Path Conditions and Principal Matching: A New Approach to Access Control

Jason Crampton
Royal Holloway University of London
Egham, United Kingdom
jason.crampton@rhul.ac.uk

James Sellwood
Royal Holloway University of London
Egham, United Kingdom
james.sellwood.2010@live.rhul.ac.uk

ABSTRACT

Traditional authorization policies are user-centric, in the sense that authorization is defined, ultimately, in terms of user identities. We believe that this user-centric approach is inappropriate for many applications, and that what should determine authorization is the relationships that exist between entities in the system. While recent research has considered the possibility of specifying authorization policies based on the relationships that exist between peers in social networks, we are not aware of the application of these ideas to general computing systems. We develop a formal access control model that makes use of ideas from relationship-based access control and a two-stage method for evaluating policies. Our policies are defined using path conditions, which are similar to regular expressions. We define semantics for path conditions, which we use to develop a rigorous method for evaluating policies. We describe the algorithm required to evaluate policies and establish its complexity. Finally, we illustrate the advantages of our model using an example and describe a preliminary implementation of our algorithm.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; H.2.0 [**Database Management**]: General—*Security, integrity and protection*

General Terms

Security, Design, Language, Theory

Keywords

access control; path condition; relationship; principal matching; authorization

1. INTRODUCTION

Access control is an essential security service in any multi-user computer system. It provides a mechanism by which different users are restricted in the actions they can perform within the system. An access control service typically comprises a policy decision point and a policy. An attempt by a user to interact with a system resource, usually known as an authorization request, is evaluated by the policy decision point and is only permitted if that interaction is authorized by the policy.

An access control model provides a syntax for authorization policies and a specification of the algorithm used by the policy decision point to evaluate requests. Many access control models focus on the user and authorizing the user to perform particular actions. As is customary in the literature, we will use the terms *subjects* and *objects* when referring to the parties who are to, respectively, perform and be the target of authorization (inter)actions.

Access control has been the subject of significant research and development in the last 40 years. As our use of technology and the connectivity of our devices has increased, the need for ever more robust and scalable access control models has also grown. New models attempt to improve on the failings of their predecessors, and often do so by redefining the policy foundations upon which authorization decisions are made. The protection matrix model, for example, simply enumerated all authorized actions. While this provides for precise specification of authorization policies, it does not scale well and is difficult to manage. In order to ease this administrative burden, various improvements have been employed by modern operating systems. The Unix operating system, for example, replaces the individual subjects with a mapping, performed at the time of request evaluation, to one of three security principals (owner, group and world) [6]. In this way, whilst each object must still be enumerated, the enumeration of subjects is limited to just these three security principals. With complex systems involving numerous users, this design dramatically reduces the space and administrative complexity of the underlying policy. However, it also greatly reduces the flexibility afforded when compared with defining authorization at the user level.

Role-based access control (RBAC), which is widely used and has been the subject of extensive research in recent years, assigns a user to one or more organizational roles. These roles are then authorized to perform certain actions on particular resources. These roles, which are defined on a per-system basis, thus reduce the administrative burden of the protection matrix (assuming the number of roles is

significantly less than the number of users), and provide a level of flexibility not available within the Unix model. This increased flexibility also restores some of the clarity that was lost when users where abstracted behind Unix's three, very general, security principals.

A significant disadvantage with RBAC is that it takes no account of the specific relationship that might exist between a user and the resource for which access is requested. Thus every user assigned to a doctor role can access all electronic health records if the doctor role is authorized to do so. Clearly, it would more appropriate if the only users that are authorized to access a particular health record have a specific relationship with the subject of that record. In short, RBAC is not as "fine-grained" as its supporters claim. RBAC models that use private or parameterized roles have been introduce to tackle these kinds of problems [9, 10, 14]. However, this often leads to a proliferation of roles that undermines the advantages provided by the basic RBAC model (as the number of roles tends towards the number of users). Thus, we believe a new approach is required: an approach that combines the scalability of RBAC with the granularity of the protection matrix model and permits the specification of authorization rules on a per user-resource basis.

Recent research on access control in social networks has used the (social) relationship(s) that exist between users in such networks as the basis for specifying authorization rules [3, 4, 5, 7]. The relationship information available in social networks provides additional context from which access control decisions can be derived. We believe that relationship-based access control could be applied in many other scenarios. In particular, the coarse-grained decision-making in RBAC can be refined using such relationship information.

