1 Introduction

During the last decade, where computational power has increased dramatically, and the use of the Internet plays a very important role both in our everyday lives and in the evolution of Computer Science as we know it, two new terms have emerged: grid and cloud computing. The former refers to the combination of computer resources from multiple administrative domains for a common goal, as mentioned in [8]. For the latter, definitions are a bit more fuzzy; for some, it refers to Internet-based computing, whereby shared resources, software and information are provided to computers and other devices on-demand, like a public utility ([9]), whereas to others it’s just anything you consume outside your firewall.

Apart from the obvious positives of grid and cloud computing, with the most important being tremendous improvement on run-times of programs due to the vastly increased computational power of each such system, these notions have brought in surface a big issue for consideration; how secure is data someone retrieves/computes through a cloud or a grid? In order for this question to be fully answered, scientists have been trying over the past few years firstly to determine exactly the notion of Security; when exactly is a program, a network, a computational system secure? And the next question is, of course, about the cost of security; how costly is it, how much overhead in both time and processing power is needed in order to classify something as secure, after certain procedures, and how can we minimize this overhead?

In order for security issues to be handled, both the cloud and grid computing communities have used Access Control Policies in order to explicitly denote when a resource is available to a particular user, with various other restrictions, such as time frames in which the policy can be used. In order for these policies to be implemented, there were several attempts to construct programming languages in which someone can easily describe various policies to apply in a particular system. The most promising efforts were XACML ([2]) and SecPAL ([3]). What is common to these languages, is that the policies are described in a higher-level language, close to plain English, and the compiler generates XML code to be applied in various web services, making its application to grid and cloud systems easier.

However, it was never clear whether these implementations are actually optimal in space and time; all we know is that we can easily describe the desired policies, but we have no knowledge of how they are implemented in the core of each language. Our purpose, is to try to implement these using Datalog ([6]) or some particular set expressions which can be optimized with methods we will see below.

2 A bit of SecPAL

Since the need for data accesses within a Grid has been increasing over the past years, a proposal for a language designed specifically for implementing Access Control Policies was published in 2007 ([3]). The language is called SecPAL and was built on top of Microsoft’s .NET platform. Below we will discuss the main features and semantics of the language, in order to justify the design choices we made while implementing the security policies in respect, by taking an example program and discussing it.

The following policy implements attribute-based access; it gives access to read and write a particular file,
only to users who possess a Stony Brook e-mail address, which must be valid within a given timeframe.

Policy Claim 1:
LA says SBa-CA can say
\[ \%p \text{ possesses } \%a \text{ (from } \%t1 \text{ until } \%t2) \]
where
\[ \%t2 - \%t1 <= "365.00:00:00", \]
\[ \%t1 <= \text{CurrentTime}(\) <= \%t2, \]
\[ \%a \text{ matches rfc822Name:".} @cs.sunysb.edu " \]

Policy Claim 2:
LA says \%p can read, write
digitalContent:" file:///GridFTPRoot/" if
\[ \%p \text{ possesses } \%a \]
where
\[ \%a \text{ matches rfc822Name:".} @cs.sunysb.edu " \]

Security Tokens
SBa-CA says K-User possesses
rfc822Name:"spyros@cs.sunysb.edu"
(from "2010-01-01T00:00:00Z" until 2010-12-31T00:00:00Z")

Authorization Query
LA says K-User can read
digitalContent:" file:///GridFTPRoot/data1 .txt"

A SecPAL program consists of three parts:
1. The Policy Claim(s) part
2. The Security Token(s) part
3. The Authorization Query part

In the first part, we describe exactly how are policy claims on the grid look like. The main entity that handles security access within the Grid, is the Local Authority; we can also have other entities with intermediate-level clearance, which handle security accesses within users. Anything that can be deducted from our program, is only what the Local Authority can say. So, in the first policy claim of our example, the Local Authority says that the intermediate authority SBa-CA can say that some user possesses a certain e-mail address, within certain dates. What is meant here, is that the fact that the user can possess the respective information can be deducted from our program, since the Local Authority can say so, after it gives access to the intermediate authority to say so. The second policy claim is very similar; a user can read a certain file within the grid, if their e-mail address matches a given pattern, and this happens only because the Local Authority says so.

In the second part, we introduce our security tokens; the data we have in our system, from which using the policy claims everything else can be deducted. In this case, the only security token we have, is that our intermediate authority says that a particular user obtains a certain e-mail address for a given period of time.

In the third and last part, we pose the authorization query we need to know the answer of; in our case, whether a particular user has access to read a certain file. Notice that the query is of the form “does the Local Authority say that ...?” , since as we said before, everything that can be deducted from our system, is what the Local Authority can say.

