Distributed authorization using delegation with acyclic paths[♦]

Antonio Lain, Miranda Mowbray
Enterprise Systems and Software Laboratory
HP Laboratories Bristol
HPL-2006-30(R.1)
September 10, 2007*

trust management,
SmartFrog

We present a new trust management scheme for distributed authorization which can be easily implemented using X.509-based certificate chains, but does not require globally unique role names. A principal proves that he has authorization for a particular action by demonstrating the existence of an acyclic chain of bindings from a specified principal to himself, where the sequence of labels in the chain matches a template. This template is in an easily-computed subset of regular path expressions. Our restrictions to acyclic paths and to a subset of path expressions enable us to permit controlled delegation, relax the requirement of global agreement on role names, and provide an intuitive abstraction. We show that some useful security properties can be determined in polynomial time. Our scheme has been used in practice to secure a management framework for distributed components: we give an overview of the implementation.

* Internal Accession Date Only
[♦]Proceedings 19th IEEE Computer Security Foundations Workshop, 5-7 July 2006, Venice, Italy

Distributed Authorization Using Delegation with Acyclic Paths[*]

Antonio Lain and Miranda Mowbray
HP Laboratories Bristol, Filton Rd, Stoke Gifford, Bristol BS34 8QZ, UK
{antonio.lain,miranda.mowbray}@hp.com

Abstract

We present a new trust management scheme for distributed authorization which can be easily implemented using X.509-based certificate chains, but does not require globally unique role names. A principal proves that he has authorization for a particular action by demonstrating the existence of an acyclic chain of bindings from a specified principal to himself, where the sequence of labels in the chain matches a template. This template is in an easily-computed subset of regular path expressions. Our restrictions to acyclic paths and to a subset of path expressions enable us to permit controlled delegation, relax the requirement of global agreement on role names, and provide an intuitive abstraction. We show that some useful security properties can be determined in polynomial time. Our scheme has been used in practice to secure a management framework for distributed components: we give an overview of the implementation.

1. Introduction

Distributed authorization schemes allow enforcement of consistent security policies at end-points, without assuming that the end-points always have connectivity to a central server. These access control decisions are not necessarily based on the identity of the requester, but on properties about him that can be derived from his credentials, thus allowing anonymous interactions and role-based models. Built-in delegation mechanisms are also critical for minimizing the amount of information that end-points need in order to take policy decisions. Moreover, since a central server is not necessary, federated systems can be supported in which client credentials can be obtained from sources that do not fully trust each other. Real life applications, such as distributed firewall management [11], benefit greatly by using a distributed authorization model, in terms of scalability and resistance to denial-of service attacks. There are currently many options available for implementing distributed authorization mechanisms, some of them reasonably mature, as we will see in Section 2.

Given all these advantages, why are distributed authorization schemes not more widely used today? Most schemes either rely on authentication based on a hierarchical naming scheme, e.g., X.509 Distinguished Names (DNs) [3], or they rely on a powerful but complex way of linking local name spaces, e.g., SDSI [19]. Clearly, the first approach is limited by the same problems that limit the adoption of X.509-based PKI solutions beyond a single organization [6]. Our personal experience with the second approach is that we found a lot of resistance to using any scheme not based on standard X.509 certificates, and also, as soon as a few local name spaces were linked, it was difficult to reason about the overall system.

Our new scheme for distributed authorization came from a compromise between the two approaches. We designed it to be easily implementable using existing X.509-based certificates, but retaining some of the flexibility of local name spaces, in particular not to require globally unique names. This led to a design in which principals are associated with nodes of a graph, local names (for instance "gold_member") are labels of directed edges between the naming principal and the named principal, and each property that one principal proves to another corresponds to the existence of an acyclic directed path from a specified anchor node to the principal proving the property, whose sequence of labels is one of a specified set of sequences. We will see in Section 5 that the existence of such a path can be trivially mapped to an X.509 certificate chain that, since we guarantee it has no cycles, can be validated with standard X.509 certificate chain tools [5]. Since the property is associated with the sequence of labels in a path from a particular an-

[*]**Copyright (c) 2006 IEEE. Reprinted from 19th IEEE Computer Security Foundations Workshop (Venice, Italy, 5-7 July 2006), pp. 257-269, ISBN 0-7695-2615-2. This material is posted here with the permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of HP Labs Bristol's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.**

chor node, rather than with a single label, we do not need to have a global agreement on the meaning of individual labels. The matching of label sequences also helps to control delegation, as we will see in Section 3.

It may be cumbersome to list all the sequences of labels for which the end-node of the path starting at a particular anchor node would be allowed a particular type of access to a resource. We therefore specify Access Control Lists (ACLs) [13] by using templates which provide a compact representation of a set of valid sequences. The set of templates that we allow is an easily-computable subset of regular path expressions.

Experience of using our scheme for securing a distributed component management framework (SmartFrog [10], see Section 5) led us to look for a higher-level abstraction for distributed access control which would allow us to reason about sets of principals with similar access control behaviour, rather than having to consider each principal separately. This abstraction, the *domain*, is described in Section 4.

The rest of this paper is organized as follows: Section 2 gives a brief overview of existing distributed authorization schemes. Section 3 describes our own in more detail. Section 4 presents it in a more formal way and derives tractability properties. Section 5 is an overview of our first implementation in SmartFrog. Section 6 summarizes our conclusions and describes future work. Finally, we give a usage example in the Appendix.

2. Related Work

Traditional access control typically relies on authentication, i.e., ensuring the identity of the requester, and then mapping that identity to a set of access rights for the protected resource. Following that model, authorization schemes based on X.509 [3] try to map the requester to a globally unique Distinguished Name (DN) derived from a hierarchical Certificate Authority (CA) structure. Some refinements that are currently being proposed include attribute certificates that can associate roles to identities [8], and proxy certificates for delegation that do not require a fresh DN [9]. Unfortunately, it is very difficult to agree on globally-unique and human-understandable names that could guide authorization decisions. This is even more difficult across federated organizations which are not mutually trusting [6].

Trust management schemes (PolicyMaker [2], KeyNote [1], and SPKI/SDSI [7, 19]) solve the global naming problem by dropping the requirement that the globally-unique identifier is easily understandable to humans, and using instead a public key. This allows direct authorization based on keys and security credentials that associate local names or attributes to them. Typically, they

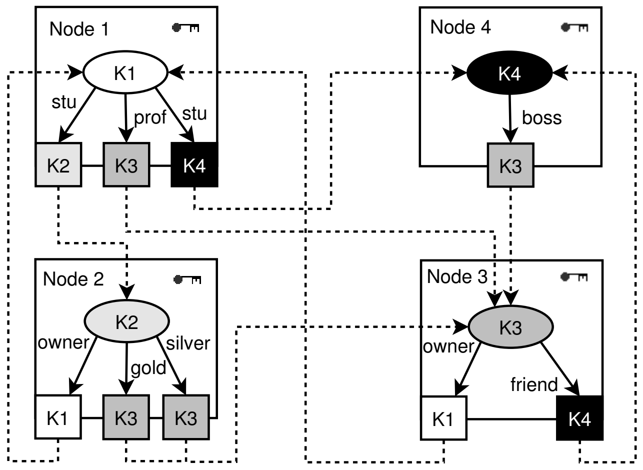

Figure 1. An example with local bindings.

also support delegation by expressing the delegation target explicitly, by its public key, or implicitly, by declaring some of its required attributes. A generic policy engine can then combine these assertions about the requesting principal, together with a local policy, to enforce access control on a particular resource. Using SDSI local names as roles was more explicitly introduced in a trust management system in [14]. New trust management schemes [17] add an XML-like flavour to the previous ideas but, unfortunately, allow non-monotonic policies in some cases.

