SPL: An access control language for security policies with complex constraints

Carlos Ribeiro, André Zúquete, Paulo Ferreira and Paulo Guedes
IST / INESC Portugal
E-mail: {Carlos.Ribeiro,Andre.Zuquete,Paulo.Ferreira,Paulo.Guedes}@inesc.pt

Abstract

Most organizations use several security policies to control different systems and data, comprising in this way a global complex policy. These security policies are often scattered over different environments, each one with its own security model, making them difficult to administer and understand. Moreover, some applications (e.g. workflow), often need to cross several of these security domains and satisfy each one of their policies, which is very difficult to accomplish when these policies are scattered over the organization, in conflict with each other and frequently expressed in different models.

This work presents a security policy language that allows organizations to express and keep their global security policies in one single description. Although flexible enough to express simultaneously several types of complex security policies, this language can be easily implemented by an event monitor.

The proposed language can express the concepts of permission and prohibition, and some restricted forms of obligation. We show how to express and implement obligation using the transaction concept, and how to use obligation to express information flow policies together with other complex security policies. We also address the problem of conflicting policies and show how to efficiently enforce the security policies expressed by the language with a security event monitor, including history-based and obligation-based security policies.

1. Introduction

Over the years, several access control policies have been proposed in the literature. Although these policies cover many different situations and data types, they are often considered in isolation, which is not suitable for organizations with complex structures and several data types, which requires the simultaneous use of different access control policies. Moreover, policies are scattered over different environments, which makes understanding and managing of global policies of organizations much more difficult.

The cooperation between MAC (Mandatory Access Control) and DAC (Discretionary Access Control) policies to achieve DAC flexibility and MAC resistance to Trojan Horse attacks, is one of the earliest examples of cooperation between policies. However, there are many other useful forms of cooperation between policies with different objectives.

For instance, an organization may wish to give to each employee the control over the documents they produce but, for payment orders, the organization may want to deny the right to approve them to those who have written the payment orders. This can be accomplished by a DAC policy combined with a simple separation of duty policy.

Another organization may wish to have a very loose policy on most of its departments, in which only a few actions are forbidden, but in one of them have a very strict policy in which only a few actions are allowed. This is a combination of an open policy with a closed one.

Unfortunately, it is not possible to predict which combinations are going to be useful on every situation or even the policies that are going to be needed.

Lately there has been a considerable interest in environments that support multiple and complex access control policies, [3, 4, 11, 12, 19, 22, 24, 27, 39]. Our work follows that path and tries to progress in terms of expressiveness and functionality.

This paper defines a security policy language (SPL), which is flexible enough to express simultaneously several types of complex authorization policies, and nevertheless is constructed with simple elements that can easily be implemented by a security event monitor.

The main goal of SPL is the definition and enforcement of authorization policies. Although recently has been given considerable interest to the joint definition of both authorization and authentication policies within the concept of *trust management* [7, 8] we believe there are considerable gains in keeping both policies apart. Not only because of the simplicity of dealing with each problem in turn but also because the granularity of authentication

needs is usually coarser than the one of authorization, i.e. a user request is often composed of several actions and while each action needs to be authorized, usually only the user request needs to be authenticated.

One of the problems of putting together several policies on the same environment is the conflict that arises from contradictory decisions produced by the different policies. SPL solves this problem by forcing security managers to take priority decisions on every policy composition.

Most multi-policy environments support several forms of discretionary and mandatory access control policies simultaneously. Some support roles and history-based policies like the Chinese wall policy and several other forms of separation of duty. However, to our knowledge, none allows the combination of general information flow policies with other policies. In this paper, we present an information flow policy expressed in SPL, which can not only coexist with other policies but makes use of them. This is achieved through a different type of rule that comprises the obligation concept.

The obligation concept is a very powerful concept to express security policies however it is very difficult to enforce within a security monitor. We show that by using the transaction concept, an access control service based in SPL may enforce some forms of obligation.

We also show how to efficiently implement both obligation and history-based policies, by building an SPL compiler which is able to optimize the information necessary to implement those policies.

The remainder of the paper is organized as follows. Section 2 presents SPL structure and basic blocks (rules, entities, sets and policies). Section 3 shows how to express three special types of constraints: history, obligation and invariant constraints. Section 4 shows some policy examples, including an information flow policy. Section 5 discusses implementation notes and shows performance results. Section 6 discusses related work. Finally, in section 7 we conclude the paper.

2. SPL Structure & Basic Blocks

SPL is a policy-oriented constraint-based language. It is composed of four basic blocks: entities, sets, rules and policies. The fundamental block of the language is the rule. Rules express constraints in terms of relations between entities and sets. Policies are complex constraints that result from the composition of rules and sets into logical units. Policies can also be composed into more complex policies until it forms a global and single policy.

Policies are a key concept of SPL, they provide the structure needed to build complex access control models (e.g. RBAC, DAC, TRBAC). In fact SPL goes beyond the simple enumeration of rules. It allows for the association of rules and sets into policies comprising the logical units of the desired model (see section 4.3).

SPL is therefore model independent but not model less, it allows for the definition of complex RBAC models with parameterized roles [20, 26] and complex role constraints. It also allows for the definition of several forms of multi-level security [1] and relaxed forms of information flow security.

In this section we present in detail each of the basic blocks comprising SPL and show how they are used in writing SPL security policies.

2.1. Entities

SPL entities are typed objects with an explicit interface by which their properties can be queried. Entities can be internal or external to the security service. Queries on the interface of external entities are translated into method or function calls on the objects or services of those entities. Ideally, these queries should not have secondary effects. In practice, this can only be assured by the security service if each of those methods and functions has been verified and annotated as stateless.

Querying external entities is not usually considered safe in security services, due to the covert channels that may result. For instance, an unclassified user can become aware of classified data by executing an action whose acceptability depends on that data and verifying if the action fails or not. Nevertheless, this technique is essential to achieve the flexibility and expressiveness necessary to some systems and applications [16]. To minimize the risk, a SPL policy should be verified before being implemented, to assure that every operation which depends on properties of external entities is allowed only if the query of those properties is allowed. However, it should be noted that this does not prevent implicit flow [13], or time channels.

Some of the entities manipulated by SPL are internal to SPL, like sets and policies, but most are external, like users, files, and events. The properties of each external entity depends heavily on the platform that implements those entities. For example, a user may have just the properties name and home-host, or he can have those and a clearance level, a signature ID and many others. This means that SPL does not restrict the properties of entities to a fixed set, instead it takes advantage of every property available to increase the power of policies.

On many SPL target platforms, the SPL entity set may form a polymorphic hierarchy, where each entity is a specialization of some other entity. In Figure 1, it is shown the entity type hierarchy used in the examples of the next sections. On the root of this hierarchy is the “object” entity type. The remaining entity types are defined by specialization of this base type.

```
type object {
  string name;      // The name of the object
  user owner;       // The owner of the object
  string type;      // A string identifying the type
  object set groups;// The sets containing the object
  string homeHost;  // The host where the user
}                   // is defined

type user extends object {
  rule set userPolicy; // User private policies
}
type operation extends object {
  number ID;           // operation Id
}
type event extends object {
  user author;         // The author of the event
  object target;       // The target of the event
  operation action;    // The performed action
  object set parameter;// The set of parameters
  number time;         // The time instant
  object task;         // The task to which the event
}                      // belongs to
```

Figure 1. Example of an entity type hierarchy definition.

2.2. Sets

Entities can be classified into sets. Sets are essential in any policy considering that they provide the necessary abstraction to achieve compactness, generalization and scalability. Without sets, each rule had to be repeated for each entity to which the rule applies.

```
external string localhost;  // An external entity
external user set AllUsers; // All the users
                            // in the system
external object set AllObjects; // All the objects
external operation set AllActions; // All the actions
external event set AllEvents; // All the events,
                              // past and future
```

Figure 2. Examples of external entities and sets.

Sets, like any other entity, may be internal or external. Some external sets are very useful to the definition of policies. For instance the sets of all users and all objects known to the system (Figure 2).

SPL supports two types of sets: groups and categories. Categories are sets defined by classification of entities according to their properties e.g. all users logged in machine A, and groups are sets defined by explicit insertion and removal of their elements. Insertion and removal of members into a group can only be done by external events since SPL should not perform operations on external or internal entities that result in changes of state. Both categories and groups are declared as sets, but are instantiated differently.

Categories are defined by restricting the elements of other sets to the ones with particular properties. This is done by the SPL restriction operator (*myset@{ logical-expression }*), which is a polymorphic operator that can be used on any type of set or rule (Figure 3)(see also section 2.3, for restriction on rules). The restriction operator has two operands, one is the set that it wants to restrict, and the other is a logical expression that must be satisfied by the elements in the set in order to belong to the restricted set. The logical expression uses properties of the elements in the set to define which members are selected. These properties are written with a dot before the name.

```
// Example of use of the restriction operator
// A category of all users that are defined locally
user set localUsers =
         AllUsers@{.homeHost = localhost };

// A group defined as empty
user set ActiveGroup = {};
```

Figure 3. Example of a category and a group.