SecPAL has of course many other features. However, in this project we only used the ones introduced while describing some security policies in [13]. The above are almost all features used in this paper, so in our analysis that follows, we will describe the rest few when we encounter them.

3 Our Approach
In our project proposal ([10]), we stated that we would try to implement these policies either in Datalog ([6]), or in particular set expressions, in order to enable certain optimizations designed for these environments. After a lot of thinking, we ended-up implementing everything in XSB ([11]), since tabling came very handy to avoid falling into infinite loops, while evaluating various authorization queries. The most important about this approach, is that we used only pure datalog plus tabling, so the optimization methods mentioned ([14], [15]) can still be applied.

The first thing we had to take care of, is how we would represent the relationships between the various actors in our system, and in the same time leave the overall behavior of the system unaffected; we wanted to be able to provide the input and output in a way as similar as possible to the way they it’s represented in SecPAL.

Recall that a policy claim is of the following form:
LA says SBa-CA can say
\[ \%p \text{ possesses } \%a \text{ (from } \%t1 \text{ until } \%t2) \]
where
\[ \%t2 - \%t1 <= "365.00:00:00", \]
\[ \%t1 <= \text{CurrentTime}(\) <= \%t2, \]
\[ \%a \text{ matches rfc822Name:".} @cs.sunysb.edu " \]
In effect, we have created a \texttt{says} predicate, which has 2 arguments; the first one is a variable for the entity we want to declare that \texttt{says} something, and the second is a placeholder for the actual action an actor does. Apart from that, we also need predicates to declare the actions themselves, like \texttt{possesses}, \texttt{reads}, \texttt{writes}. The \texttt{possesses} predicate has 4 arguments; the first is a variable for the name of the entity that actually possesses something, the second is a variable for the name of the possession held by the previous argument’s entity, and the two last ones stand for the starting and ending times respective, within which the possession is valid. The \texttt{reads} and \texttt{writes} predicates are exactly the same; they have 2 arguments, the first one for the entity that reads (or writes respectively) a file, and the second one is a placeholder for the name of the file itself.

We have also created some helper predicates, as following:

- The \texttt{lt} predicate takes two timestamps as arguments, and returns \texttt{true} if the first is earlier than the second, and \texttt{fail} otherwise
- The \texttt{member} predicate takes as arguments an element and a list, and returns \texttt{true} if the element is found within the list, and \texttt{fail} otherwise
- The \texttt{current.time} predicate defines the current time of the system
- The \texttt{la} predicate defines Local Authorities within our system
- The \texttt{uva} and \texttt{ksts} predicates define entities with intermediate-level clearance

Also, the current time is represented by a list with 3 elements; the first is for the hour, the second for the minutes, and the third for the seconds of current time. For example, the time \texttt{12:11:42} is represented straightforwardly as \texttt{[12,11,42]}. The given e-mail address to validate is also a list of as many elements as dots and the “@” symbol split an RFC-822 e-mail address. For example, the e-mail address \texttt{spyros@cs.sunysb.edu} is represented as \texttt{[spyros,cs,sunysb,edu]}. 

So, for the code snippet given above, the respective XSB code is the following:

\begin{verbatim}
says (L, possesses (P,A,T1,T2)) :-
l a (L) ,
says (U, possesses (P,A,T1,T2)) ,
 uva (U) ,
 member (cs ,A) ,
 member (sunysb ,A) ,
 member (edu ,A) ,
current.time (T) ,
l t (T1,T) ,
l t (T,T2) .
\end{verbatim}

We can understand what this policy claim does, in respect in the one define above in SecPAL, as follows: \texttt{L says} that some user \texttt{P possesses} an e-mail address \texttt{A} within the timeframe \texttt{T1} - \texttt{T2}, if \texttt{L} is on our Local Authority list (\texttt{la(L)}), if there exists a fact in our system that says that some intermediate-level clearance entity (\texttt{uva(U)}) that \texttt{says} that some user \texttt{P possesses} an e-mail address \texttt{A} within the timeframe \texttt{T1} - \texttt{T2}, and if the e-mail address is properly validated. The calls to \texttt{member} and \texttt{lt} just verify that the given e-mail address actually matches e-mail addresses of the Stony Brook domain (\texttt{cs.sunysb.edu}), and that the current time is within the timeframe given, respectively.

\section{The Implementation}

We followed the same pattern for all of the implemented policies. In this paragraph, we will discuss thoroughly the exact implementations of each policy, and give examples of running our designed system in XSB. The only predicate we table, is \texttt{says} because it’s the predicate that distributes access control within the actors/entities in our system.