Our approach can be described as a minimalist role-based trust management framework, in which all the roles correspond to the existence of an acyclic path from a given anchor principal to the requester with an specified sequence of labels. Since the role is defined by the sequence of labels and the existence of such a path, not just on the existence of a single label, the individual labels do not have to have a unique global interpretation. The requirement that the path be acyclic allows us to reuse existing security mechanisms conforming with [5] and (as we will see) makes delegation more intuitive. Also, we do not allow attribute-based delegation in the traditional sense [19], and this enables the policy engine to be very simple, and makes it much easier for clients to figure out which credentials are required [15]. However, if we define a role by simple enumeration of all the relevant sequences of labels, this could lead to unmanageably complex ACLs. Therefore, we introduce a notation, based on an easily-computable subset of regular path expressions [16], that allows us a compact expression of ACLs based on path label sequences.

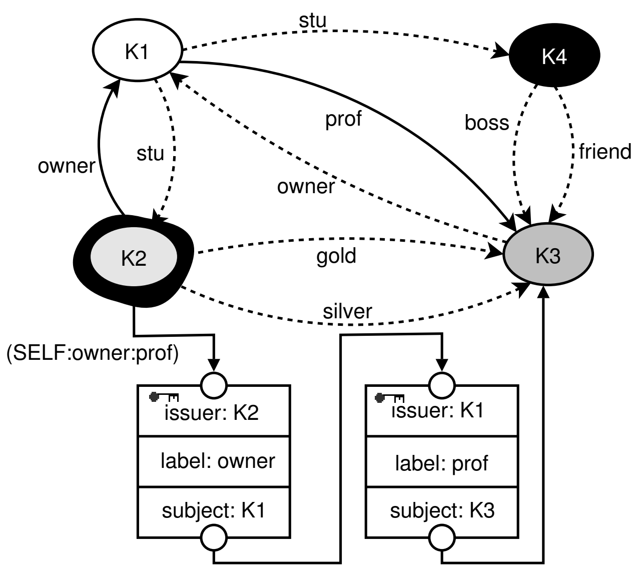

Figure 2. Graph representation of Figure 1.

3. Overview

In this section we will give an indication of how our distributed authorization framework works, delaying to Section 4 a more formal presentation. Figure 1 shows a typical set-up with four principals $K1, K2, K3$ and $K4$, represented by their public keys. Each principal has a set of local bindings that associate labels to other principals. These labels may describe properties of the principals. Labels are not necessary locally unique, for instance $K1$ binds both $K2$ and $K4$ with label stu. A principal can have more than one local binding, for instance $K2$ binds $K3$ both with label $silver$ and with label $gold$. Therefore, to identify a principal uniquely we should rely on its public key, not on labels.

Figure 2 shows how we implement distributed authorization in this framework. The graph shown is just a simple abstraction of Figure 1 where principals become nodes in the graphs and directed labelled edges represent local bindings between the principals. In this context, a node makes a local binding visible to the world by issuing a certificate that contains his public key, the target's public key, and the label of that binding. Note that similarly we can make a path in the graph visible by forming a certificate chain with one certificate per edge in the traversed sequence. For example, since there is a directed path from $K2$ to $K3$ traversing edges $(K2, owner, K1)$ and $(K1, prof, K3)$, $K3$ could obtain the certificates associated with these edges to form a chain, and by presenting this chain and demonstrating knowledge of the private key matching $K3$, could prove that there is a path with labels $owner : prof$ from $K2$ to $K3$. The path can be seen to be acyclic since the keys $K1, K2, K3$ in the certificate chain are all different. Now $K2$ could allow access to a local resource to requesting principals who can prove the existence of an acyclic path with labels $owner : prof$ from $K2$ to the requester.

Instead of directly listing valid label sequences in an ACL [13] associated with a resource, we use a more compact representation of sets of paths whose existence will permit the end node to access the resource. This representation, which we call a Simple Path Constraint (SPC) uses a subset of regular path expressions [16] that is easy to compute. An SPC starts with the anchor for the path, which is either $SELF$ if the path starts at the principal that is enforcing access control for the resource, or is the public key of a principal from whom the path will begin. After the anchor, an SPC has a sequence of patterns, where each pattern is matched against single edge labels. (We could, for instance, use Perl patterns over the set of labels. If a is a label we write a also for the pattern that matches a and no other labels.) The SPC optionally terminates with $\ldots$, which matches an arbitrary sequence of labels. In addition there is a special SPC $ANYBODY$ which allows any principal to access a resource, and an operator $\vee$ for combining SPCs, which allows the condition to be expressed that access is allowed to those principals who can prove the existence of a path corresponding to any one of a finite number of SPCs.

An SPC stands for the set of paths which start at the specified anchor node and whose sequence of labels matches an initial subsequence of the sequence of patterns given in the SPC. For example, in Figure 3, $K5$ is the dean of a university college with a secretary $K6$, and $K7$ is a professor in that school with university students $K8$ and $K9$. We also assume that a high-school student $K10$ is doing a part-time job helping the secretary with paperwork. Using $prof, stu$ as labels, the SPC $(SELF : prof : stu)$ is associated with university student status and can be proved by $K8$ and $K9$ to $K5$ since there are acyclic paths from $K5$ to $K8$ and to $K9$ whose sequence of labels is $prof : stu$. $K7$ can also prove the SPC $(SELF : prof : stu)$ to $K5$ by proving the subsequence $(SELF : prof)$. The dean $K5$ can prove any SPC to himself. Similarly, the SPC $(SELF : admin : stu)$ is associated with high-school student worker status, and the principals $K10$, $K6$ and $K5$ can prove it to $K5$.

We allow the matching of initial subsequences, rather than requiring a match to the full sequence, because of an attack called the "sock-puppet attack". To understand this attack, look again at Figure 3. Suppose the dean required a full match to the sequence $(SELF : prof : stu)$ to identify students. Then the professor $K7$ could pretend to be a student, by creating a fictitious student (the "sock-puppet") with public and private keys under her control, and creating a binding from herself to the sock-puppet with label stu. The dean could not differentiate between real students and

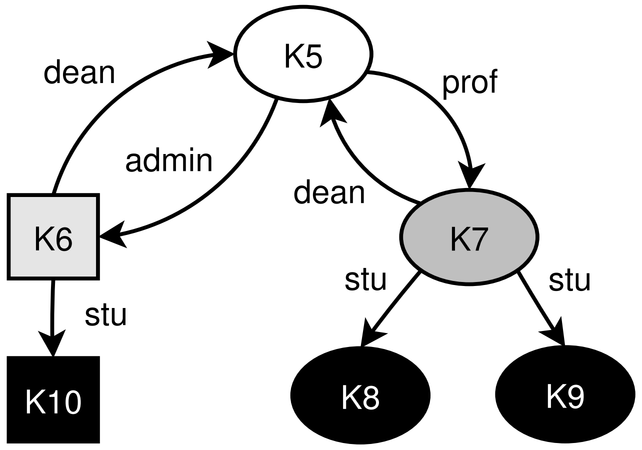

Figure 3. University example with SPCs.

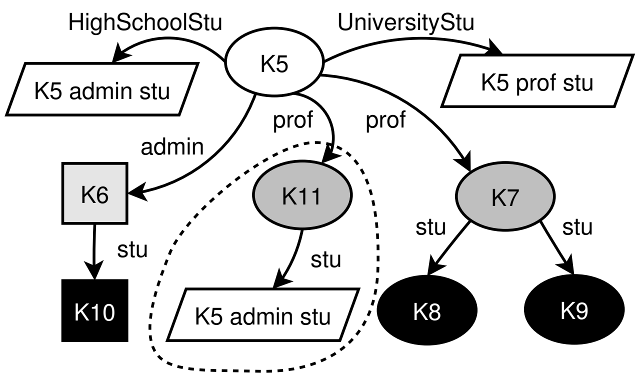

Figure 4. University example with SDSI.

the sock-puppet. We make this fact explicit in the model by allowing the matching of initial subsequences, so that anyone who can prove $(SELF : prof)$ to $K5$ can also prove $(SELF : prof : stu)$ to $K5$. We also assume that any principal can prove any SPC to himself.