SPL defines five more set operators: the index operator ($myset[nth]$), which applied to a set returns the nth element of the set; the membership operator (*element IN myset*); the cardinal operator ($\#myset$) that returns the number of elements in a set; the join operator ($myset1 + myset2$); and the meet operator ($myset1 * myset2$).

2.3. Constraint rules

SPL is a constraint-based language. Constraint languages are widely used to express systems, plans [38] or access control policies [4].

The language is composed of individual rules, which are logical expressions that can take three values: “allow”, “deny”, and “notapply”. Their goal is to decide on the acceptability of each event under the control of the access control service that implements the language. To make this decision, rules have an implicit parameter that represents the event upon which the rule is deciding. Because this event is usually the current event, it is referred as “ce”.

A rule can be simple or composed. A simple rule is comprised of two logical binary expressions, one to establish the domain of applicability and another to decide on the acceptability of the event.

```
[label :] domain-expression :: decide-expression
```

Figure 4. Syntax of a simple SPL rule.

The SPL syntax for a simple rule (Figure 4) has two parts: an optional label; and two logic expressions separated by a special marker (’::’), representing the domain-expression and the decide-expression respectively.

The domain and decide expressions are simple binary expressions with the logic operators ’&’, ’|’ and ’$\sim$’,

respectively for the conjunction, disjunction and negation, the equality/inequality operators '=', '!=', '<', '>', '>=', '=<', and the special values "true" and "false".

The domain-decide construction should not be confused with a simple binary implication. If a binary implication was used, every rule would be implicitly open, i.e. it would allow every event not in the domain, which is contrary to SPL design principle of being a model independent language.

```
// Every event on an object owned by the
// author of the event is allowed
OwnerRule:  ce.target.owner = ce.author ::  true;

// Payment order approvals cannot be done
// by the owner of payment order
DutySep:  ce.target.type = "paymentOrder" &
          ce.action.name = "approve"
       ::  ce.author != ce.target.owner;
```

Figure 5. Simple rule examples.

Figure 5, shows two simple rules, labeled 'OwnerRule' and 'DutySep' respectively. The first one states that events acting on a target object owned by the author of the event (ce.target.owner = ce.author) is always allowed (decide-expression always true). The second rule states that payment order approvals are only allowed if the author is not the owner of the payment order.

The domain-decide type of construction described above is simple, yet it is more powerful than the permission and prohibition construction [23], in which each rule is exclusively a permission or a prohibition. A permission/prohibition rule just identifies the events that are allowed/denied from others. It cannot identify simultaneously the events that are allowed, the events that are denied, and the events that are not allowed or denied. Moreover, a permission or a prohibition can be expressed quite simply with the domain-decide construction by making the decide-expression true or false, respectively, for every event where the domain-expression is true.

α	β	α AND β	α OR β	NOT α
Allow	Allow	Allow	Allow	Deny
Deny	Allow	Deny	Allow	Allow
NotApply	χ	χ	χ	NotApply
Allow	Deny	Deny	Allow	
Deny	Deny	Deny	Deny	
χ	NotApply	χ	χ	

Table 1. Tri-value algebra operations definition: AND, OR and NOT. χ stands for a variable which can assume any value.

A rule can be composed of other rules through a specific tri-value algebra with three logic operators: conjunction ('AND'); disjunction ('OR'); and negation ('NOT'). These operators behave as their binary homonyms if the "notapply" value is not used (with the "allow" and "deny" being equal to "true" and "false", respectively). The primary characteristic of this logic is that the "notapply" value is the neutral element of every operation (Table 1).

```
// Implicit deny rule.
deny:  true ::  false;

// Simple rule conjunction, with default deny value
OwnerRule AND DutySep OR deny;

// DutySep has a higher priority then OwnerRule
DutySep OR (DutySep AND OwnerRule);
```

Figure 6. Composing rules with a tri-value algebra.

This tri-value logic allows some interesting constructs for access control expressiveness. For instance, a default value can be expressed by a special rule in which the domain-expression is always true and the decide-expression is true or false depending on the default value being "allow" or "deny"(Figure 6). Another interesting construction presented in Figure 6, shows how to express priorities between rules. The result of the composition is the result of the "DutySep" rule, except when this rule is not applicable, in which case the result is equal to the result of "OwnerRule".

```
// Universal quantifier syntax
FORALL var IN set { rule_skeleton(var) }

// Existential quantifier syntax
EXIST var IN set { rule_skeleton(var) }
```

Figure 7. Universal and existential quantifiers syntax.

In order to increase the flexibility of composition, SPL defines universal and existential quantifiers, as the tri-value conjunction and disjunction of all the rules resulting from the replacement of the enumeration variable in the rule skeleton, by each value in the set (Figure 7).

```
// Apply all rules in the userPolicy set restricted
// to targets of the same owner
FORALL r IN u.userPolicy {
 r @ { .target.owner = u }
}
```

Figure 8. Example of the restriction operand applied to rules.

Rules do not have to be written at the same time by the same author, in fact they are usually written dynamically

by several authors. Often it is necessary to restrict the domain of applicability of a rule previously written, by the same author or by a different one, without removing it completely. For instance, a rule may state that the private rules of users can only apply to target objects belonging to them. In SPL this is achieved by applying the polymorphic restriction operand (presented in section 2.1) to rules and policies, in order to restrict their domain of applicability (Figure 8). It should be noted that, in this case, the elements in the set being restricted are events.

If $r(D(event), A(event))$ represents a rule or a policy with a domain expression $D(event)$ and an applicability expression $A(event)$, and $R(event)$ is a logical expression on events then the restriction

$$r@R \equiv r'(D(event)\&R(event), A(event)).$$

2.4. Policies

A SPL policy is a group of rules and sets that govern a particular domain of events. Each policy has one "Query Rule" (QR)(identified by a question mark before the name of the rule), that relates all the rules specified in the policy. This rule uses the algebra defined earlier to specify which rules should be enforced and how. The domain of applicability of a policy is the domain of applicability of the QR.

In a SPL policy some of the sets can be parameters that are passed to the policy whenever it is instantiated (or, more correctly, activated). This allows for the construction of several abstract policies, which may be activated several times with different parameters. For instance, it is possible to have a generic DAC policy, a generic separation of duty policy, or a simple generic ACL policy (Figure 9).

```
policy ACL(
 user set AllowUsers, // Users that are allowed to
                      // perform restricted actions
 object set ProtObjects,    // The protected objects
 interface RestrictActions) // The restricted actions
{

?Psimple:
 ce.action IN RestrictActions & // if event action
                                // is restricted
 ce.target IN ProtObjects // and target object
                          // is protected then
 ::ce.author IN AllowUsers// the event is allowed
                          // if the author is allowed
}
```

Figure 9. Generic policy implementing an ACL tuple.

When instantiated, a policy acts as a rule and can be included into another policy by composing it with other rules through the tri-value algebra. As in several object-oriented languages, instantiation is performed by the "new" keyword. Figure 10, shows a security policy ('InvoiceManag') that activates an ACL policy and delegates into it the decision on event acceptability.

```
policy InvoiceManag
{
 // Clerks would usually be a role
 // but for simplicity here it is a group
 user set clerks ;

 // Invoices are all object of type invoice
 object set invoices =
           AllObjects@{ .doctype = "invoice" };

 // In this simple policy clerks can
 // perform every action on invoices
 DoInvoices:  new ACL(clerks, invoices, AllActions);

 ?usingACL: DoInvoices;
}
```

Figure 10. A simple example of policy instantiation.

The ability to compose policies into more complex policies, using the tri-value algebra, is one of the important features of SPL, because it allows for the development of libraries of common security policies. These security policies can then be used as building blocks for more complex security policies, thus simplifying the specification of security policies of complex organizations.

The natural SPL policy sharing mechanism is delegation, but SPL also supports policy inheritance to simplify some sharing situations. For example, defining a policy similar to another policy with just one rule slightly different is much more difficult with delegation than with inheritance. In the example presented in Figure 11 it is defined a policy that extends the "InvoiceManag" policy by restricting the domain of the rule "DoInvoices" to the events with write actions.

```
Policy RestrictInvoiceManag extends InvoiceManag
{
 // Now only write actions are allowed
 DoInvoices:
    super.DoInvoices@{.action.name = "write"};

 // The query rule is inherit from the super
}
```

Figure 11. Example of policy inheritance.