The initial data on our system, are about defining Local Authorities, intermediate-level entities, and the current date, are common to all the cases below, so we only mention them once, below:

\begin{verbatim}
current.time ([14,34,25]) .
l a (localauth) .
uva (uvauth) .
uva (uvaldap) .
ksts (ks) .
\end{verbatim}

\subsection{Case 1 : Attribute - based Access}

The SecPAL code for the first case, is shown below. In this case, the Local Authority authorizes an intermediate actor to say that some user possesses a certain e-mail address, within a given timeframe. As per our previous description, this is equivalent with the Local Authority itself saying that the user has the respective property, hence generating authorization tokens from the security token given.

\begin{verbatim}
Policy Claim 1:
\end{verbatim}
LA says SBa–CA can say
%p possesses %a (from %t1 until %t2)
where
%t2 − %t1 <= "365.00:00:00",
%t1 <= CurrentTime() <= %t2,
%a matches rfc822Name:”.*@cs.sunysb.edu”

Policy Claim 2:
LA says %p can read, write
digitalContent:” file://GridFTPRoot/”
if
%p possesses %a
where
%a matches rfc822Name:”.*@cs.sunysb.edu”

Security Tokens
SBa–CA says K–User possesses
rfc822Name:” spyros@cs.sunysb.edu”
(from “2010–01–01T00:00:00Z” until
2010–12–31T00:00:00Z”)

Authorization Query
LA says K–User can read
digitalContent:” file://GridFTPRoot/data1 .txt”?

Our XSB code for this first case, is shown below:
Policy Claims
says(L, possesses(P,A,T1,T2)) :-
la(L),
says(U, possesses(P,A,T1,T2)),
ufa(U),
member(cs ,A),
member(sunysb,A),
member(edu,A),
current_time(T),
lT(T1,T),
lT(T,T2).
says(L, reads(P, file)) :-
la(L),
says(L, possesses(P,A……)).
says(L, writes(P, file)) :-
la(L),
says(L, possesses(P,A……)).

Security Tokens
says(uvauth, possesses(k_usr ,[spyros,cs,sunysb,edu], [11,12,13], [17,21,23])).
says(uvauth, possesses(s_usr ,[katerina,liu,edu], [11,12,13], [23,59,59])).
says(uvauth, possesses(p_usr ,[al,cs,sunysb,edu], [9,12,13], [12,59,59])).

Authorization Token
says(localauth, reads(k_usr, file)).
says(localauth, reads(s_usr, file)).
says(localauth, reads(p_usr, file)).

Posing the Authorization Queries on our system, we expect only the first to succeed, and the other two to fail. The reason is that both the e-mail address k_usr possesses is valid and it’s valid when the query is being asked (at current time), whereas in the second case, the e-mail address possessed by s_usr is invalid, and in the third case, although the e-mail address possessed by p_usr is valid, it’s timeframe ends before current time. Truly, our system responds:
| ?- says(localauth, reads(k_usr, file)). yes
| ?- says(localauth, reads(s_usr, file)) .
| ?- says(localauth, reads(p_usr, file)) .

If we change the current time to 23:59:59, we expect none of the users to be able to read the file, since all the e-mail addresses expier before that:
| ?- says(localauth, reads(k_usr, file)). no
| ?- says(localauth, reads(s_usr, file)). no
| ?- says(localauth, reads(p_usr, file)) .

4.2 Case 2 : Role-based Access

In this case, we introduce roles in our system, so that the Local Authority authorizes users to do certain actions, only if they belong to particular roles. The SecPAL code for this second case, is as follows:
Policy Claim 1
LA says SBa–CA can say
%p possesses %a (from %t1 until %t2)
where
%t2 − %t1 <= "365.00:00:00",
%t1 <= CurrentTime() <= %t2,
%a matches rfc822Name:”.*@cs.sunysb.edu”

Policy Claim 2
LA says SBa–LDAP can say
%p possesses %a (from %t1 until %t2) if
%p possesses %b
where
%t2 − %t1 <= "365.00:00:00",
%t1 <= CurrentTime() <= %t2,
%a matches roleName:”ˆ (Admin | Faculty | Student)$”
Policy Claim 3
LA says %p can read, write
digitalContent:” file:// GridFTPRoot”
if
%p possesses %a
where
%a matches rfc822Name: ”.*@cs.sunysb.edu"

Security Token 1
SBa-CA says K-User possesses
rfc822Name: ”spyros@cs.sunysb.edu”
(from ”2010-01-01T00:00:00Z” until ”2010-12-31T00:00:00Z”)

Security Token 2
SBa-LDAP says K-User possesses
roleName: ”Student”
(from ”2010-01-01T00:00:00Z” until ”2010-12-31T00:00:00Z”)

Authorization Query
LA says K-User can read
digitalContent: ” file:// GridFTPRoot/data1.txt”?