In our framework only acyclic paths can be used to prove any role. This requirement is needed if we want smooth integration with X.509 certificate chains (see [5]), since they do not handle cycles well. Also, if paths containing cycles were allowed, this could sometimes make the access control rules less intuitive. In Figure 3, the dean probably wants to specify different access controls for administrative tasks than for dealing with the academic members of the school, i.e., his professor and her students. If he identifies academic members with $(SELF : prof : ...)$, that is, professors and anyone to whom professors have a binding, and cycles are allowed in paths, then his secretary can prove academic status to him! This is done using the path from $K5$ to $K7$ with label $prof$, then back to $K5$ with label $dean$, and finally, from $K5$ to $K6$ with label $admin$, which satisfies $(SELF : prof : ...)$.

We could use SDSI extended names to represent the previous example, as shown in Figure 4, but it is more difficult to scope the meaning of local name stu. The dean could create the local names $HighSchoolStu$ mapped to extended name $(K5\ admin\ stu)$ and $UniversityStu$ mapped to $(K5\ prof\ stu)$, and assign different access control roles to each of them. Unfortunately, the flexibility of extended names makes this fragile: if we have a new professor $K11$ in the department, he might assume that the secretary is maintaining the list of university students and just define his local name stu to map to the extended name $(K5\ admin\ stu)$. This completely defeats the access control measures of the dean, since now any high school student working for his secretary can prove $UniversityStu$. Note that the dean cannot control the length (or nature) of the delegation chain, and that syntactically-similar local names with different semantics are difficult to scope inside an extended name, because SDSI's name inference rules can produce non-explicit dependencies. The result is that most practical uses of SDSI extended names as authorization roles require a global agreement on each identifier they contain.

Let us compare our model with SDSI for this example. First, in our model the dean has tighter control of the delegation chain, because there are no name inference rules and every certificate in the chain must have a label matching a pattern in the SPC. This makes it easier to scope the meaning of local names in roles. Second, even though a professor could easily claim university student status to the dean, this is not explicit in SDSI. Third, the restriction to acyclic paths makes it very easy to implement our model on top of existing X.509 certificate path checkers/builders, whereas SDSI requires a completely new security infrastructure. However, in a more generic context the flexibility of SDSI names could be more important than the benefits of our approach.

When we started using SmartFrog for deploying a large number of components (see Section 5), we quickly realized that we needed a higher abstraction to represent sets of principals with consistent access control policies, rather than reasoning about each principal separately. For this purpose, we grouped principals into *domains* so that principals in the same *domain* have a common understanding on how to protect their critical resources. This common understanding relies on sharing an SPC that can consistently be proven to members of the *domain*. Typically, this SPC protects critical resources, allowing strong access to these only to principals who can prove the SPC. Note that a principal cannot in general change the local bindings of other members of the *domain*; to do that requires permission for a different type of access, which we call meta access. Two principals in the same *domain* may allow meta access to different sets of principals. Also, we want to designate special principals in a *domain* with resources that act as gateways to other domains, and to do that, we relax the access control policies to these resources, allowing a weak type of access, since gateways are designed in such a way that they cannot affect critical resources. Note that we could express every access control in terms of SPCs, allowing a fully distributed authorization model, but in this paper we de-couple how we form the domains from the authorization technology used.

To finish this section, we list some advantages and disadvantages of our new authorization framework. On the positive side, our authorization requirements are based on roles which do not need to be globally unique, and this avoids the creation of a complex parallel naming structure (as in X.509) or agreement on the global meaning of special names or tags (as in SDSI/SPKI).

There are also some limitations of this approach. The absence of name inference rules makes it more difficult to change authorization behaviour, and although by using SPCs we can express a large set of conditions on acyclic paths, we cannot allow access control specifications to contain arbitrary regular path expressions without access control computation becoming intractable (see Section 4). Also, our delegation is not composable: for example if $K2$ can prove $(SELF : a : b)$ to $K1$ and $K3$ can prove $(SELF : c)$ to $K2$, it is not necessarily true that $K3$ can prove $(SELF : a : b : c)$ to $K1$. Whether our scheme's limitations outweigh its benefits will depend on the application domain: but we had a good fit with the SmartFrog framework, since SmartFrog provides an easy way for a designer to set up and change access control lists, and tools help with the lack of composability.

4. System Model

In this section we give a system model. We show that it can be deduced in polynomial time whether one principal has access of a given type to another principal. We then discuss the *domains*, which are sets of principals with similar behaviour with respect to a specified type of access.

To define the system model, we start with a finite set A of labels used in bindings, and a set P of patterns over A. We require that the set of patterns P that will be used in the system model has the properties that (i) each pattern matches at least one $a \in A$, (ii) for $a \in A$, $p \in P$ there is a bound on the time it takes to determine whether or not a matches p, and (iii) for each $a \in A$ there is an element of P that is matched by $a' \in A$ if and only if $a' = a$. We let a (ambiguously) denote this element of P as well as the original element in A.

The system model will contain a finite nonempty set X of principals, identified by their public keys.

Now we use X and P to construct the set of expressions $\mathcal{L}$. Elements of $\mathcal{L}$ are $\vee$-combinations of the simple path constraints mentioned in Section 3, and will be used to specify access control lists.

Definition

$\mathcal{L}$ is the set of expressions whose syntax is given by

$$\mathcal{L} ::= ANYBODY \mid L_0 \mid L_0 : \ldots \mid L_0 \vee L$$

$$\mathcal{L}_0 ::= SELF \mid \ x \mid L_0 : p$$

$$\text{for } L \in \mathcal{L}\backslash\{ANYBODY\}, L_0 \in \mathcal{L}_0, x \in X, p \in P$$

(In order to ensure that there is no ambiguity between different elements of $\mathcal{L}$, we assume that SELF and ANYBODY are not members of P, and that the symbols ":", "$\vee$" and "..." do not occur as part of any element of P.)

In practice it is unrealistic to allow access control to be specified using elements of $\mathcal{L}$ containing arbitrarily many symbols. We assume that there is a fixed positive integer b such that only elements of $\mathcal{L}$ with b or fewer symbols can be used in access control specifications.

Definition The *system model* consists of a finite nonempty set X of principals, represented by their public keys; a set of patterns P over a finite set of labels A; a subset B of $(X \times A \times X)$ with $(x, a, y) \in B \Rightarrow x \neq y$ which will represent the bindings; a finite set AT of access types including two distinguished elements $META$ and $STRONG$; and a function AccessControl:$X \times AT \rightarrow \{L : \ L \in \mathcal{L}, L \text{ has at most } b \text{ symbols}\}$.

Any access type in AT other than $META$ and $STRONG$ is referred to generically as $WEAK$. Some shorthand which we will use in the rest of this section is $\mathcal{L}_1$, $SELF : s$ for $s \in P^*$, and $L[x/SELF]$. These are defined as follows.

Definition We write $\mathcal{L}_1$ for the set of elements of $\mathcal{L}$ which are of the form $SELF : s$ for some $s \in A^*$.

If $s = p_1.p_2 \ldots p_n \in A^*$ we write $SELF : s$ to mean $SELF : p_1 : \ldots : p_n$. (If s is the empty string, $SELF : s$ just means $SELF$.) We say that $a_1 \ldots a_n$ matches s iff a_i matches p_i for all $1 \leq i \leq n$.

If $L \in \mathcal{L}$ and $x \in X$, we write $L[x/SELF]$ for the elements of $\mathcal{L}$ obtained from L by substituting each appearance of $SELF$ in L by x.

The interpretation of B, the set of bindings, is that $(x, a, y) \in B$ iff the principal y can prove it satisfies a to x, using a binding.