SPL policies are active only if instantiated and inserted into another policy, except for the master policy which is activated implicitly by the security service. The result is a hierarchical tree of active policies with the master policy on top. This structure has several advantages over a flat one [4, 23, 39]. First, it clearly identifies which rules are related with each other, simplifying the global understanding of the policy. Second, it allows the dynamic ac-

tivation and deactivation of policies, by inserting and removing them from other policies. Third, it partially solves the problem of conflicting policies.

2.5. Conflict Solving

SPL supports non-monotonic policies in the sense that it is able to express both positive and negative constraints at the same time. The ability to express non-monotonic policies has long been recognized as very important for the expressibility of security policies [24, 25]. Notably the C2 level of TCSEC standard [14] includes this explicit requirement.

The increased expressibility added by non-monotonicity does not come without cost, it leads to potential conflicts between contradictory rules. Usually these conflicts are solved by the introduction of implicit priority algorithms that choose which rule overrides the other. Some of these algorithms are very simple (e.g. negative rules overrides positive ones) others are more complex and use not only the rules type but also the authority of the rules issuers (i.e. rules issued by a higher authority manager override others), the specificity of the rules (often more specific rules should override more general ones), and the issuing time of the rules (more recently rules override older ones) [2, 25]. This approach is very intuitive and natural but it has some drawbacks. It is not unusual for a high authoritative manager to issue a rule which may be overridden by a low authoritative manager, or to express a mandatory general rule which should not be overridden.

Another strategy is to stratify the security rules and include a special layer of rules to decide which rules should override the others [3, 23]. SPL follows this strategy but instead of creating a special layer of rules to solve conflicts, SPL forces the manager to combine polices into a unique structure which is by definition free of conflicts. In SPL, every active security policy must be in the referred hierarchical delegation tree of policies. Therefore, if two active policies give conflicting results to the same event (one denying it, and the other allowing it), then somewhere up the hierarchical tree they must be combined in one tri-value expression that inherently solves the conflict. If the two policies are combined using a tri-value "AND" then the event is denied. If they are combined using a tri-value "OR" the event is allowed.

However, this solution cannot be applied to every type of security policy inconsistency, because (i) some types of inconsistencies are not conflicts and (ii) some should not be solved by an automated process. For instance, the security conflicts produced by design errors should not be implicitly solved because that would masquerade the design error. In [31] we describe a tool which is able to detect several types of inconsistencies in SPL and can be easily expanded to check for inconsistencies between the secuirty policy and other specifications.

3. Special Constraints

The language described in the previous section can be used to express several types of constraints, including complex constraints that require special implementation considerations. In this section we show how to express and implement with an event monitor, three special types of constraints: history based constraints, obligation constrains and invariant constraints.

3.1. History constraints

Several security policies require events to be recorded, in order to implement constraints with dependencies on the past. Among them, the Chinese wall policy [10] is one of the best known. But many other forms of separation of duty [34] and information flow policies [28] also require event recording.

The importance of history-based polices has been recognized by several authors [15, 32, 40], however to our knowledge none is able to simultaneous express concisely and implement efficiently history-based policies.

In SPL history-based policies are expressed by simple quantification rules over the abstract `PastEvents` set. Each of these rules declares and quantifies one event variable, used to classify each type of past event monitored by the security monitor. Thus in SPL, to monitor a sequence of events it is necessary to cascade several (one for each type of event) quantification rules over the `PastEvents` set. Figure 12 shows a history-based policy which denies any event with an action different from "read" on a target which has been "verified" and "approved" in sequence.

```
policy HistorySequence
{
?HistorySequence:
 FORALL e1 IN PastEvents {
  FORALL e2 IN PastEvents {
   ce.target = e1.target &
   ce.target = e2.target &
   e1.time < e2.time &
   e1.action.name = "verify" &
   e2.action.name = "approve"
   ::  ce.action.name = "read"
  };
 }
}
```

Figure 12. A history-based policy with sequence events.

This approach makes it very simple to express history-based policies based on simple sequences of events, but slightly harder to express history-based policies based on state machines. To express this type of polices it is necessary to define one event variable for each event leaving

each state and write constraints expressing the temporal dependencies between those events. Nevertheless we believe that most history-base policies are of the first type, thus any state machine based approach would be unnecessarily complex.

3.2. Obligation constraints

SPL is able to express the concepts of permission, prohibition and obligation. While the first two are usually supported by access control services, the last one is not. One exception is [12], which defines a modal logic, based on deontic logic to express security policies. However, although it presents a clear definition of obligation it does not propose a solution to implement it by an access control service.

3.2.1. Enforceable obligations

To act upon rules, an access control service must know when there is an attempt to violate them and what to then. On most access control services the violations attempts of rules based on the prohibition concept are detected when an event requesting an action occurs, and in that case, the action requested is denied. With rules based on obligation the time at which a violations attempt occurs (violation attempt time) and the action to perform (default action) when that happens are not so easy to define. First, because a generic obligation (Statement 1) does not need to have a deadline and second because there is no generic action to perform in case of violation attempt.

Principle_O *must do* Action_O (1)

SPL does not allow generic obligations. Instead it supports, with some restrictions, another useful form of obligation that comprises a trigger action (Statement 2).

Principle_O *must do* Action_O
if Principle_T *has done* Action_T (2)

This form of obligation has a much more simple definition for default action then the generic obligation. While with the generic type of obligation a system is in an unstable state until the obligation is fulfilled, with the triggered obligation a system has two stable states, one before the trigger action and one after the obligation is fulfilled. Thus, when the trigger action is executed but the correspondent obligation is not fulfilled the natural default action for the system with this type of obligation is to return to the stable state before the trigger action.

However defining a default action does not by itself solve the problem. Using simple logic it is possible to rewrite statement 2 into statement 3[1],

Principle_T *cannot do* Action_T
if Principle_O *will not do* Action_O (3)

which specifies a constraint with a dependency on a future action. Schneider [36] states that it is not possible to enforce a security policy in which the acceptability of an execution depends on possible future executions, with a monitor like construction. Informally his argument is quite simple. Given the sequences of executions τ' and τ, in which τ is the prefix of some execution of τ', it is not possible to allow τ on the basis that one of its extensions τ' is allowed by the security policy, because the system could stop before τ'.

The key issue is the notion of execution. To Schneider an execution is simultaneously the unit by which the security policy governs the execution of a system and the only atomic unit present in the system. We believe that there are advantages in separating thesetwo concepts. In fact, it is not unusual for atomic requests to be composed of several actions which are themselves subjected to the security policy. Here atomic means in the sense of the transaction ACID properties: either all happens or none happens. Inside these atomic requests it is possible to define security policies with dependencies in future actions, because it is not possible for a system to stop execution before all sequence is completed.

Therefore security policies with dependencies in the future are enforceable but only if they are confined to the bounds of an atomic execution. Thus, in order for a triggered obligation policy to be enforceable it is necessary that the violation time be less or equal to the upper bound of an atomic execution.

Albeit restricted to the bounds of atomic executions, this type of constraint (triggered obligation) is useful in many situations. For instance, the user is obligated to register after it starts using the software, or the information flow policy presented in section 4.5. In fact, in most situations it is possible to find a trigger action for an obligation, however it is not always possible to perform both the trigger action and the obligatory action inside an atomic execution, because some actions cannot be undone, e.g. sending a document to a printer or showing some text on the screen.

These actions are called *real actions* on transaction management systems [21] and are already known to require special treatment by those systems in order to achieve atomicity. Usually the system delays the execution of such actions until all the other actions are executed, but if the action cannot be reordered the system is not able to ensure atomicity. The problem is slightly more complex than in usual transaction management systems because the set of actions identified as *real actions* must include actions that change human knowledge state (e.g. showing

[1] $O \Leftarrow T \equiv \neg T \Leftarrow \neg O$

some text on the screen), which are not often considered.

3.2.2. Expressing obligations

Expressing an obligation constraint in SPL is as simple as expressing a history-based constraint. As was showed in the previous section (§3.2.1) the kind of obligations enforced by SPL can be expressed as constraints with a dependency in the future. Therefore by symmetry with the constraints with dependencies in the past, the natural way to express an obligation constraint in SPL is using quantification rules over a special abstract set `FutureEvents`. As in the symmetric situation, each of those rules declares and quantifies one event variable, used to classify each type of future event monitored. In Figure 13 it is shown an obligation based policy which states that if someone `executes` the `goodies` application he/she must eventually (in the near future) register itself as a user. Another example is present in Figure 20 in which obligation is used to express a relaxed form of information flow policy.

```
policy Register
{
?Register:
 EXIST fe IN FutureEvents {
  ce.action.name = "execute" &
  ce.target = "goodies" ::
  ce.author = fe.author &
  fe.action.name = "register" &
  fe.target.name = "RegisterServer"
  fe.parameters[0] = "goodies"
 };
}
```

Figure 13. An obligation-based policy.