Our XSB code for this case, is a straightforward application of what we have mentioned in the previous paragraphs, and is in fact very similar to the previous case (policy claim 1); we just add another say(L, ) predicate, to accomodate the need for verification of the role a user has, before granting access to manipulate a file. This is done in policy claim 2, where the Local Authority can say something (hence that something is deductible from the system) if both the first intermediate-level actor says he can possess the appropriate e-mail address, and the other intermediate-level actor says he has the appropriate role. The code is as follows:

Policy Claim 1
says (L, possesses (P, A, T1, T2)) :-
la(L),
says(uvauth, possesses (P, A, T1, T2),
member(cs, A),
member(sunysb, A),
member(edu, A),
current_time(T),
lt (T1, T),
lt (T, T2),
member (cs, A),
member (sunysb, A),
member (edu, A).

Policy Claim 2
says (L, possesses (P, B, T1, T2)) :-
la(L),
says(uvauth, possesses (P, A, T1, T2),
current_time(T),
lt (T1, T),
lt (T, T2),
member (cs, A),
member (sunysb, A),
member (edu, A).

Policy Claim 3
says(L, reads (P, file_fac)) :-
la(L),
says(L, possesses (P, faculty, ),
says(uvauth, possesses (P, faculty, ),
member(cs, A),
member(sunysb, A),
member(edu, A).

Security Token 1
says(uvaldap, possesses (k_usr, [spyros, cs, sunysb, edu], [11, 12, 13, [17, 21, 23]])).
says(uvaldap, possesses (s_usr, student, [spyros, cs, sunysb, edu], [11, 12, 13, [23, 59, 59]])).
says(uvaldap, possesses (p_usr, student, [9, 12, 13, [17, 21, 23]])).

Security Token 2
says(uvaldap, possesses (k_usr, [spyros, cs, sunysb, edu], [11, 12, 13, [17, 21, 23]])).
says(uvaldap, possesses (s_usr, student, [spyros, cs, sunysb, edu], [11, 12, 13, [23, 59, 59]])).
says(uvaldap, possesses (p_usr, student, [9, 12, 13, [12, 59, 59]])).

Authorization Query
says(localauth, reads (k_usr, file_fac)).
says(localauth, reads (s_usr, file_fac)).
says(localauth, reads (p_usr, file_fac)).

After posing the Authorization Queries on our system, the answer is the expected; since user k_usr possesses both the e-mail address of the correct form, valid at the time of the query, and the appropriate role (faculty), he is granted access to the file:

| yes |
| no |
| no |

If we change the appropriate Security Token to denote that k_usr possesses a non-faculty role (for example, a student role), the system will act as expected, denying access:
4.3 Case 3: Role-Deny Access

This case is the exact contrapositive of the previous one: a user can access some resource, only if he does not belong in a particular role; any other is acceptable. The original SecPAL code follows:

**Policy Claim 1**

LA says SBa–CA can say
\[
\text{A} \text{ possesses } \text{A} \text{ (from } \text{T} \text{ until } \text{T}')
\]
where
\[
\text{T}' - \text{T} \leq 10000000000, \\
\text{T} \leq \text{CurrentTime()} \leq \text{T}', \\
\text{A} \text{ matches rfc822Name: "@cs.sunysb.edu "}
\]

**Policy Claim 2**

LA says SBa–LDAP can say
\[
\text{A} \text{ possesses } \text{A} \text{ (from } \text{T} \text{ until } \text{T}')
\]
where
\[
\text{T}' - \text{T} \leq 10000000000, \\
\text{T} \leq \text{CurrentTime()} \leq \text{T}', \\
\text{A} \text{ matches roleName: "Admin|Faculty|Student"}
\]

**Policy Claim 3**

LA says %p can read, write digitalContent: "file://GridFTPRoot/data1 .txt" AND NOT LA says K–User possesses roleName: "Student"

Our XSB code is again very similar to the previous ones. The only difference, is that now in our Authorization Query he actually have two conditions that need validation, and one of them is in negative form (NOT). Thus, we will use the Negation as Failure of Prolog, in order to denote that we want our query to be true, only when the first part is true and the second false (when the first part succeeds, and the second fails).