4.1. Meanings of AccessControl expressions

The interpretation of AccessControl is that x gives permission to y for type t access iff either $x = y$, or y can prove to x that y satisfies $AccessControl(x,t)$, using a chain of bindings. In detail:

If $AccessControl(x,t) = y$ for some $y \in X$, this means that the only nodes permitted type t access to x are x itself and y.

If $AccessControl(x,t) = SELF$, this means that no node other than x itself is permitted t-level access to x.

If $AccessControl(x,t) = ANYBODY$ this means that x grants type t access to all principals.

The functions : p, for $p \in P$, model single-step delegation. If

$$AccessControl(x,t) \;=\; y : p_1 : \ldots : p_n$$

for some $n > 0$, this means that x grants type t access to z if and only if either $x = z$, or (there is a chain of bindings of length $m \leq n$ from y to z such that no principal appears more than once in the chain, and for $1 \leq i \leq m$, the label of the i^{th} binding in the chain matches p_i). For $s \in P^*$, $AccessControl(x,t) = SELF : s$ means the same as $AccessControl(x,t) = x : s$. Note that we allow the specification of either a fixed or a relative anchor of the chain of bindings, the relative anchor being $SELF$. Note also that if z can prove an access condition $SELF : s'$ to x for some initial substring s' of $s \in P^*$, then z can also prove access condition $SELF : s$ to x. (The same property holds if the relative anchor specification $SELF$ is replaced by a fixed anchor specification y).

In the examples so far a maximum depth of delegation is specified, but we are able to express certain types of requirements which allow unbounded delegation using expressions in $\mathcal{L}$ which contain the symbol "...". If

$$AccessControl(x,t) = y : s : \ldots$$

for some $s \in A^*$, this means that x grants type t access to z if and only if either $x = z$, or (there is a chain of bindings from y to z in which no principal appears more than once, such that the labels in the chain match an initial substring of s), or (for some $z' \in X$ there is a chain of bindings from y to z via z' in which no principal appears more than once, such that the labels in the chain from y to z' match the string s). The meaning of $AccessControl(x,t) = SELF : s : \ldots$ is the same as $AccessControl(x,t) = x : s : \ldots$ Thus in these access conditions the anchor and patterns for the labels for the first $length(s)$ levels of delegation are specified, but there is no restriction on the labels for subsequent delegation levels or the number of such levels.

Finally, $\vee$ has the usual logical-or semantics. We allow finite $\vee$-combinations of elements of $\mathcal{L}$ to be in $\mathcal{L}$ because in practice it is useful to increase the expressivity in this way. For our applications we sometimes would like x to allow type t access to y if y can satisfy one of a finite number of different specified conditions.

Note that for all permitted values of $AccessControl(x,t)$, x is allowed type t access to itself. It is a property of the systems that we model that a principal always has type t access to itself, for all access types t.

4.2. Access changes

If y has $META$ access to x then y can change the values of $B(x, \_, \_)$ and AccessControl$(x, _)$. If y has $STRONG$ access to x then y can perform sensitive operations on x, but cannot perform the changes allowed by $META$ access. The other access types describe different types of $WEAK$ access to x, including access to a gateway component in x. Notice that if for some principal x, $AccessControl(x,t)$ = $SELF$ for all access types t other than $META$ and $STRONG$, then only x itself is allowed $WEAK$ access to x, and hence x does not contain a gateway component.

Since x has $META$ access to itself, x can change its own access controls and bindings. However, in our application, principals are typically not allowed to decide autonomously which $META$-level changes (if any) to carry out. Instead, there is a system designer with a global view who understands the system and delegates the carrying out of these operations to the principals.

A malicious principal might autonomously decide to carry out changes at the $STRONG$ or $META$ level. Analyzing the general effects of a malicious principal is outside the scope of this paper. However, the way that access control is specified takes into account one particular malicious behaviour, which is the *sock-puppet attack* mentioned in Section 3. In a sock-puppet attack, a principal z creates another principal (the sock-puppet) completely under the control of z, creates bindings from z to the sock-puppet, and exploits the fact that any access permissions granted to the sock-puppet are effectively granted to z. Suppose principal z uses a sock-puppet to gain access of a particular type, by exhibiting a chain of bindings which match a string $s \in P^*$ from a specified anchor principal to the sock-puppet. Since the sock-puppet can only have bindings to it from z or from another sock-puppet of z, this chain of bindings must pass through z. Therefore there is a chain of bindings from the anchor principal to z which matches an initial substring of s. But this means that z could use this chain of bindings to gain access, without having to use the sock-puppet. So our method of specifying access control ensures that z cannot use sock-puppets to gain any access permissions that z would not have without using a sock-puppet.

4.3. Definition of $\rightarrow_L$

The purpose of this subsection is to set up definitions which we will use later on, in particular the definition of the relation $\rightarrow_L$ for $L \in \mathcal{L}$, which will be used to model the access control for the system. We will show later (in Lemmas 1 and 3) that the set of bindings B is characterized by the formulae $x \rightarrow_L y$ that it models, and y has type t access to x if and only if B models $x \rightarrow_{AccessControl(x,t)} y$.

The relation $\rightarrow_L$ is easier to define in the case $L \in \mathcal{L}_1$ than in general. If $L = SELF : p_1 : \ldots : p_n$, and $x \neq y$, then B models $x \rightarrow_L$ iff there is an acyclic chain of bindings from x to y whose labels match patterns $p_1, \ldots, p_n$ in order. We begin our construction of $\rightarrow_L$ by defining a simpler relation, $\hookrightarrow_{L1,Y}$ for $L1 \in \mathcal{L}_1, Y \subseteq X$, where if $x \neq y$ and $LI \in \mathcal{L}_1$ then $x \rightarrow_{L1} y$ is equivalent to ($x \hookrightarrow_{L1,Y} y$ for some Y), and the subscript Y records the set of principals through which the chain passes.

Given B, $x, y \in X, Y \subseteq X$, and $L_1 \in \mathcal{L}_1$, we define $B \models (x \hookrightarrow_{L_1,Y} y)$ if and only if this can be derived using the following derivation rules.

SELF:	$(x \in X) \Rightarrow (B \models x \hookrightarrow_{SELF,\{x\}} x)$
Binding:	$(B \models x \hookrightarrow_{L_1,Y} y, (y,a,z) \in B,$
	a matches $p \in P, z \notin Y)$
	$\Rightarrow (B \models x \hookrightarrow_{L_1:p,Y\cup\{z\}} z)$

It is straightforward to check by induction on n that $(B \models x \hookrightarrow_{SELF:p_1:\ldots:p_d,Y} y) \Leftrightarrow$ (there are some $x = x_1, x_2, \ldots x_{d+1} = y \in X$, all distinct, such that Y=$\{x_1 \ldots, x_{d+1}\}$, and for all $1 \leq i \leq d$, $(x_i, a_i, x_{i+1}) \in B$ for some $a_i \in A$ matching p_i.)

Now we use $\hookrightarrow_{L_1,Y}$ to define the relation $\rightarrow_L$ for each $L \in \mathcal{L}$. If $L \in \mathcal{L}$ and $x, y \in X$, define $B \models x \rightarrow_L y$ if and only if it this can be derived using the two rules above, together with following additional rules.