3.3. Invariant constraints

An invariant rule is a very useful type of rule, which specifies that a condition on some object properties should hold before and after every event. These rules are a special type of a more general group of rules that are expressed in terms of results of actions, instead of actions themselves.

SPL is an event-oriented language, in the sense that the goal of each rule is to decide if an event should be allowed or denied, thus invariant rules cannot be expressed directly in SPL, because their goal is not an event in itself but the result of that event. A system with a rule that allows every event, if a condition holds and denies it otherwise, could end up in a deadlock, because the event which caused the condition to be broken was already allowed when the situation is detected. However if the rule states that an event is allowed, if for all the following events the condition holds, and is denied otherwise, the system denies the event that would have broken the condition, preventing it.

A rule expressed as stated, has a similar construction of statement 3, thus it can be expressed and implemented as an “obligation to comply with the invariant condition”.

4. Examples

In this section, we present some security policy examples expressed in SPL to show how SPL copes with different types of security policy paradigms.

4.1. DAC

Although there are many different policies in the DAC category they all share a common base that comprises the essential of DAC. This common base can be implemented by a SPL policy, which can later be used to build several DAC policies.

```
policy DAC
{
 // Owner can do everything to their objects
 authorRule:  ce.target.owner = ce.author ::  true;

 // User policies are applied restricted to their
 // own objects
 userPolicyRule:
  FORALL u IN AllUsers {
   FORALL r IN u.userPolicy {
    r @ { ce.target.owner = u }
  }};

 // The policy denies any event not allowed by any
 // of the rules
 ?DAC: authorRule OR userPolicyRule OR deny;
}
```

Figure 14. An example of a general DAC policy.

A DAC policy has two constraints (Figure 14). The first constraint states that the owner of an object can perform every action on it. The second, states that every user policy should be restricted to the targets owned by the owner of the policy. The QR rule states that if any of these constraints allows an event to happen the event is allowed, otherwise it is forbidden.

SPL can express several types of separation of duty policies. One of the simpler may be implemented with the rule “DutySep” presented in figure 5. This rule states that payment orders cannot be approved by the same users who wrote them.

The combination of the “sepDuty” rule with the DAC policy presented in Figure 15, implements the policy stated in the introduction, where an organization gives to each employee the control over the documents they produce, with the exception of payment orders that cannot be approved by the some user that wrote them.

4.2. ChineseWall

The Chinese wall policy is a monotonic security policy, designed for open systems. Briefly the policy states that

```
policy DAC_SepDuty
{
 // A DAC instantiation
 myDAC: new DAC;

 // Payment order approvals cannot be done
 // by the owner of payment order
 DutySep:  ce.target.type = "paymentOrder" &
           ce.action.name = "approve"
        ::ce.author != ce.target.owner;

 // Events are allowed only if both
 // rules do not deny it
 ?DAC_SepDuty:  myDAC AND DutySep;
}
```

Figure 15. Combination of a DAC policy with a separation of duty policy.

objects are classified into classes of conflicting interests, and a user can access every object, but only one from each class of interest.

There are many ways to write the Chinese wall policy in SPL terms. One of the simpler is presented in Figure 16, in which only one class of interest is defined. The policy defines one set and one rule. The set contains all the objects with the same conflict of interests. The rule states that the current event is denied if the target of the event is in the "interest class" and exists a past event performed by the same user on a different target that belongs to that "interest class".

```
policy ChineseWall
{
object set InterestClass;

?ChineseWall:
 FORALL e IN PastEvents {
  ce.target IN InterestClass &
  e.target IN InterestClass &
  ce.author = e.author &
  ce.target != e.target &
  ::  false
 };
}
```

Figure 16. A specification for the Chinese wall policy.

Usually an organization implementing a Chinese wall policy has several classes of conflicting interests. The above policy has just one class, but can be instantiated several times, one for each class of interest.

The decide-expression of the rule has a constant value, which is consistent with the monotonicity of the Chinese wall definition. This definition specifies the events that must be denied but leaves for complementary policies the decision upon the ones that are accepted.

4.3. Roles

Although they do not always agree on the definition of role [23] most security systems and services support some form of role-based access control (RBAC).

Roles can be very complex entities, comprising constraints on role membership, constraints on role activation, and constraints on role use [18, 35, 37]. To allow all these constraints and eventually others, SPL roles are themselves policies that can be defined as required and used in other policies whenever necessary.

Roles can be composed of several sets and constraints. However, the simpler form of role has only two sets, one with the users that are allowed to play the role and another with the users who are playing the role. Obviously only the users in the first set should be allowed to be inserted in the second set (Figure 17).

```
policy simpleRole (user set Authorized,
                   user set Active)
{
 // Events inserting a user into
 // the Active set are allowed only if
 // that user is in the Authorized set
 ?simpleRole:  ce.action.name = "insert" &
               ce.target = Active
            ::  ce.parameter[1] IN Authorized ;
}
```

Figure 17. The figure represents a simple role policy.

In this model only the users who are in the active set should have the necessary authorizations to play the role. These authorizations are not included in the role type definition, because they are different for each specific role, but they are included in the definition of each specific role.

```
policy Clerk
{
// All users of localhost are members of RoleUsers
user set RoleUsers = AllUsers@{ .host = localhost };

// Invoices are all objects of type invoice
object set Invoices =
           AllObjects@{ .doctype = "invoice"};

// The set of users playing the role starts empty.
user set ActiveGroup = {};

// Members of RoleUsers may play the Clerk role
ClerkRule:  new simpleRole(RoleUsers,ActiveGroup );

// All members of ActiveGroup may access Invoices
InvoiceRule:
   new ACL(ActiveGroup, Invoices, AllActions);

?Clerk:  ClerkRule AND InvoiceRule;
}
```

Figure 18. Example of a specific policy instantiation.

In Figure 18, we show a specific role definition policy, which states that users of "localhost" may assume the "Clerk" role and that every "Clerk" may access invoices. The policy has two rules. One rule ("ClerkRule") is an instantiation of the "simpleRole" policy and defines the components of the role. The other ("InvoiceRule") is an instantiation of the ACL policy and defines the specific authorizations of the role. The link between the two rules is the "ActiveGroup" set, which is simultaneously the set of users playing the role and the set of users allowed to perform actions on invoices.

4.4. Closed and Open policies

Security policies can be open or closed. Closed policies deny everything that is not specifically allowed, and open policies allow everything that is not specifically denied. Closed policies are generally considered safer but open policies are considered more suitable for loose environments, like the ones used in cooperative work [16].

```
// An open policy
?Clerk:  ClerkRule AND InvoiceRule AND allow

// A close policy
?Clerk:  ClerkRule AND InvoiceRule OR deny
```

Figure 19. Different QR to transform the Clerk policy into a closed or an open policy.

The "Clerk" policy defined in figure 18 is neither closed nor open. To be one or the other the domain of applicability must be universal. Hence to make the "Clerk" an open or closed policy all is needed is to modify the QR to allow or deny the events not belonging to the domain of the original policy (Figure 19).

4.5. Information flow policy

Although SPL is a constraint-based language it is possible to express some relaxed forms of information flow policies with it.

As original explained in [13] and formal proved by [36] information flow policies cannot be fully enforced by event monitors, because event monitors do not know about other allowed sequences of executions of the same application and thus they cannot know about implicit flows. Implicit flows result from the knowledge on the sequences of executions allowed by an application. If some application requires that variable Y takes the value a whenever variable X is greater then b then there is a flow of information from X to Y although there is no explicit storage path for information on variable X to variable Y.

However, in some situations [16] the information leak resulting from implicit flow does not poses a serious security risk, whether because the information on variables determining the sequence of execution is public or because it is not possible to infer the sequence of executions from the results of that sequence. For these situations it is possible to define information flow policies enforceable by security monitors, because the regulation of explicit information flow from storage to storage can be performed with just the knowledge of past executions.

Nevertheless, SPL cannot enforce or even express such restricted form of information flow policy based only on information from past executions, because SPL is an event-oriented language and every history information is kept on events it would be necessary to express a policy that recursively verifies each source of every information flow event in the storage-path of the information which are going to be written by the current event. Expressing such a policy in SPL it is not possible both because SPL does not allow recursive policies and because it would incur on a high performance penalty.

To express such information flow policies, SPL uses the obligation concept to force the application to summarized the information flow into the existing SPL rules. The policy in figure 20 states that each object which receives information from another object, should be subjected to the same set of rules as the originator object. This is achieved by an obligation rule that forces the receiving object to belong to the same groups of the originator object.

```
policy InfoFlow ()
{

 interface ReadFlowActions;
 interface WriteFlowActions;
 object set ProtObjects;

 ?InfoFlow:
  FORALL pe IN PastEvents {
   FORALL g IN pe.target.groups {
    EXISTS fe IN FutureEvents {
     ce.action IN WriteFlowActions &

     pe.target IN ProtObjects &
     pe.action IN ReadFlowActions &

     ce.task = pe.task &

     ::  ce.target IN g
  } } };
}
```

Figure 20. An information flow policy.