**Security Claim 1**

\[
\text{says (L, possesses (P, A, T1, T2)) :- la (L),} \\
\text{says (uvauth, possesses (P, A, T1, T2)),} \\
\text{member (cs, A),} \\
\text{member (sunysb, A),} \\
\text{member (edu, A),} \\
\text{current_time (T),} \\
\text{lt (T1, T),} \\
\text{lt (T, T2).}
\]

**Security Claim 2**

\[
\text{says (L, possesses (P, B, T1, T2)) :- la (L),} \\
\text{says (uvldap, possesses (P, B, T1, T2)),} \\
\text{current_time (T),} \\
\text{lt (T1, T),} \\
\text{lt (T, T2).}
\]

**Security Claim 3**

\[
\text{says (L, reads (P, file_fac )) :- la (L),} \\
\text{says (L, possesses (P, faculty, ...)).}
\]

**Security Token 1**

\[
\text{says (uvldap, possesses (k_usr, faculty }, [11, 12, 13], [17, 21, 23])}. \\
\text{says (uvldap, possesses (s_usr, student }, [11, 12, 13], [23, 59, 59])}. \\
\text{says (uvldap, possesses (p_usr, student }, [9, 12, 13], [17, 21, 23])).
\]

**Security Token 2**

\[
\text{says (uvaht, possesses (k_usr, spyros, cs, sunysb, edu), [11, 12, 13], [17, 21, 23])}. \\
\text{says (uvaldap, possesses (s_usr, katerina, liu, edu), [11, 12, 13], [23, 59, 59])}. \\
\text{says (uvaldap, possesses (p_usr, al, cs, sunysb, edu), [9, 12, 13], [12, 59, 59])).}
\]
Authorization Query

\[ q : - \text{says}(\text{localauth}, \text{reads}(k_{\text{usr}}, \text{file}_{\text{fac}})), \]
\[ + \text{says}(\text{localauth}, \text{possesses}(k_{\text{usr}}, \text{faculty}, \ldots)) \]

The answer to query \( q \) is, of course, no, since the Local Authority may authorize the user \( k_{\text{usr}} \) to read the respected file, but since accesses are allowed only from users which DO NOT possess the role of faculty, access will be denied:

\[ | \ \neg q. \]
\[ \text{no} \]

If we change the Authorization Query to look like this, so that everyone else except student can be granted access, we get a different result:

New Authorization Query

\[ q : - \text{says}(\text{localauth}, \text{reads}(k_{\text{usr}}, \text{file}_{\text{fac}})), \]
\[ + \text{says}(\text{localauth}, \text{possesses}(k_{\text{usr}}, \text{student}, \ldots)) \]

\[ | \ \neg q. \]
\[ \text{yes} \]

4.4 Case 4: Impersonation-based Access

In this case, some users can be granted access to act as others, they thus inherit all the access rights the user they can act as have. In SecPAL, this is implemented in the following way:

Policy Claim 1

LA says SBa-CA can say
\[ \%p \text{ possesses } \%a \text{ (from } \%t1 \text{ until } \%t2) \]
where
\[ \%t2 - \%t1 \text{ <= } "365.00:00:00", \]
\[ \%t1 \text{ <= CurrentTime()} \text{ <= } \%t2, \]
\[ \%a \text{ matches rfc822Name:}\text{".}@cs.sunysb.edu" \]

Policy Claim 2

LA says \( \%p \) can read, write
digitalContent:" file://GridFTPRoot/"
if
\[ \%p \text{ possesses } \%a \]
where
\[ \%a \text{ matches rfc822Name:}\text{".}@cs.sunysb.edu" \]

Policy Claim 3

LA says SBa-CA can say \( \%x \) can act as \( \%y \)
( from \( \%t1 \) to \( \$t2 \) ) where
\[ \%t2 - \$t2 \text{ <= } "365.00:00:00", \]
\[ \%t1 \text{ <= CurrentTime()} \text{ <= } \%t2 \]

Security Token 1

SBa-CA says K-User possesses
rfc822Name:"spyros@cs.sunysb.edu"
(from "2010-01-01T00:00:00Z" until 2010-12-31T00:00:00Z")

Security Token 2

SBa-CA says K-User2 can act as K-User

Authorization Query

LA says K-User2 can read
digitalContent:" file://GridFTPRoot/data1.txt"?

In order to implement this policy, we needed to introduce a new predicate acts, to denote that some user can act as another one, and to facilitate the security rights transfer from the later to the former. So, the Local Authority says that some user can act as another, (so this fact is deductible from our system), if one of the intermediate-level actors can say that this can happen, and if they possess the appropriate e-mails. The says predicate is augmented with a definition to facilitate the transfer of security rights between users, by just calling the acts and says(L,possesses(\ldots)) predicates, with the right arguments. The complete XSB code is shown below:

Policy Claim 1

\[ \text{says}(L, \text{possesses}(P,A,T1,T2)) : - \]
\[ \text{la}(L), \text{uva}(U), \text{member}(cs,A), \text{member}(sunysb,A), \text{member}(edu,A), \text{current_time}(T), \]
\[ \text{lt}(T1,T), \text{lt}(T,T2) \]

Policy Claim 2

\[ \text{says}(L, \text{acts}(X,Y)) : - \]
\[ \text{la}(L), \text{uva}(U), \text{says}(U, \text{possesses}(P,A,T1,T2)), \]
\[ \text{says}(L, \text{possesses}(Y,A,T1,T2)), \text{current_time}(T), \]
\[ \text{lt}(T1,T), \text{lt}(T,T2) \]

Policy Claim 3

\[ \text{says}(L, \text{reads}(P, \text{file})) : - \]
\[ \text{la}(L), \]
says (L, possesses (P, A, \ldots)).
says (L, writes (P, file)) :-
   la (L),
says (L, possesses (P, A, \ldots)).