Drop-the-set:	$(B \models x \hookrightarrow_{L_1,Y} y)$
	$\Rightarrow (B \models x \rightarrow_{L_1} y)$
Sock-puppet:	$(L_1 \in \mathcal{L}_1, p \in P, B \models x \rightarrow_{L_1} y)$
	$\Rightarrow (B \models x \rightarrow_{L_1:p} y)$
Dots-initial:	$(L_1 \in \mathcal{L}_1, B \models x \rightarrow_{L_1} y)$
	$\Rightarrow (B \models x \rightarrow_{L_1:\ldots} y)$
Dots-substring:	$(L_1 \in \mathcal{L}_1, p \in P, B \models x \rightarrow_{L_1:p\ldots} y)$
	$\Rightarrow (x \rightarrow_{L_1\ldots} y)$
Identity:	$(x \in X, L \in \mathcal{L}) \Rightarrow (B \models x \rightarrow_L x)$
Public-key:	$(B \models x \rightarrow_L y, L = L_1$ or
	$L = L_1 : \ldots, L_1 \in \mathcal{L}_1, z \in X)$
	$\Rightarrow (B \models z \rightarrow_{L[x/SELF]} y)$
Anybody:	$x, y \in X$
	$\Rightarrow (B \models x \rightarrow_{ANYBODY} y)$
Or:	$(L_1 \vee L_2 \in \mathcal{L}, B \models x \rightarrow_{L_1} y$
	or $B \models x \rightarrow_{L_2} y)$
	$\Rightarrow (B \models x \rightarrow_{L_1 \vee L_2} y)$

We now show that B is characterized by the set of formulae $x \rightarrow_L y$ that it models. If B is a binding, write $formulae(B)$ for the set of functions $x \rightarrow_L y$ with $x, y \in X, L \in \mathcal{L}$ such that $B \models x \rightarrow_L y$.

Lemma 1 If B, B' are bindings, i.e., they are subsets of $X \times A \times X$ such that $((x,a,y) \in B$ or $\in B') \Rightarrow x \neq y$, then

$$B = B' \Leftrightarrow formulae(B) = formulae(B')$$

Proof $\Leftarrow$ is trivial. To prove $\Rightarrow$, consider the formula $x \rightarrow_{SELF:a} y$, where $x \neq y \in X, a \in A$. Suppose that $B \models x \rightarrow_{SELF:a} y$. (Recall that a refers ambiguously to the element of A and the element of P which is matched by only this element of A.) Then there is a derivation of $B \models x \rightarrow_{SELF:a} y$ using the derivation rules. The final step of the derivation must use the Drop-the-set rule or the Sock-puppet rule. If it uses the Sock-puppet rule, the derivation must have previously proved $x \rightarrow_{SELF} y$; but it is not possible to prove this, since $x \neq y$. Therefore, the final step must have used the Drop-the-set rule, and the derivation must have previously proved $B \models x \hookrightarrow_{SELF:a,Y} y$ for some $Y \subseteq X$. In turn, the final step of the derivation of $B \models x \hookrightarrow_{SELF:a,Y} y$ must use the Binding rule and the fact that $((x,a,y) \in B)$; if $(x,a,y) \notin B$ then there is no way to derive $B \models x \hookrightarrow_{SELF:a,Y} y$. Conversely, if $(x,a,y) \in B$ then there is a derivation of $B \models x \rightarrow_{SELF:a} y$ using the derivation rules SELF, Binding and Drop-the-set. Therefore we have $B = \{(x,a,y) : x \neq y \in X, a \in A, (B \models x \rightarrow_{SELF:a} y)\}$, and the result follows. □

4.4. Graph interpretation of $\models$

We now give a description of $\models$ in terms of finding acyclic paths in a labelled directed graph.

Lemma 2 Suppose $s = p_1, \ldots, p_d \in P^*$. Let $G(B,s)$ be the directed graph whose set of nodes is X, with edges labelled with elements of $\{p_1, \ldots p_d\}$, such that for all $x_1, x_2 \in X$ and $1 \leq i \leq d$ there is a (unique) directed edge from x_1 to x_2 labelled p_i if and only if $((x_1, a, x_2) \in B$ for some a matching p_i). Suppose $x \neq y \in X$. Then:
(i) $B \models x \hookrightarrow_{SELF:s,Y} y$ for some $Y \subseteq X \Leftrightarrow$ (there is an acyclic directed path from x to y in $G(B,s)$ whose labels in order form the string s).
(ii) $B \models x \rightarrow_{SELF:s} y \Leftrightarrow$ (there is an acyclic directed path from x to y in $G(B,s)$ whose labels in order form an initial substring of s).
(iii) $B \models x \rightarrow_{SELF:s:\ldots} y \Leftrightarrow$ (there is an acyclic directed path from x to y in $G(B,s)$ whose labels in order form a string which is either an initial substring or an initial superstring of s).
(iv) For all $z \in X$, $B \models x \rightarrow_{z:s} y \Leftrightarrow$ (there is an acyclic directed path from z to y in $G(B,s)$ whose labels in order form an initial substring of s).
(v) For all $z \in X$, $B \models x \rightarrow_{z:s:\ldots} y \Leftrightarrow$ (there is an acyclic directed path from z to y in $G(B,s)$ whose labels in order form a string which is either an initial substring or an initial

superstring of s).
(vi) $B \models x \rightarrow_{ANYBODY} y$.
(vii) If $L_1 \vee L_2 \in \mathcal{L}$, then $B \models x \rightarrow_{L_1 \vee L_2} \Leftrightarrow (B \models x \rightarrow_{L_1} y$ or $B \models x \rightarrow_{L_2} y)$

Proof It is easy to prove parts (i), (ii) by induction on the length of s. For part (iii), observe that the final steps in any derivation of $B \models x \rightarrow_{SELF:s:\ldots} y$ must be an application of Dots-initial rule followed by zero or more applications of the Dots-substring rule. Therefore $B \models x \rightarrow_{SELF:s:\ldots} y$ $\Leftrightarrow$ ($B \models x \rightarrow_{SELF:s'} y$ for some initial superstring s' of s). Notice that a string s'' is an initial substring of an initial superstring of s iff s'' is either an initial substring of s or an initial superstring of s. Hence part (iii) follows from part (ii). To prove parts (iv), (v), observe that $B \models x \rightarrow_{z:s} y$ $\Leftrightarrow$ ($x = y$ or $B \models z \rightarrow_{z:s} y$) and that $B \models x \rightarrow_{z:s:\ldots} y$ $\Leftrightarrow$ ($x = y$ or $B \models z \rightarrow_{z:s:\ldots} y$). Therefore parts (iv), (v) follow from parts (ii), (iii). Parts (vi),(vii) are trivial to prove. □

It follows directly from Lemma 2 and Subsection 4.1 that $\rightarrow_L$ correctly models the access control, in the sense that the following holds:

Lemma 3 For all $x, y \in T, t \in AT$, y has type t access to x if and only if $B \models (x \rightarrow_{AccessControl(x,t)} y)$.

4.5. Tractability

We have shown that given $x, y \in X$ and $t \in AT$, the question of whether or not y has type t access to x depends on whether there is an acyclic directed path from x to y with a particular property in a particular labelled directed graph. Mendelzon and Wood show in [16] that for some properties that can be expressed as regular expressions on the set of labels, (for instance, the property that the path has an even length), the problem of determining in a labelled directed graph whether there is an acyclic directed path with the property is NP-complete. Mendelzon and Wood's approach to this issue is to restrict the set of properties and/or the set of graphs considered, to ensure that if there is any path containing cycles with this property then there is an acyclic path with the property, and then use polynomial-time algorithms to find a path (possibly containing cycles) with the property. However, this approach is not useful for us, because for example in Figure 3 there is a path containing a cycle from the dean to the secretary whose first label is $prof$, but we do not want the secretary to have the same permissions as the department members at the end of an acyclic path from the dean whose first label is $prof$. In this subsection we prove that provided the access condition AccessControl(x, t) is in the restricted set of expressions $\mathcal{L}$, the problem of deciding whether y has type t access to x can be decided in polynomial time.

In [4] there is a polynomial-time algorithm for deciding whether there is a SPKI/SDSI certificate chain authorizing a client to access a resource. Unfortunately the certificate chain found may contain cycles, and as a result it is not possible to use a direct analogy of the algorithm in [4] in our context.

Our proof will use the following two general Lemmas showing that certain kinds of acyclic paths in a directed graph can be found in polynomial time. It is not difficult to prove them by induction on d.