The `?infoFlow` rule of figure 20 traces indirect information flow between read and write events performed by the same task. The rule states that if the action of the current event is a write action and the current task has read a protected object (i.e. one of the events that has read a protected object were done in the context of the task of the current event), then there is a time in the future (i.e. a future event exists) in which all the sets containing the

protected object also contain the target object. This rule assures that every rule that applies to a protected object which was read by that task also applies to the receiving object, including the rule itself, i.e. the receiving object becomes a protected object too.

It should be noted that it is the application obligation to ensure that all the sets are updated properly. The application can perform this task either by itself or by using a security library created for that purpose. The security monitor duty is to ensure that that obligation is fulfilled. The security monitor cannot update itself the sets because it cannot perform operations which result in state changes.

Unlike other models where information flow policies are defined, the SPL model allows non-monotonic policies. This property may produce an uncommon result on some information flow policies. When there is a permission rule that supersedes a prohibition rule, an object to which the access was restricted may become unrestricted just because it received information from an unrestricted object. Although uncommon, the result is correct because denying may not always be the safer action. For example, an organization may state that the president should be able to access every document containing organization's classified data, but he does not need to be able to access employer's private data. Then if an employer includes classified data into a private document, that document should become accessible to the president.

5. Implementation and Results

One of the problems of expressive security frameworks like SPL, is the low efficiency of their implementations. While usual frameworks built upon access control lists, labels or unix permission bits were designed to be efficient, SPL was designed to be expressive.

In this section we show that using a mixture of compilation and query techniques it is possible to achieve acceptable performance results, even for policies with thousands of rules. We have designed and implemented a compiler for SPL (which generates standard java), which is able to detect special SPL constructions and generate the most efficient code to implement them.

Given the resemblance of SPL structure with java structure most of the compiler actions are simple translations: each SPL policy is directly translated into a java class; each rule is translated into a tri-value function without parameters (with the exception of the query rule which has one parameter – the current event); each entity is translated into a java interface; and each set variable is translated into a java variable of type `SplSet`, which defines an interface to access several kinds of sets (external sets, subsets of external sets, internal sets).

As defined in §2.3, rules can be simple rules comprised of a domain-expression and a decide-expression or they can be a composition of other rules. Thus functions implementing rules can be from simple if-clauses with two logical expressions (one for the domain- and another for the decide-expression) to complex combinations of other functions (e.g. simple combination using tri-value operators; quantification of rules over sets; quantification over history events; quantification over future events).

Wherever a policy instance is used in place of a rule the compiler executes an automatic cast operation, consisting in making explicit the call to the query rule of the policy. Thus the overall structure of the generated code can be seen as a tree of tri-value functions calling other functions, in which the root is the function resulting from the translation of the query rule of the master policy and the leaves are the functions resulting from simple rules.

Although most SPL constructs can be efficiently implemented in java by direct translation, some constructs and structural problems require a deeper analysis. In the remaining of this section we address those problems, and show some performance measures that validate the solutions.

5.1. Scalability

One of SPL major design problems is scalability. While in common ACL based systems, only the access control entries (ACE) belonging to the ACL of each target object are evaluated on each access, in SPL potentially every rule has to be evaluated for every access. This is a problem on systems with thousands of rules, users and objects.

SPL is a logical-based language, thus it is possible to apply some evaluation optimizations. In a conjunction of rules (tri-value conjunction as defined in §2.3) if one rule evaluates to "deny" then it is not necessary to evaluate the remaining rules (similar for disjunction of rules and "allow" values). Unfortunately these optimizations are not very useful, because disjunction of rules are rare and the optimization applicable to conjunctions can only optimize the deniable of events.

Another more useful optimization can be applied to the restriction operation (*expression(event)@rule*). The "restriction operation" restricts the domain of applicability of a rule to the set of events satisfying a logical expression. Thus if that expression evaluates to "false" it is not necessary to evaluate the rule. This optimization is very useful on those situations where rules are explicitly organized in domains of applicability (e.g. rules that apply only to targets produced by one branch of an organization). However it is not enough to prevent the unnecessary evaluation of not applicable rules inside the same domain. Wherever the restriction operation is not used, to reach the conclusion that one branch of the evaluation tree is not applicable to a particular event it is necessary to evaluate each domain expression of every leaf rule in that branch.

One solution would be to build a virtual restriction operation in which the restriction expression would be the logical disjunction of each domain expression[2] of every leaf rule in the branch. Although very efficient in detecting not applicable branches this solution penalizes applicable branches with redundant evaluation of domain expressions in each node of the evaluation tree.

The solution used in SPL is based on the assumption that most expressed rules are target-limited, in the sense that they are applied to only a limited set of targets. SPL is able to express rules not target-limited (e.g. all actions performed by some user), nevertheless we believe that most security policies expressed in SPL will be target-limited. This assumption is based on the observation that most current security policies are target-limited, e.g. all ACL based policies, chinese wall policies, DAC policies. RBAC is not target-limited but is used in conjunction with rules which are target-limited.

Based on this assumption we have designed a simple target-based index for rules, which allows for quick cuts on branches of the rule evaluation tree. The system creates an index for each target. Each index is maintained on the correspondent target as a label and keeps the information of every rule that may be applicable to an event with that target. The representation of that information on the current prototype is kept on a bit stream with one bit for each rule in the system. However given the sparse nature of the information (we expect that only a few rules are applicable to each target as in current ACL based systems) it is possible to develop more compact structures.

On the tests done so far this index technique has proved to be efficient, showing on average a speed-up of one order of magnitude (see 5.4).

5.2. History-based policies

A monitor-like security service has to decide for each event whether it should allow the event to happen or not. The decision must be taken at the time the event is requested with the information available at that time. Thus is order to implement history-based polices any monitor-like security service has to record information about past events.

Some security services record events implicitly in their own data structures [28] (mostly using labels) others record them explicitly into an event log [4], which can later be queried for specific events. The later solution is more flexible than the former but if the event log becomes too big, the memory space required to keep that log may be unlimited and the time required to execute each query could have a significant impact on the performance of the system.

In this section we show that it is possible to implement efficiently the log solution, both in terms of memory-space and performance. The main achievement is obtained by a compiling algorithm that optimizes the amount of information to be saved and the way that information should be queried. We show that although this algorithm does not obtain the best results for all history-based policies, the results obtained for most common policies are equivalent to those obtained by label-based implementations [33].

The goal of this algorithm is three-folded. First, the security manager should selectively log just the events required by the history-based policies specified, e.g. if a policy needs to know if a document was signed, there is no need to record events that are not "sign events". Second, the security manager should selectively log just the fields of the events required by the history policies specified, e.g. policy wants to decide based on whether or not the author of the current event has signed a document, it is not necessary to record the "parameters" field of signature events. Third, security manager should use the best possible query for each history-based policy (equality terms can be searched in $O(1)$ and are preferred to inequality terms) and the best information structure to support that query (a hash table is preferred for an equality search but for an inequality search a balanced tree might be better).

Instead of building a log for every history-based policies the compiler builds a specific and fined tuned log for each history-based policy. This solution has several advantages. First it divides the problem reducing the number of events required to be searched. Second it allows for a better adaptation of the base structure to each query, because each log can be kept by a different structure. And third it simplifies insertion and removal of policies. The problem of this solution is the potential for maintaining redundant information in several logs. However, given that the information kept by each log is the minimum information necessary to that policy, the level of redundancy expected is similar to the level of redundancy of label-based implementations, where the labels used by different policies may also be redundant. Nevertheless, this problem can be further reduced by sharing logs with the same signature (same events to log, same fields of those events to log, same base structure) between policies [3].

Figure 21 shows a simplified version of the code generated by the compilation of a history-based rule. The expression `MyRule(e, ce)` represents a generic rule that may be composed of other rules.

The algorithm has four phases. The first phase is just the removal of the invariant conditionals from the loop. In this phase the compiler tries to build a logical expression (referred as "invariantConditionals" in Figure 21) with terms from the domain expression of `MyRule` which are manda-

[2] Obviously a reduced canonical form.

[3] This feature is not implemented in the current prototype.

```
MyPolicy: FORALL e IN PastEvents MyRule(e, ce)
```

(a)

```
triVal MyPolicy(event ce) {
  if( !invariantConditionals(ce) ) return notapply
  while( MySpecialLog.hasMoreElements(ce) )
    x = stripped_MyRule( MySpecialLog.next(ce) );
    if( x == deny ) return deny;
    if( x == allow ) return markallow;
  }
  if(!markallow) return notapply
}

class MySpecialLog {
  HashTable Log;
  void insert(event e) {
    if( PastDependentTermsOfRule(e) )
      log.insert(new RequiredFieldsOf(e))
  }
  boolean hasMoreElements(event ce) {
    return log.find(new indexFieldsOf(ce);
  }
  RequiredFieldsOf nextElement(ce) {
    return log.next(new indexFieldsOf(ce));
  }
}
```

(b)

Figure 21. Translation of history rules. (a) is the SPL representation of a generic history-based rule. (b) is a simplified version of the java code resulted from compilation

tory for the applicability of the rule and are not dependent of variable e. This expression is then used to perform a preliminary test of applicability of the rule the current event.