Security Token 1
says (uvauth, possesses (k_usr, [spyros, cs, sunysb, edu], [11,12,13], [17,21,23])).
says (uvauth, possesses (s_usr, [katerina, liu, edu], [11,12,13], [23,59,59])).
says (uvauth, possesses (p_usr, [al, cs, sunysb, edu], [9,12,13], [12,59,59])).

Security Token 2
says (uvauth, acts (k_user2, k_usr)).

Authorization Query
| ?- says (localauth, reads (k_user2, file)).
   yes |
| ?- says (localauth, reads (s_usr, file)).
   no |
| ?- says (localauth, reads (p_usr, file)).
   no |

Note that if we change our security token 2, so that 
\texttt{k\_user2} acts as \texttt{s\_usr}, he will not have access to the 
resource, since \texttt{s\_usr} doesn’t have either:

Security Token 2
says (uvauth, acts (k_user2, s_usr)).

Authorization Query
| ?- says (localauth, reads (k_user2, file)).
   no |
| ?- says (localauth, reads (s_usr, file)).
   no |
| ?- says (localauth, reads (p_usr, file)).
   no |

4.5 Case 5 : Delegation-based Access

In this case, a single user can denote that he can 
transfer (some or all of) his rights to another user. 
However, it is not stated anywhere that the second 
user must obtain a valid e-mail address, so we don’t 
make this assumption either. The code in SecPAL is 
as follows:

Policy Claim 1
LA says SBa-CA can say
\%p possesses \%a (from \%t1 until \%t2)
where
\%t2 - \%t1 <= ”365.00:00:00”,”
\%t1 <= CurrentTime() <= \%t2,
\%a matches rfc822Name:”.*@cs.sunysb.edu”

Policy Claim 2
LA says \%p can read, write
digitalContent:”file:///GridFTPRoot/”
if
\%p possesses \%a
where
\%a matches rfc822Name:”.*@cs.sunysb.edu”

Policy Claim 3
LA says \%p can say \%x can \%v
digitalContent:”file:///GridFTPRoot/”
if
\%p can \%v digitalContent:”file:///GridFTPRoot/”

Security Token 1
SBa-CA says K-User possesses
rfc822Name:”spyros@cs.sunysb.edu”
(from ”2010-01-01T00:00:00Z” until 
2010-12-31T00:00:00Z”)

Security Token 2
K-User says K-User2 can read
digitalContent:”file:///GridFTPRoot/”

Authorization Query
LA says K-User2 can read
digitalContent:”file:///GridFTPRoot/data1.txt”?

Note that in the policy claim 3, the variable \texttt{x} is not 
used at all; this means that we only care about the 
user that transfers his rights, and the actual act that 
the user to whom the rights are transferred can do. 
Thus, in our implementation, we use the existing def-
definition of the \texttt{says} predicate, and augment it to ac-
modate the changes we need; we add two new clauses 
for \texttt{reads} and \texttt{writes}, so that the Local Authority 
can also say something, if another user that \texttt{has ac-
access to do the same action} says it. The \texttt{XSB} code 
is as follows:

Policy Claim 1
says (L, possesses (P, A, T1, T2)) :-
   la (L),
says (U, possesses (P, A, T1, T2)),
   uva (U),
   member (cs, A),
   member (sunysb, A),
   member (edu, A),
   current_time (T),
   lt (T, T1),
   lt (T, T2).

Policy Claim 2
says (L, reads (P, file)) :-
   la (L),
says (L, possesses (P, A, \ldots)).
\( \text{says}(L, \text{writes}(P, \text{file})) : \)
\( \text{la}(L), \)
\( \text{says}(L, \text{possesses}(P, A, .., .)) . \)

**Policy Claim 3**
\( \text{says}(L, \text{reads}(X,Y)) : \)
\( \text{la}(L), \)
\( \text{says}(V, \text{reads}(X,Y)), \)
\( \text{says}(L, \text{reads}(V,Y)). \)

\( \text{says}(L, \text{writes}(X,Y)) : \)
\( \text{la}(L), \)
\( \text{says}(V, \text{writes}(X,Y)), \)
\( \text{says}(L, \text{reads}(V,Y)). \)