Lemma 4 For fixed $d \geq 0$ there is an $O(n^{max\{d,1\}})$ algorithm to solve the following problem. Given a directed graph G with node set $X = \{x_1, \ldots x_n\}$, labelled using a finite label set in such a way that there are no two edges with the same label between the same ordered pair of nodes, and a string s of labels with $length(s) = d$, construct the Boolean function $Sub(G, x_i, s)$ on X, where $Sub(G, x_i, s)[x_j]$=**T** iff there is an acyclic directed path in G from x_i to x_j such that the labels of the edges of the path in order form an initial substring of s.

Lemma 5 As Lemma 4, but with "$O(n^{max\{d,1\}})$" replaced by "$O(n^{d+2})$", "$Sub(G, x_i, s)$" replaced by "$Super(G, x_i, s)$", and "an initial substring of s" replaced by "an initial substring or an initial superstring of s".

The following definition will be handy for stating the results of this subsection.

Definition For $L \in \mathcal{L}$, the *explicit delegation depth* $d(L)$ of L is given by the following equations:
1. $d(ANYBODY) = 0$
2. If $L = L_1 \vee L_2$ then $d(L)$ =max$\{d(L_1), d(L_2)\}$
3. If $s \in P^n, n \geq 0, x \in X$ then $d(SELF : s) = d(SELF : s : \ldots) = n$
4. If $L = L'[x/SELF], L' \in \mathcal{L}_1$ then $d(L) = d(L')$

Now we can state a result from which the main result of this subsection follows immediately. Note that in our applications our graphs are generally sparse, and the order of the time necessary to calculate the functions described is considerably lower than the upper bound given in Lemma 6. In this paper we are not trying to obtain the best bounds possible, only to obtain polynomial bounds.

Lemma 6 For fixed integers d, d', there is an $O(|X|^{d+2})$ algorithm for constructing the function $M_{x,L} : X \rightarrow$ {**T**,**F**} such that $M_{x,L}(y)$=**T** iff $x \Rightarrow_L y$, for any $x \in X$ and any expression L in $\mathcal{L}$ such that the explicit delegation depth of L is at most d and L contains at most d' instances of the symbol $\vee$.

Proof If $d = 0$ then L is either $ANYBODY$, $SELF$, or x for some $x \in X$, and the result follows easily. So assume $d > 0$; in particular, $d(L) = d$ implies $L \neq ANYBODY$. Any $L \in \mathcal{L}$ such that $d(L) = d$ and L contains at most d' instances of the symbol $\vee$ can be expressed in the form $L_1 \vee L_2 \vee \ldots L_e$ for some $1 \leq e \leq d' + 1$, such that $d(L_i) \leq d$ for all i, and each L_i is either in $\mathcal{L}_1$ or is equal to $L'_i[z_i/SELF]$ for some $z_i \in X, L'_i \in \mathcal{L}_1$. By part (vii) of Lemma 2, $M_L = M_{L_1} \vee \ldots \vee M_{L_e}$, so it suffices to prove the case $e = 1$.

If $L = L'[z/SELF]$ for some $L' \in \mathcal{L}_1, z \in X$ we have $M_{x,L}(x, y)$=**T** $\Leftrightarrow x \rightarrow_{L'[z/SELF]} y \Leftrightarrow z \rightarrow_{L'} y \Leftrightarrow M_{z,L'}(y)$=**T**. So it suffices to prove the case where L is restricted to be in $\mathcal{L}_1$, i.e., to show that there is an $O(|X|^{d+2})$ algorithm to construct $M_{x,L}$ for any $x \in X$ and any L of the form $L = SELF : s$ or $SELF : s : \ldots, s \in P^*$. By induction on d we can assume that $d(L) = d$, i.e., $s \in P^d$.

Let $s = p_1 \ldots p_d \in P^d$. Let $G(B, s)$ be the graph with nodes X and a (unique) edge from x to y labelled p_i $(1 \leq i \leq d)$ if and only if $(x, a, y) \in B$ for some $a \in A$ matching p_i. Since A is a fixed finite set and there is an $O(1)$ bound on the length of time it takes to determine whether a matches p_i, there is an $O(|X|^2)$ bound on the length of time it takes to construct $G(B, s)$. Part (ii) of Lemma 2 implies that determining whether $B \models x \rightarrow_L y$ is equivalent to determining whether there is an acyclic graph from x to y in $G(B, s)$ whose labels in order match an initial substring of s. It follows by Lemma 4 that once $G(B, s)$ has been constructed, there is an $O(|X|^{max\{d,1\}})$ algorithm for determining $M_{x,SELF:s}$

Similarly, it can be shown using part (iii) of Lemma 2 and Lemma 5 that given $G(B, s)$, there is an $O(|X|^{d+2})$ algorithm for determining $M_{SELF:s:\ldots}$, and the result follows. □

The main result of this subsection follows immediately from Lemmas 6 and 3.

Theorem Given principals x, y and access-type t, it can be decided in polynomial time whether y has access of type t to x.

4.6. Domains

When reasoning about the security of the system, it is useful to have an abstraction level which allows reasoning about sets of principals with similar behaviour, instead of having to consider each principal separately. In this section we define the *domains*, which are sets of elements all of whom behave in a similar way with respect to a specified type of access.

For $t \in AT, x, y \in X$, write $Acc(x, t, y)$ for the Boolean which is equal to **T** iff x permits access type t to y. Lemma 3 expresses $Acc(x, t, y)$ in terms of $B, \models$. However, for all the definitions and lemmas in this subsection other than Lemma 8, we will not use B or $\models$ directly; we will only use Acc. So these definitions and lemmas can still be used if different derivation rules are used, or B is changed, or indeed an arbitrary semantics is given to Acc - provided that Acc satisfies $Acc(x, t, x)$ =**T** for all x and t. (Recall that for every $x \in X, t \in AT$, x permits access of type t to itself.)

For $t \in AT$, we define the relation $\sim_t$ on X as follows.

Definition If $x, y \in X, t \in AT$, then $x \sim_t y$ if and only if the following three properties all hold:

(**In-edge consistency**)

$$\forall z \in X,\ Acc(z, t, x) = Acc(z, t, y)$$

(**Out-edge consistency**)

$$\forall z \in X,\ Acc(x, t, z)\} = Acc(y, t, z)$$

(**Scope consistency**)

$$AccessControl(x, t) = AccessControl(y, t)$$

In-edge consistency means that for all z, x has type t access to some z if and only if y has type t access to z. Out-edge consistency means that for all z, z has type t access to x if and only if z has type t access to y. Scope consistency means that x and y use a syntactically identical simple path constraint to specify which principals have type t access to them. We impose the scope consistency requirement for $x \sim_t y$ because it assists with the visualization of the access properties of the system, making them easier for a system designer to understand, and because (except in worst cases) it allows pruning of some search spaces involved in finding *domains* and checking security properties.

It is easy to check that the relation $\sim_t$ on X satisfies reflexivity, symmetry and transitivity, that is, it is an equivalence relation. Now we can define the t-domains.

Definition The t-domains are the equivalence classes of X under $\sim_t$.

Clearly, each principal is in exactly one t-domain. Moreover, the t-domains are consistent, in the following way:

Lemma 7 If $x \sim_t y$ then x, y will agree about the members of their t-domain, in the sense that $x \sim_t y$, $z \in X$ implies ($x \sim_t z$ iff $y \sim_t z$).

Proof Follows immediately from the fact that $\sim_t$ is an equivalence relation.

Since $Acc(x, t, x)$ holds for all x and all t, it follows by in-edge consistency that if x and y are in the same t-domain then $Acc(x, t, y)$.

We can think of a t-domain as a set of principals which all behave in the same way with respect to access of type t. This abstraction can be useful to visualize and reason about the behaviour of the system with respect to this type of access. (For use in our implementations, we are particularly interested in the $STRONG$-domains and the $META$-domains). Principals in the same t-domain will in general have different behaviour with respect to other types of access. For example, a web server may allow low-level access to entities outside the $STRONG$-domain of the web server, but these entities in general will not be able to access web pages from other nodes in the same $STRONG$-domain as the web server.