The second phase also builds a logical expression with terms from the domain expression of `MyRule`, but with terms dependent on variable e and not dependent on current event. The goal of this expression ("PastDependentTermsOfMyRule") is to filter the events that need to go into the log.

The third and fourth phases build respectively one class object with the fields of variable e used in `MyRule` (referred as "RequiredFieldsOf") and another class objects with the fields of e which compared with logical expressions dependent on the current event. The former is used to record only the information on past events which are useful to the security policy. The later is used to search the log for events with those fields equal to the ones in the object [4].

The main drawback of this algorithm is that history-based policies cannot decide on events prior to their activatio, i.e. the system only records events for each history-based policy after the policy exists.

To illustrate the algorithm results we will show how a SPL policy expressing the Chinese wall policy (Figure 16) is enforced by a monitor generated by the SPL compiler.

The first and second phase of the algorithm tries to identify logical expressions built form terms of the domain expression which are mandatory true for the applicability of the policy. In this policy (Figure 16) the domain expression is composed by a conjunction of simple terms. Thus any term can be used independently for the construction of those expressions. The problem is more complex when the domain expression is composed of both conjunctions and disjunctions, in which case may not always be possible to completely unfold the logical expression [5].

For the Chinese wall policy the "invariantConditionals" logical expression is composed of just one term `ce.target IN InterestClass`, thus according with figure 21 the policy returns "notapply" if the target of current event is not in the class of interest, which is conformable with the expected behavior. The "PastDependentTermsOfRule" logical expression is also composed by only one term `e.target IN InterestClass`, thus only the events over objects in the interest class are logged.

The "RequiredFieldsOf" object for this Chinese wall policy is composed by the "author" and "target" fields of the "event" class object, and the "indexFieldsOf" object is composed of just the "author" field. Thus the log just keeps information about the target and the author of each recorded event and it is queried by events with a specific author.

Because the log does not have to keep repetitions, and the specific nature of the Chinese wall policy disallows the existence of more than one element with the same author, the maximum length of the log is the number of different users in the system. Usually the length of the log is much less then the number of users, because not every user access a target in the "interest class". However, if the length of the table supporting the log is equal to the number of users, then the query can be performed by direct addressing the user field, followed by a comparison of the target field.

This is much similar to the classic label implementation [33] where each user has one label for each interest class, which contains *nil* if the user did not access any target in the "interest class" or the identification of the target accessed. However, the described implementation results from the "compilation" of a language which is able

[4] The current compiler prototype can only implement logs with hash tables, thus it does not handle effectively policies where at least one field of e is not equally compared with a logical expression dependent on the current event.

[5] This is usually the case when several rules are combined with tri-logical operations, because the overall domain-expression is the disjunction of the domain-expression of each rule and the domain-expression of each basic rule is usually a conjunction of terms

to express simultaneously several other policies, including other history dependent policies, while the classic label implementation is hardcoded in the user management structures.

This technique can be applied to other history-based policies which are usually implemented with labels. The reason why these policies can be implemented efficiently by a SPL compiler lies on its ability to keep their relevant history information in small pieces of data (the labels), directly addressed by one entity (users, objects, etc.). Therefore, a SPL compiler which is able to detect exactly which history information is relevant to the policy and is able to index the resulting table by the most appropriate entity (or entity property) can achieve similar efficiency results as label-based implementations.

5.3. Obligation-based policies

As explained in §3.2 the obligation-based security policies enforceable by event monitors are only the ones that can be completely resolved inside an atomic execution. The monitor generated by the SPL compiler does not provide code to make those sequences of actions to behave atomically, instead it relies on applications to define those sequences of actions and on a transaction monitor to implement it. Thus the problem of enforcing obligation-based security policies is reduced to allowing or not the event that instruct the transaction monitor to *commit* a transaction, whether or not all the obligations were fulfilled at the time of that event.

A security policy that allows or denies an event (the commit event) depending on whether or not some events were executed (the obligations) is a history-based policy. Thus the enforcement of an obligation-based policy controlling a particular type of event can be done by a history-based policy controlling the event that commits the transaction on which the original event was executed.

The transformation from the obligation-based policy to the history-based policy can be achieved in two steps. The first step called "aging" consists of replacing references to events by older references. References to the current event are replaced by references to a past event called "trigger-event". References to past event are replaced by references to a past event but with an additional constraint specifying that this event occurs before the trigger-event. References to future events are replaced by references to past events with the additional constraint of occurring after the trigger-event. The second step consists of inserting in this policy an explicit reference to the event that requests the transaction-commit. This event becomes the current event of new policy and is related with the trigger-event by means of the transaction id in which the trigger-event was performed.

Figure 22 shows the history-based version of the obligation-based policy shown in figure 20. In the current prototype this transformation is mixed with the translation to java, thus the SPL representation of history-based versions of obligation-based policies never take place.

```
policy HistoryInfoFlow ()
{
 interface ReadFlowActions;
 interface WriteFlowActions;
 object set ProtObjects;

 ?InfoFlow:
  FORALL te IN PastEvents {
   FORALL pe IN PastEvents {
    FORALL g IN pe.target.groups {
     EXISTS fe IN PastEvents {
      ce.action.name = "commit" &        // New

      fe.time > te.time &                // New

      te.transaction = ce.parameter[0] & // New
      te.action IN WriteFlowActions &

      pe.time < te.time &                // New
      pe.target IN ProtObjects &
      pe.action IN ReadFlowActions &
      pe.task = te.task &
      ::  te.target IN g
  } } };
}
```

Figure 22. The transformation of the information flow policy of figure 20 into a history-based policy.

5.4. Results

Access control monitors are used in several environments. Although they are used as services which are seldom queried by other services [29]. they are also used at the center of systems being queried by every element in the system for almost every action, thus their performance has an important impact on the overall performance of the system. All measurements presented in this section were taken on a personal computer with a Pentium II at 333MHz running the Sun Java 1.2.2 virtual machine over Windows NT 4.0.

The performance of an access control monitor is measured by the time it takes to respond to a query. However more important than knowing the absolute value of time taken by the monitor to solve a query, which varies with the platform and the intermediate compiler used, is the dynamic behavior of the monitor with policy and log scalability, i.e. "How is the query delay affected by the number of queries answered on history-based policies ?" or "How does the query delay evolves with the size of the policy ?".

To answer the first question we have developed a test based on the Chinese Wall policy. In this test we measured the time to solve a query for the acceptability of events produced by 100 different users by a monitor enforcing

a Chinese Wall policy with 10 interest classes, with 10 objects per class. The time for each query was taken each 100 events to verify the effect of event logging over the query performance.

The events were chosen such that their targets would always be in one class of interest and that the expected answer to the query would always be positive ("allow"). This is the most common behavior (in normal systems most actions are allowed) and unfortunately it is also the worst case for this and most policies expressed in SPL. This behavior is shared by every policy which uses conjunctions as their predominant composition construction. In this situation the Chinese Wall policy is composed by a conjunction of 10 policies showed in Figure 16, one for each interest class. If one of those ten policies denies an event then there is no need to evaluate the remaining policies. However, for events which are allowed all the policies are evaluated.

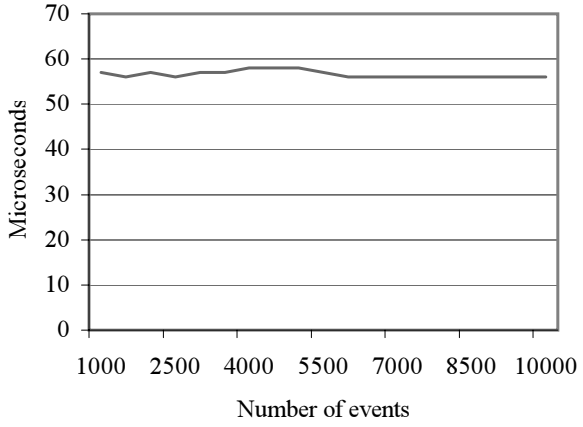

Figure 23. Chinese Wall dependency with the number of events queried

The results presented in Figure 23 show that the time taken to solve a query to the Chinese Wall policy does not depend on the number of events queried, thus proving that the solution used to minimize the impact of logging on the overall performance of the monitor is effective.