In this paper, therefore, we develop a novel access control model in which policies are specified in terms of path conditions. To a crude approximation our model takes inspiration from three sources: the overall design of the decision algorithm is similar to Unix; the path conditions are similar in spirit to some of the proposals for relationship-based access control; and the use of implementation-specific authorization principals bears some resemblance to RBAC. We believe our path conditions provide a more rigorous foundation for access control mechanisms than existing proposals for relationship-based access control. We also believe our use of authorization principals provides the desired scalability.

Our model introduces several novel contributions, the most significant being a generic model for access control systems using relationships that is not limited to social networks but can be used to describe access control within more traditional and more diverse environments. Our support for logical entities, as well as the more usual users and resources, allows for a fine grained definition of authorization capable of taking into consideration relevant contextual information encoded in the relationships a request's participants have with other entities. This is balanced with our abstraction of authorization policy to principals rather than subjects, allowing a scalable system which remains powerful and expressive.

In the next section, we describe our model for access control. This section includes the definitions of path conditions, principal-matching rules and authorization policies, and an explanation of how requests are evaluated. In Section 3, we consider the algorithm for matching principals in more detail, presenting a pseudo-code listing, an analysis of the algorithm's complexity and a description of a preliminary implementation in Python. We also describe the results of some simple experiments. We then compare our model to existing, related work and conclude the paper with a summary of our contributions and ideas for future work. The appendix includes an extended example, fragments of which are used throughout the paper. This extended example is used in our experiments.

2. THE AUTHORIZATION MODEL

Informally our model is based on the idea of a labelled graph, in which nodes represent entities within the system and edges represent relationships between entities. Nodes may represent concrete entities, such as users and resources, or logical entities, with which other entities are associated. The relationships' labels are used to define path conditions which can be matched by chains of edges within the graph. A path condition, essentially, identifies a set of authorization principals that is associated with a request. Those principals are authorized to perform actions, thus determining whether a request is authorized or not. Thus our model uses a two-stage decision process: we first identify the principals relevant to the request and then determine whether those principals are authorized.

As we allow entities of various types within our graph, we can make use of a variety of kinds of relationship when processing the authorization decision. If we were to solely include users within our graph, then it could mimic a social network and would be limited to inter-personal relationships for access control policy definition. By including group and resource entity types, we expand the possible subjects and objects, and also the possible relationships which can inform authorization decisions. In Section 2.5, we show that the RBAC model can be seen as an instance of our model. If, however, we include additional entity types and relationships then we can make more fine-grained decisions, as illustrated by the extended example in Appendix A.

2.1 The System Model

Formally, we assume the existence of a set of system entities, which includes the sets of subjects and objects. Each entity has a type and relationships may exist between certain types of entities. Some relationships, such as Sibling-of, are symmetric, while others, such as Brother-of, are not. A system model defines the types along with the entity relationships that are permitted.

DEFINITION 1. *A* system model *comprises a set of types T, a set of relationship labels R, a set of* symmetric *relationship labels $S \subseteq R$ and a* permissible relationship graph *$G_{PR} = (V_{PR}, E_{PR})$, where $V_{PR} = T$ and $E_{PR} \subseteq T \times T \times R$.*

The example in Appendix A defines a number of types, including Group, Project and User, and the relationship type Client-of. Part of the permissible relationship graph includes the edges (Group, Project) and (Group, User), both labelled with the Client-of relationship. Figure 6 (in the appendix) defines the entire permissible relationship graph.

DEFINITION 2. *Given a system model (T, R, S, G_{PR}), a* system instance *is defined by a* system graph *$G = (V, E)$ where V is the set of entities and $E \subseteq V \times V \times R$. We say*

G is well-formed *if for each entity v in V, $\tau(v) \in T$, and for every edge $(v, v', r) \in E$, $(\tau(v), \tau(v'), r) \in E_{PR}$.*

The system model constrains the 'shape' of the system graph by restricting the edges that can be specified. Note that we may have multiple edges between two entities in our system graph (because two or more relationships may exist between vertices). Such a graph is sometimes called a *multigraph*. We will depict an edge (v, v', s), when $s \in S$, without arrowheads, as can be seen in Figure 1 in the case of the Sibling-of relationship. (Due to the symmetry of s, the edge (v, v', s) implies an edge (v', v, s) and vice versa.) An edge (v, v', r), when $r \in R \setminus S$, is directed from v to v', depicted with an arrowhead pointing towards v' (see Figure 1a). The directed edges (v, v', r) and (v', v, r) represent two different relationships. Of course both may belong to E, in which case this will be depicted with arrowheads at both ends of the link between v and v' (see Figure 1b).
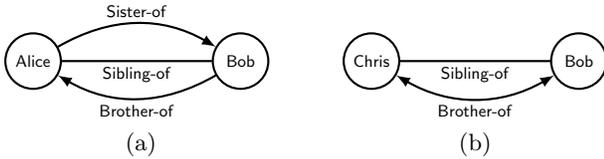