We now show, using Lemma 3, that given t it can be decided in polynomial time what the t-domains are. We assume that AT, A and b remain fixed as X grows. Once again we note that for our applications, the order of time necessary to do these calculations is considerably lower than the upper bound given here.

Lemma 8 Let $t \in AT$. Set $d = max\{d(L) : L = AccessControl(x, t'), x \in X, t' \in AT\}$. There is an $O(|X|^{d+3})$ algorithm to calculate the t-domains.

Proof Recall that there is a fixed bound b on the number of symbols in an expression L such that $L = AccessControl(x, t')$ for some $x \in X, t' \in AT$. The explicit delegation depth of an expression is never greater than the number of symbols in the expression, and it follows that d is well-defined. For $x, y \in X, t' \in AT$, checking whether or not $AccessControl(x, t) = AccessControl(y, t)$ involves checking identity of at most b pairs of symbols, and since $|AT|$ is fixed it follows that there is an $O(|X|^2)$ algorithm to check this for all $x, y \in X$. By Lemmas 3,6, there is an $O(|X|^{d+3})$ algorithm to calculate $Acc(x, t, y)$ for all $x, y \in X$.

Once all the values $Acc(x, t, y)$ have been calculated, determine the t-domain containing x for each $x \in X$, and the set of unique t-domains, as follows. Order the elements of X as $x_1, \ldots x_{|X|}$. Set $Reps$, a list of t-domain representatives, to be (initially) equal to the string just containing some x_1, and set $Domain(x_1) = 1$. Do the following for $1 \leq i \leq |X|$: (Check whether there is $j \in Reps$ such that (x_i, x_j) satisfy the three conditions for $x_i \sim_t x_j$ given in Definition 4.6. If so, set $Domain(x_i) = j$. If not, set $Domain(i) = i$ and append i to $Reps$.) The t-domains are the sets $\{x_j : Domain(j) = i\}$ for $i \in Reps$.

Everything after the calculation of the values of $Acc(x, t, y)$ can be done in $O(|X|^2)$ time. The result follows. □

5. Implementation Issues

We have implemented the authorization model described in this paper and used it successfully to secure SmartFrog [10]. SmartFrog is an open source framework, written in Java, for managing the life-cycle of distributed components. SmartFrog provides a language, with powerful templating and linking capabilities, to describe name/value pairs associated with attributes of these components. (If you do not like the default description language, it is very easy to plug another description language, for example XML, into the framework.) SmartFrog also provides a component model that defines a simple life-cycle for individual components, and describes how they compose to produce distributed work-flows that might, for example, coordinate the starting or stopping of a complex distributed application. Moreover, it also provides a fully-distributed deployment engine, in which a peer creates local components according to descriptions, and forwards to other peers partial descriptions that detail other components that need to be deployed somewhere else. There is no centralized control; any peer could deploy or forward any description. When a hierarchy of components is created, possibly across multiple nodes, an implicit directory-based service is created that allows users to navigate the hierarchy, regardless of where the components are deployed. The framework also provides mechanisms to link these hierarchies, and these mechanisms are independent of the implementation of the components.

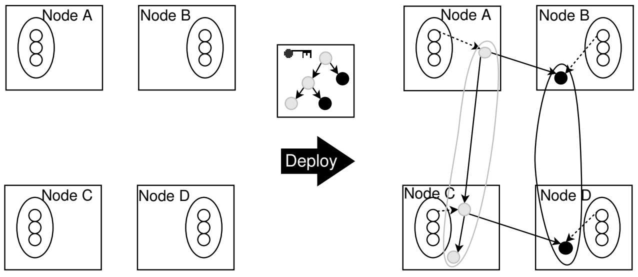

Figure 5. SmartFrog deployment example.

Figure 5 shows how the authorization model fits into the SmartFrog context. We start with a set of nodes, each running an instance of the deployment engine, and identified by its public key. These correspond to the principals in the system model. Note that these principals do not have any trust relationship between themselves, but they each have an owner, which is another principal not shown in Figure 5 that has $META$ access to them. This owner will write and sign a description that annotates components to be deployed with security attributes expressed as SPCs, and the result is that $STRONG$-domains will be formed dynamically. For example, in Figure 5 after deployment of the signed description there are two $STRONG$-domains, one containing nodes A and C and the other one nodes B and D. These domains could use existing local bindings that have been preconfigured statically, or some of the deployed components could create new bindings.

So what do $META$, $STRONG$ and $WEAK$ access correspond to in SmartFrog? All locally-deployed components are allowed to interact between themselves freely, so they should all have the same $STRONG$ and $META$ access control lists governing access from the outside world. There are four main types of controlled external interactions. The first two types are that a component can export remote methods that can be called by external components (using RMI [18]); and that a component can call remote methods invoked on external components that have exported them. These two types of interactions have the potential to carry out sensitive operations on the components, but not to remotely change local bindings, so they are allowed if the external and local components have permission for $STRONG$ access to each other. However, we can create gateways between $STRONG$-domains which are set up by identifying a special component that will play that role, and this allows these interactions if the participating principals have $WEAK$ access to each other, and one of them contains a gateway component. This component exports a subset of its remote methods to principals in other $STRONG$-domains, and it can also make controlled external calls to components in other $STRONG$-domains. Extra care is taken in ensuring that these remote methods will not give access to critical resources or carry out sensitive operations on the components. The second two types of controlled external interactions are that a component can load external code or files; and that a component can locally deploy a SmartFrog description, which can have the effect of creating new components or changing local bindings. These may change the set of local bindings, and so are allowed only if the code, files or description (respectively) are signed with the key of a principal which is allowed $META$ access to the local component.

The signing of SmartFrog descriptions relies on a recursive canonicalization process that allows a sub-description to be replaced by its hash without affecting signature validation. This allows intermediate peers to forward partial descriptions without re-signing them. SmartFrog also provides mechanisms to allow trusted third parties to make controlled changes to sub-descriptions, with their own self-contained signature. The trusted third parties are specified in immutable parts of the description; there is language support for specifying which parts are immutable, and who can change the customizable parts. See [12] for details.

The mapping of exported local bindings onto X.509 certificates is straightforward. We do not use X.509 Distinguished Names (DNs) for authorization. In the place of the DN in the certificate we put a string derived from a hash of the appropriate principal's public key. We add the edge label to one of the standard fields available in the X.509 certificate format. We always validate the certificates first using a standard X.509 certificate chain checker (this allow us to use standard revocation methods), then we check there are no cycles in the path, and we extract in order the labels from the certificate chain to form a sequence of labels for the path which are matched to the access control lists as described in Section 4. We have customized Java SSL, class loaders, and jar file verification libraries to perform these extra checks.

6. Conclusions

In this paper we have presented a new trust management scheme, which uses delegation with acyclic paths. We have given a formal definition for the scheme and for a higher-level abstraction, the domains. We have also described our implementation in the context of SmartFrog, a management framework for distributed components.

Our future plans are to build tools based on the formal definition in order to analyse the effects of different security policies, and to assist the design of a system satisfying a given security requirement.

Acknowledgements

We would like to thank the other members of the Serrano team for critical support in the development of this research. In particular, Patrick Goldsack, as a technical lead for the Serrano team, heavily influenced most aspects of this work. We also would like to thank Brian Monahan from HPL-TSL for earlier discussions on formal modelling and tool support for SmartFrog security.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, Sept. 1999.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. 17th Symposium on Security and Privacy*, pages 164–173, Los Alamitos, 1996. IEEE Computer Society Press.

[3] CCITT (Consultative Committee on International Telegraphy and Telephony). *Recommendation $X.509$: The Directory—Authentication Framework*, 1988.

[4] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, November 2001.

[5] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas. Internet X.509 Public Key Infrastructure: Certification Path Building. RFC 4158, Sept. 2005.

[6] C. Ellison and B. Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. *Computer Security Journal*, 16(1), 2000. http://www.schneier.com/paper-pki.pdf.