The time needed to solve a query to the Chinese Wall policy is also not affected by the number of users or the number of objects in each class of interest. But it is severally affected by the number of classes of interest (Figure 24). This result is a direct consequence of the number of rules used to build the Chinese Wall policy with different numbers of classes of interest. The Chinese Wall defined in Figure 16 requires the definition of one rule for each class of interest, thus for Chinese Wall policies with more classes of interest the monitor needs to evaluate more rules for each query.

The index solution presented in §5.1 can minimize the

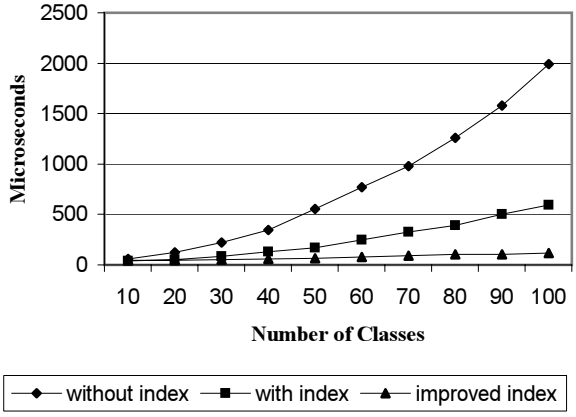

Figure 24. Chinese Wall scalability with the number of classes of interest.

problem as shown by Figure 24. However it is not enough for policies like the Chinese Wall or any other policy with one single large conjunction of rules. On these policies the index effectiveness is small because the branches in the evaluation tree of those policies are small. Thus the cuts which the index is able to perform are necessarily small. These type of policies require better indexes. For instance, indexes with several layers of indexes over indexes. This solution is not implemenetd in the current prototype but its effect can be measured because it whould be similar to rearrange the policies in order to have a deeper evaluation tree. For instance, the big conjuntion of rules of the Chinese Wall policy can be rearranged into a conjuntion of conjuntions using the associative property of conjuntions. The results of Figure 24 shows the effectiveness of such approach.

	Non indexed μs	Indexed	
		normal (μs)	optimized (μs)
ChineseWall 10	56	1.5 (37)	1.3 (43)
ChineseWall 100	1992	3.3 (597)	31.6 (63)
Global	303	7.6 (40)	-

Table 2. Speedup results, with respect to non indexed queries, for queries with normal index and with index over rearranged policies.

Table 2 shows some examples of index effectiveness. Two Chinese Wall policies with 10 and 100 classes of interest, were tested without index, with index and with index over rearranged forms of the policies. Although the index speedup is not very sensible for the Chinese Wall policy with 10 classes of interest it becomes important for the Chinese Wall with 100 classes of interest. The

other example shown in Table 2 is the "Global" policy described in appendix A, which is a policy with 4120[6] rules divided into 100 domains over 5 continents. Albeit simple for a real policy of an organization this policy is complex enough to represent the target policies of SPL. The policy was tested for 5000 users and 12000 targets, exhibiting a $40\mu s$ delay for each query, which is an acceptable value for the size of the policy and the underlying platform used.

6. Related Work

Much work has been done on multi-policy environments primarily to solve the conflict raised by having different policies governing the same subject. Some of this work tries to solve the problem using specific system mechanisms [11, 19], but most define unified frameworks in which different policies can be expressed [3, 4, 12, 22, 24, 27, 39].

Minsky and Ungureanu [27] define a formalism and an environment to specify and enforce security policies in distributed systems. Their environment assumes a message monitor that intercepts every message sent or received, and runs security policies. A security policy is composed of a set of clauses. Each one defining the actions that the monitor should take on intercepting each message. The authors show that the formalism is powerful enough to express complex policies, but it is not clear how they deal with conflicting policies.

Woo and Lam [39] show how default logic can be used to express authorization rules. Roughly, each rule is composed by three binary formulas (g, f, f'): formula g defines the actions allowed by the rule; formula f defines the actions that must be allowed by other rules in order for this rule to be active; and formula f' defines the actions that must not be allowed by other rules. This construction is very powerful for relating rules with each other, producing very expressive policies. Nevertheless, we believe that the algebra for security rules proposed by us is able to express most security policies using simpler and more compact rules.

Adage [4] authorization rules are very similar to our own. They both have a domain of applicability function and a decision function. However, Adage does not specify an algebra for rules and polices, which makes them much more difficult to compose into complex policies and to express default behavior. Furthermore, their work does not provide a conflict resolution mechanism or an efficient implementation of history-based policies [40].

Conflict resolution approaches defined by Bertino et al [3] and Jajodia et al [23] have some similarities. In [3], rules are classified into two categories: strong and weak. The strong rules may override weak rules, but not other strong rules. This means that conflicts may still arise between strong rules and have to be solved by other means.

Jajodia et al [23] define a logical language with ten predicate symbols. Three of those are authorization predicates (dercando, cando, do), used to define the allowed actions. Although not explicit these predicates define three levels of authorization with dercando as the weakest and do as the strongest. However the problem remains, because conflicts may still arise between "do" rules [24].

Another approach to conflict resolution, presented in [2] and [25] uses elements like rule authorship authority, rule specificity and rule recency to prioritize rules. Although simple and natural this approach may lead to undesired behavior. It is not uncommon for high authority manager to issue a rule which may be overridden by a low authority manager, or to express a mandatory general rule which should not be overridden.

In [9, 8] Blaze at al proposed a different concept called "Trust Management". Their work starts by identifying that in the services that receive signed requests, the principal issuing a request is the key that signs the request. Thus if the policy maintained by the authorization service is organized in terms of keys instead of names (user names, role names, service name, etc.) it is not necessary to perform the extra step of checking the authenticity of the request.

They propose a tool, the *PolicyMaker trust management system*, which is able to express in a single common language authorization policies, certificates and trust relationships, thus integrating whole these concepts.

The trust policy engine replies to a request based on the local policy and trust assertions and on the certificates provided by the requester. The engine checks if the request is authorized by the local policy assertions, or if there is a path of trust assertions from a local assertion to a key signing a policy certificate that allows the request. This solution clearly scales better than a global static policy.

KeyNote [6]and SPKI [17] are two other examples of systems comprising the notion of trust management. KeyNote derives from PolicyMaker and shares the same principals. However, KeyNote was designed to simplify the integration of the service with the client applications. Thus KeyNote has a built-in credential verification system and a simple notation to express authorization predicates. SPKI (Simple Public Key Infrastructure) on the other hand, results from the extension of the certificates kept by a Public Key Infrastructure to allow authorization certificates. Although, slightly more restrictive than KeyNote, SPKI shares the same fundamental features of KeyNote and PolicyMaker. SPKI also (i) uses Keys as principals, (ii) allows trust to be delegated from one key to another (iii) allows policies to be inserted dynamically

[6]Only 1690 rules can be directly counted from SPL specification. The remaining rules are inserted into the "userPolicy" group of each user.

in the form of certificate.

Although with similar results each of these trust-management systems have a different compliance checking engine. The one from PolicyMaker is the most general, in the sense that it can use arbitrary functions to express assertions, provided that they are monotonic. On the other hand the compliance checking engine from SPKI allows a limited type of assertions but it allows negative assertions.

In [5] Blaze at al. shows that checking the compliance for the general problem is NP-hard and gives several alternatives with different levels of expressiveness and usability. However, the best balance between expressiveness and usability is still an open issue. We believe that the SPL's compliance checking engine is a fair alternative, because although it is harder to dynamically insert policies (policies can only be added at specific points) it has a good performance, and does not compromise in any way expressiveness.

Most of these environments can state both positive and negative authorization rules. In [12] it is showed that obligation can also be a very powerful concept to express security policies, however it is not clear how can it be implemented.

Although expressive enough to handle most of the usual policies, including the ones with history dependence, like the Chinese wall and several other separation of duty policies, none of the above environments supports obligation constraints or information flow policies as SPL does.

7. Conclusion

We have defined an access control language that supports simultaneously multiple complex policies, and has a higher expressive power than other multi-policy environments. The language uses its hierarchical based, policy-oriented structure to solve conflicts between simultaneously active policies.

The language was designed to be easily enforced by a security monitor. We have shown how index techniques can be applied to the policy structure to implement efficiently most security policies. Special care was taken on the enforcement of history-based constraints. We have shown that by generating specific and special tuned logs for each history-based policy it is possible to implement SPL history-based policies as efficiently as handcoded label-based implementations.

The language goes beyond the permission/prohibition concepts of security and shows how to express and implement the obligation concept. It uses this concept to express a relaxed form of information flow policy, thereby showing that some forms of information flow policies can be expressed in SPL and that they can coexist with other policies.

This work is just a first step towards a security framework, which also includes the specification and enforcement of authentication policies, tools to verify the consistency of both specification and tools to verify the cross consistency of both specifications with other systems in the organization. Namely we have already defined a tool that verifies the cross consistency of an authorization policy described in SPL and a workflow specification [30].

