usr.getRole(),phase, true,"readScores")){
        System.out.println("Access␣Denied.");
        System.exit(1);
      }
    }
  }
```