**Security Token 1**
\( \text{says}(uvauth, \text{possesses}(kusr, [spyros, cs, sunysb, edu], [11,12,13], [17,21,23])). \)
\( \text{says}(uvauth, \text{possesses}(susr, [katerina, liu, edu], [11,12,13], [23,59,59])). \)
\( \text{says}(uvauth, \text{possesses}(pusr, [al, cs, sunysb, edu], [9,12,13], [12,59,59])). \)

**Security Token 2**
\( \text{says}(kusr, \text{reads}(kusr3, \text{file})). \)
\( \text{says}(susr, \text{reads}(kusr4, \text{file})). \)

**Authorization Query**
\( \text{says}(localauth, \text{reads}(kusr2, \text{file})). \)
\( \text{says}(localauth, \text{reads}(kusr3, \text{file})). \)

The behavior of our system is also in this case, the one that should. In the first of the two queries, the system answers yes, since \( kusr \) that transfers his rights to \( kusr2 \) indeed has access to read \text{file}, whereas, \( susr \) that transfers his rights to \( kusr3 \) does not have access on the file.

\[ \text{Authorization Query} \]
\( | ?- \text{says}(localauth, \text{reads}(kusr2, \text{file})). \)
\( \text{yes} \)
\( | ?- \text{says}(localauth, \text{reads}(kusr3, \text{file})). \)
\( \text{no} \)

4.6 Case 6: Capability-based Access

In this last case of security policies presented, we use the notion of **Capabilities**. A capability is an unforgeable token, which when is presented to the Local authority decision about permission granting or denying is based only on the possession of the token itself, and not on the actual identity of the holder. So, in our case, the Local Authority doesn’t care who the bearer is, as long as they actually are in possession of it. The code in **SecPAL** follows:

**Policy Claim**
\( \text{LA says K-STS can say \%p can read, write digitalContent: file:///GridFTPRoot/} \)

**Authorization Query**
\( \text{LA says K-User can read digitalContent: file:///GridFTPRoot/} \)

The implementation for this last one is pretty straightforward; we define a simple version of the \( \text{says} \) predicate, which gives access to users to manipulate files, only if another intermediate-level actor (K-STS) says so. The **XSB** code follows:

**Policy Claim**
\( \text{says}(L,D) : \)
\( \text{la}(L), \)
\( \text{says}(K,D), \)
\( \text{ksts}(K). \)

**Authorization Query**
\( | ?- \text{says}(localauth, \text{reads}(kusr, \text{file})). \)
\( \text{yes} \)

5 Optimizations

In the previous paragraphs, we saw how we can implement in **XSB** security policies for Grid computing, written in **SecPAL**. In this paragraph, we will explore the possibility of optimizations on the derived **XSB** code.

The particular reason we chose **XSB** for our implementation, was mainly the method designed by Liu ([15]), which not only optimizes Datalog code, but gives tight running time and space asymptotic boundaries. All of the rules from above can be re-written in the appropriate form with only two calls in the rule’s body, so that the incrementalization techniques described can be applied successfully. We can have, thus, after a particularly thorough analysis, tight boundaries on our program’s performance, and hence can compare it with the actual **SecPAL** implementation.

The key feature of the incremental algorithm proposed in [15], is effectively the sharing of various variables, constants, and even whole predicate calls within rules. By identifying and grouping together shared data, the cost of actually changing the result of a computation when new rules are added or deleted consists only of calculating the new information, rather than recalculating everything from
scratch. The most important use of this analysis, as said before, is that after the rules are written in incremental form, we can automatically conclude on the time and space complexity of our code; thus being able to decide whether this implementation in XSB is better or worse than the original one in SecPAL. However, as we will further discuss later, efficiency is not the only key part of Access Control policies for grid computing.

The method in discuss also supports some kind of negation (as long as it’s stratified), and that’s exactly the kind we used in our implementation for policy number 3. Some extentions have been made for Datalog programs with data structures other than just lists, so we can use the method in its full extent, in order to satisfy the needs of any security policy needed to implement.

6 Discussion

In the previous paragraph, we explored how optimizations can be applied to our derived code, in order not only to actually optimize the code itself, but to also have fixed asymptotic boundaries about running time and space needed.

Despite the fact that efficiency is in most of the cases the key part of discussion for all kinds of applications, most of the times there are a lot of other factors to consider, before declaring a solution as successful. In this particular case, of implementing security policies for grid (and cloud) computing, there are more to explore than efficiency; the most important being the actual deployment of the implemented code to the grid.