[7] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet Engineering Task Force Draft IETF, July 1997.

[8] S. Farrell and R. Housley. An Internet Attribute Certificate Profile for Authorization. RFC 3281 (Proposed Standard), Apr. 2002.

[9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-98)*, pages 83–92, New York, Nov. 3–5 1998. ACM Press.

[10] Hewlett-Packard. *The SmartFrog Reference Manual*, July 2005. http://www.smartfrog.org.

[11] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In S. Jajodia and P. Samarati, editors, *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)*, pages 190–199, N.Y., Nov. 1–4 2000. ACM Press.

[12] A. Lain. *Using the new SmartFrog Security*. Hewlett-Packard, Jan. 2006. http://www.smartfrog.org.

[13] B. W. Lampson. Protection. In *5th Princeton Symposium on Information Sciences and Systems,*. Princeton University, Mar. 1971. Reprinted in Operating Systems Review 8,1 January 74.

[14] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 114–130, Oakland, CA, May 2002. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[15] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In P. Samarati, editor, *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 156–165, Philadelphia, PA, USA, Nov. 2001. ACM Press.

[16] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[17] OASIS Open. OASIS eXtensible Access Control Markup Language (XACML) TC, 2006.

[18] R. Oberg. *Mastering RMI: Developing Enterprise Applications in Java and EJB*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[19] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, Apr. 1996. SDSI Version 1.0.

Appendix: Yet Another University Example

Figure 6 shows a simplified example of how we could manage interactions between professors, university students, teaching assistants (TAs) and the dean in a university department. The dean $P1$ is in a $STRONG$-domain defined by the SPC $(SELF : dean)$, and can prove this SPC to all other principals since they all have a local binding with label $dean$ to $P1$. The dean defines who are valid professors, ($P2, P3$ and $P4$ in the Figure), by creating local bindings with label $prof$. The professors form another $STRONG$-domain with scope $(SELF : dean : prof)$. Professors define who are their students ($P5$ and $P6$), and TAs ($P7$ and $P8$), by creating local bindings with labels stu and $ta\_course\_$, respectively. Note how professors use a simple convention to generate local bindings for the TAs that allows customization based on the course they are teaching while respecting a pattern that is easy to match for all TAs. For example, if TA $P7$ teaches course 101 and TA $P8$ teaches course 211, and their corresponding professors assigned them labels ta_101_ and ta_211_, respectively, which are both easily matched by the pattern $ta\_*\_$. The students form a $STRONG$-domain with scope $(SELF : dean : prof : stu)$ and the TAs form another $STRONG$-domain with scope $(SELF : dean : prof : ta\_*\_)$. Note that a student or a TA has $STRONG$ access to any other

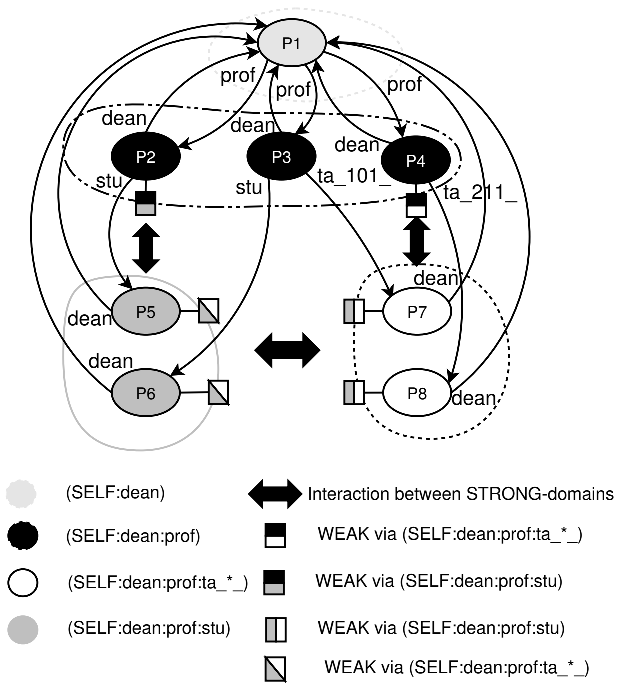
P1
prof
prof
prof
dean
dean
dean
P2
P3
P4
stu
stu
ta_101_
ta_211_
dean
P5
P7
dean
P6
P8
dean
dean
(SELF:dean)
Interaction between STRONG-domains
(SELF:dean:prof)
WEAK via (SELF:dean:prof:ta_*_)
(SELF:dean:prof:ta_*_)
WEAK via (SELF:dean:prof:stu)
(SELF:dean:prof:stu)
WEAK via (SELF:dean:prof:stu)
WEAK via (SELF:dean:prof:ta_*_)

Figure 6. Extended university example.

principal with the same role, regardless of who is her professor. We assume that the professors have $META$ access to the students and TAs, and that the dean has $META$ access to everyone. The natural hierarchy of $STRONG$ access between $STRONG$-domains is implicitly derived in this example by our choice of SPCs for $STRONG$ access control.

So what $WEAK$ interactions are allowed between principals? The dean is too busy to be interrupted by anybody. The professors have designated one special gateway professor, $P2$, to deal with student issues, and another, $P4$, to deal with TA issues. The students and TAs are allowed $WEAK$ access to the relevant gateway professor, for example by submitting requests using a safe interface. All students allow all TAs $WEAK$ access to send them assignments and similarly, all TAs allow all students $WEAK$ access to receive completed assignments from them.

Table 1 lists $AccessControl(x, t)$ for all principals x and access types t. Note that $WEAK$ access restricted to $(SELF)$ means that a principal is not acting as a gateway.

Students could further refine the $WEAK$ interactions that they have with TAs by using the previously-ignored course fields in the labels. For example, students may want to ensure that they only receive assignments from TAs that are teaching the courses that they are attending. This can be easily implemented by using the complete label to define a new $WEAK$ access control enforced by students,

Table 1. **Access controls in the university department example.**

Principal	Description	$STRONG$	$META$	$WEAK$
$P1$	Dean	(SELF:dean)	(SELF)	(SELF)
$P2$	Professor	(SELF:dean:prof)	(SELF:dean)	(SELF:dean:prof:stu)
$P3$	Professor	(SELF:dean:prof)	(SELF:dean)	(SELF)
$P4$	Professor	(SELF:dean:prof)	(SELF:dean)	(SELF:dean:prof:ta_*_)
$P5$	Student	(SELF:dean:prof:stu)	(SELF:dean:prof)	(SELF:dean:prof:ta_*_)
$P6$	Student	(SELF:dean:prof:stu)	(SELF:dean:prof)	(SELF:dean:prof:ta_*_)
$P7$	TA	(SELF:dean:prof:ta_*_)	(SELF:dean:prof)	(SELF:dean:prof:stu)
$P8$	TA	(SELF:dean:prof:ta_*_)	(SELF:dean:prof)	(SELF:dean:prof:stu)

for example a student x attending only course 101 could set $AccessControl(x, WEAK)$ to be $(SELF : dean : prof : ta\_101\_)$. The TAs always have certificates containing a fully specified label, so they do not need to be aware of this refinement in the access control imposed by the student.

It can be argued that this simple example is just a hierarchical structure that does not require the handling of arbitrary graphs. However, note how the fact that labels do not need to be locally unique decouples knowledge that a principal is a student from knowledge of who their professor is, and how the substring rule implicitly models the department hierarchy. Using syntactic conventions on the labels allows us an alternative mechanism for describing the structure of the trust relationships between principals. Moreover, if students limit $META$ access to themselves, they can overlay a completely independent friends-and-family web of security relationships that use the same public/private pair, but cannot be affected by professors (or the dean!). Achieving these properties in other security frameworks is not simple. In particular, if the students want an overlay independent of the dean in a framework using X.509-style Distinguished Names (DNs), they will probably have to acquire different DNs from the ones provided by the school.