References

[1] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol 1, MITRE Corp., Bedford, MA, Nov. 1973.

[2] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proceeding of the 12th IEEE Computer Security Workshop*. IEEE Computer Society Press, July 1999.

[3] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy*, pages 94–109, Oakland, CA, 1996. IEEE Computer Society Press.

[4] W. R. Bevier and W. D. Young. A constraint language for adage. Technical report, Computational Logic, Inc., Apr. 1997.

[5] Blaze, Feigenbaum, and Strauss. Compliance checking in the policymaker trust management system. In *FC: International Conference on Financial Cryptography*. LNCS, Springer-Verlag, 1998.

[6] M. Blaze, J. Feigenbaum, and J. Ionnidis. The KeyNote trust-management system version. Technical report, Internet RFC 2704, September 1999.

[7] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[8] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, May 1996. IEEE Computer Society,Technical Committee on Security and Privacy, IEEE Computer Society Press.

[9] M. Blaze, J. Pigenbaum, and A. D. Keromytis. Key note: Trust management for public-key infrastructures. *Lecture Notes in Computer Science*, 1550:59–??, 1999.

[10] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *SympSecPr*, Research in Security and Privacy, pages 206–214, Oakland, CA, May 1989. IEEE, IEEECSP.

[11] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 1–14, Berkeley, Jan. 26–29, 1998. Usenix Association.

[12] F. Cuppens and C. Saurel. Specifying a security policy: A case study. In *IEEE Computer Society Computer Security Foundations Workshop (CSFW9)*, pages 123–135, 1996.

[13] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 20, July 1977.

[14] DoD. Dod trusted computer system evaluation criteria. Technical Report 5200.28-STD, DoD, Dec.26 1985.

[15] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *5th ACM Conference on Computer and Communications Security*, pages 38–48, San Francisco, California, Nov. 1998. ACM Press.

[16] W. K. Edwards. Policies and roles in collaborative applications. In *Proceedings of the ACM 1996 Conference on Computer Supported Work*, pages 11–20, New York, Nov. 16–20, 1996. ACM Press.

[17] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory. Internet RFC 2693, Sept. 1999.

[18] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, Feb. 1999.

[19] T. Fraser and L. Badger. Ensuring continuity during dynamic security policy reconfiguration in dte. In *Proceedings of the 1998 IEEE Conference on Security and Privacy (SSP '98)*, pages 15–26. IEEE, May 1998.

[20] L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 153–159, New York, Nov. 6–7 1997. ACM Press.

[21] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[22] J. A. Hoagland, R. Pandley, and K. N. Levitt. Security policy specification using a graphical approach. Technical report, Department of Computer Science, University of California Davis, July 1998.

[23] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *SympSecPr*, Research in Security and Privacy, Oakland, CA, May 1997. IEEECSP.

[24] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):474–485, June 1997.

[25] N. Li, J. Feigenbaum, and B. N. Grosof. A logic-based knowledge representation for authorization with delegation. In *Proceeding of the 12th IEEE Computer Security Workshop*. IEEE Computer Society Press, July 1999.

[26] E. C. Lupu and M. Sloman. Reconciling role-based management and role-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 135–142, New York, Nov. 6–7 1997. ACM Press.

[27] N. H. Minsky and V. Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 131–142, Berkeley, Jan. 26–29, 1998. Usenix Association.

[28] M. J. Nash and K. R. Poland. Some conundrums concerning separation of duty. In *Proceedings of the 1990 IEEE Conference on Security and Privacy (SSP '90)*, pages 201–209. IEEE, May 1990.

[29] H. packard Publication Services. HP praesidium / authorization server white paper. Published in Internet, 1998.

[30] C. Ribeiro and P. Guedes. Verifying workflow processes against organization security policies. In *Proceedings of the 8th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 190–191, Standford, California, June16-18 1999. IEEE Computer Society, IEEE Computer Society.

[31] C. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes. Security policy consistency. Technical Report RT/03/00, INESC, 2000.

[32] R. Sandhu. Separation of duties in computarized information systems. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, Halifax, UK, Sept.18–21 1990.

[33] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9, Nov. 1993.

[34] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control: A multi-dimensional view. In *10th Annual Computer Security Applications Conference*, pages 54–62, 1994.

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

[36] F. B. Schneider. Enforceable security policies. *The ACM Transactions on Information and System Security*, 3(1), February 2000.

[37] Simon and Zurko. Separation of duty in role-based environments. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[38] A. Tate. Representing plans as a set of constraints - the <I-N-OVA> model. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 221–228. AAAI Press, 1996.

[39] T. Y. C. Woo and S. S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 1992 IEEE Computer Society Symposium on Security and Privacy (SSP '92)*, pages 33–51. IEEE, May 1992.

[40] M. E. Zurko, R. Simon, and T. Sanfilipo. A user-centered, modular authorization service built on an rbac foundation. In *Proceeding of the 12th IEEE Computer Security Workshop*. IEEE Computer Society Press, July 1999.

A. Appendix

```
package Global;
import SplInterfaces;

alias object set collection;
alias user set team;

policy Group {
  team UserGroup;
  ?Group: ce.action.name = "read" &
          ce.target.owner IN UserGroup &
          ce.author IN UserGroup :: true;
}

policy AclDomain {
  collection DomainTargets;
  owner: ce.author = ce.target.owner :: true;
```

```
  given:
    FORALL r IN ce.target.owner.userPolicy {r};
  group0: new Group;
  group1: new Group;
  group2: new Group;
  group3: new Group;
  group4: new Group;
  group5: new Group;
  group6: new Group;
  group7: new Group;
  group8: new Group;
  group9: new Group;
  groups: group0 OR group1 OR group2 OR
          group3 OR group4 OR group5 OR
          group6 OR group7 OR group8 OR group9;
  total: {groups OR owner} AND given OR deny;
  ?AclDomain: total@{ .target IN DomainTargets };
}

policy ACE(object target, operation action,
           user author, boolean result) {
  ?ACE: ce.target = target & ce.action = action &
        ce.author = author :: result;
}

policy ChineseClass
{
  collection InterestClass;
  ?ChineseClass:
    NOT EXIST e IN PastEvents {
      ce.target IN InterestClass &
      e.target IN InterestClass &
      e.author = ce.author &
      e.target != ce.target :: true
    };
}

policy Role {
  team Authorized;
  team Active;
  ?Role: ce.action.name = "insert" &
         ce.target = Active ::
       ce.parameter[0] IN Authorized;
}

policy ChineseRBAC {
  collection DomainTargets;
  broker: new Role;
  inspector: new Role;
  china0: new ChineseClass;
  china1: new ChineseClass;
  china2: new ChineseClass;
  china3: new ChineseClass;
  china4: new ChineseClass;
  china5: new ChineseClass;
  china6: new ChineseClass;
  china7: new ChineseClass;
  china8: new ChineseClass;
  china9: new ChineseClass;
  ChineseWall: china0 AND china1 AND china2 AND
               china3 AND china4 AND china5 AND
               china6 AND china7 AND china8 AND china9;
  LocalChina: ChineseWall@{.author IN broker.Active};
  Inpection: ce.author IN inspector.Active ::
             ce.action.name = "read";
  total: {LocalChina AND Inpection} OR deny;
  ?ChineseRBAC: total@{.target IN DomainTargets };
}
policy Continent {
  acl0: new AclDomain;
  acl1: new AclDomain;
  acl2: new AclDomain;
  acl3: new AclDomain;
  acl4: new AclDomain;
  acl5: new AclDomain;
  acl6: new AclDomain;
  acl7: new AclDomain;
  acl8: new AclDomain;
  acl9: new AclDomain;
  acls: acl0 AND acl1 AND acl2 AND
        acl3 AND acl4 AND acl5 AND
        acl6 AND acl7 AND acl8 AND acl9;
  rbac0: new ChineseRBAC;
  rbac1: new ChineseRBAC;
  rbac2: new ChineseRBAC;
  rbac3: new ChineseRBAC;
  rbac4: new ChineseRBAC;
  rbac5: new ChineseRBAC;
  rbac6: new ChineseRBAC;
  rbac7: new ChineseRBAC;
  rbac8: new ChineseRBAC;
  rbac9: new ChineseRBAC;
  rbacs: rbac0 AND rbac1 AND rbac2 AND
         rbac3 AND rbac4 AND rbac5 AND
         rbac6 AND rbac7 AND rbac8 AND rbac5;
  ?Continent: acls AND rbacs;
}

policy Global {
  europe: new Continent;
  america: new Continent;
  asia: new Continent;
  africa: new Continent;
  oceania: new Continent;
  ?Global: europe AND america AND
           asia AND africa AND oceania;
}
```