SecPAL was introduced in order to solve the problem of straightforwardly describing security access policies for grid computing, so that even someone with little knowledge of computer programming can code a sufficient amount of them. To be able to to this automatically, Microsoft researchers provided a logic-programming-style language, with almost minimal syntax to make coding easier to the user, and left all the other problems of applying these policies on the grid, setting-up the inference engine, e.t.c. to the system administrators. In [13], the SecPAL system they use consists of 7 components; two of them are the SecPAL engine and handler, one of them is the GridFTP.NET Service which connects all the components together, and the rest three are the policy web services, repository and intermediate-access level actors. In some of the policies described (specifically, in cases 2, 3 and 5), some of the data introduced had to explicitly be written in certain components, other than the SecPAL engine/handler themselves (for example, on the Token Web Service).

Of course, we cannot have any guaranteed time/space complexities about programs in SecPAL, for many reasons, the most important being that the source code for it is closed, and not everyone can have access to it, to analyze it properly. The closest we can get, is actually running experiments, and for that, expensive machinery which are set-up to run as grids are needed. Now, the tradeoffs between our implementation and the one in [13] seem more obvious: We wrote simple and clean Datalog code, for which we can have automatically asymptotic boundaries on space and time complexity of any program, derived directly from the method itself, whereas SecPAL was introduced to automatically facilitate the deployment of the policies on the grid. We haven’t talked about how these rules will affect various machines on the grid, where each piece of data will be stored, where will the Local Authority be installed, and how security accesses will be distributed among actors through the grid. These are all key elements of any grid setup, and appropriate care should be taken, so that each one must be facilitated properly. This is the exact reason why SecPAL was proposed and implemented; to make the programmers and administrators care more on the policies themselves, than on how they will deploy the policy deduction system on their grid.

Our opinion, is that every language is written for its own purpose, and since SecPAL was introduced for a specific reason, it’s somewhat purposeless to talk about using other languages and methods, which may give us automatic and tight space and time asymptotic complexities, but will make it extremely difficult to deploy the whole system on the grid. Nevertheless, it’s always important to have multiple methods of solving a problem, and after all are implemented and compared to each other, useful conclusions can be drawn.

7 Conclusion and Further Work

In this project, we have implemented 6 different access control security policies for grid computing in Datalog, using the XSB Logic Programming system. After applying the method developed by Liu ([15]), we get time and space complexities for the incremental and optimized program, which is very useful for analysis and comparisson purposes. Although
the method is able to be applied in a vast domain of problems (and security policies is one of them), for the particular case of grid (and cloud) computing there are other factors that affect the choice of implementation technique and language, with the most important being the need to easily deploy the designed system on the grid.

In order to compare the two implementations, the transformation of our derived code is needed by a fully-automated system, in respect to the above method. After such a system is implemented, and the code we presented in this project is incrementalized and optimized, measurements can be taken about the time and space the two approaches consume, while trying to distribute access control between actors within a grid system. This should not be an easy task, since the two methods vary extensively on the deployment techniques; while SecPAL is build on top of Microsoft’s .NET platform, and various other components have been built to facilitate the deployment on the grid, our datalog programs have no notion of distributivity of resources between different parts of a grid, and thus new systems must be designed to accommodate the operation of a grid.

8 Acknowledgments

During this project, we have talked to a number of people, both regarding security policies for grid and cloud computing, but for the actual implementation using datalog itself. It was surprising that experts in the field, like Professor Radu Sion ([18]) knew nothing about security policies for grid (and cloud) computing; this means that maybe the idea for decentralized security control has been abandoned during the last two years.

Google was also our friend in the quest for more security policies to implement, but unfortunately none of the following keywords gave useful results (even on scholar.google.com): security policies for grid computing, security policies for cloud computing, access control grid computing, access control cloud computing, SecPAL security policies

Finally, while implementing the policies described, we spent some time talking to Tuncay Sekle, whose advice were more than helpful for our design.

References


[14] **Liu, Y., and Brook, S.** Iterate, incrementa-
ize, and implement: A systematic approach to.

[15] **Liu, Y., and Stoller, S.** From datalog rules
to efficient programs with time and space guar-
antees. *ACM Transactions on Programming
Languages and Systems (TOPLAS)* 31, 6 (2009),
21.

[16] **Paige, R., and Koenig, S.** Finite differenc-
ing of computable expressions. *ACM Transac-
tions on Programming Languages and Systems
(TOPLAS) 4*, 3 (1982), 454.

[17] **Samarati, P., and de Vimercati, S.** Ac-
cess control: Policies, models, and mechanisms.
*Foundations of Security Analysis and Design*
(2001), 137–196.


[20] **Thompson, M., Johnston, W., Mudumbai,
S., Hoo, G., Jackson, K., and Essiari, A.**
Certificate-based access control for widely dis-
tributed resources.