Figure 1: Illustrating different edges in the system graph

Figure 7 (in the appendix) depicts a system graph containing a substantial number of nodes of different types and the relationships that exist between those nodes. Figure 2 shows a simple example of a system graph for illustrative purposes, based on the one in the appendix. Users (such as U_1) are associated with projects (P_1); documents (D_1 and D_2) are grouped together in folders (F_1 and F_2) and allocated to one or more projects, either as part of a group or as a single resource. Relationships include Participant-of, Supervises, Resource-for, and Member-of.
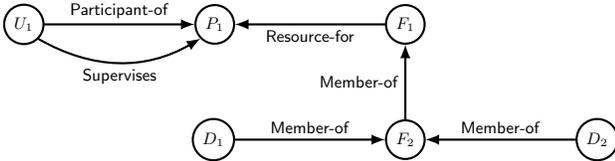

Figure 2: A fragment of a system graph

2.2 Path Conditions

We use path conditions to match requests to principals (described in Section 2.4). In this section, we define the syntax and semantics of path conditions, and establish some basic properties of path conditions, thereby allowing us to restrict our attention to simple path conditions.

DEFINITION 3. *Given a set of relationships R, we define a* path condition *recursively:*

- $\diamond$ *is a path condition;*

- r *is a path condition, for all $r \in R$;*

- *if π and π' are path conditions, then $\pi ; \pi'$, π^+ and $\overline{\pi}$ are path conditions.*

A path condition *of the form r or $\overline{r}$, where $r \in R$, is said to be an* edge condition.

Informally, $\pi ; \pi'$ represents the concatenation of two path conditions; π^+ represents one or more occurrences, in sequence, of π; and $\overline{\pi}$ represents π reversed. We define $\diamond$ for completeness. We note that individual edge conditions could be encoded using attribute-based access control (ABAC) but it is hard to see how ABAC could be easily employed to encode longer chains of relationships.

DEFINITION 4. *Given a system graph $G = (V, E)$ and $u, v \in V$, we write $G, u, v \models \pi$ to denote that G, u and v satisfy path condition π. Formally, for all G, u, v, π, π':*

- $G, u, v \models \diamond$ *iff $v = u$;*

- $G, u, v \models r$ *iff $(u, v, r) \in E$;*

- $G, u, v \models \pi ; \pi'$ *iff there exists $w \in V$ such that $G, u, w \models \pi$ and $G, w, v \models \pi'$;*

- $G, u, v \models \pi^+$ *iff $G, u, v \models \pi$ or $G, u, v \models \pi ; \pi^+$;*

- $G, u, v \models \overline{\pi}$ *iff $G, v, u \models \pi$.*

Note that an edge condition is satisfied by nodes that are adjacent in the system graph. We use $\diamond$ to identify an empty path condition, which is of particular use in our path-matching algorithm in Section 3.1.

In the context of the graph in Figure 2, for example, we have $G, U_1, F_1 \models$ Participant-of ; $\overline{\text{Resource-for}}$ since $G, U_1, P_1 \models$ Participant-of and $G, F_1, P_1 \models$ Resource-for.

DEFINITION 5. *Path conditions π and π' are said to be* equivalent, *denoted $\pi \equiv \pi'$, if, for all system graphs $G = (V, E)$ and all $u, v \in V$ we have*

$$G, u, v \models \pi \quad \text{if and only if} \quad G, u, v \models \pi'.$$

PROPOSITION 1. *For all path conditions π_1 and π_2: (i) $\pi_1 \equiv \pi_1 ; \diamond \equiv \diamond ; \pi_1$ (ii) $\overline{\diamond} \equiv \diamond$ (iii) $\overline{\pi_1 ; \pi_2} \equiv \overline{\pi_2} ; \overline{\pi_1}$ (iv) $\overline{\pi_1^+} \equiv \overline{\pi_1}^+$.*

PROOF. All results follow immediately from Definitions 4 and 5. Consider (iii), for example. By definition, $G, u, v \models \overline{\pi_1 ; \pi_2}$ if and only if $G, v, u \models \pi_1 ; \pi_2$. And $G, v, u \models \pi_1 ; \pi_2$ if and only there exists w such that $G, v, w \models \pi_1$ and $G, w, u \models \pi_2$. Thus we have $G, u, v \models \overline{\pi_1 ; \pi_2}$ if and only if there exists w such that $G, w, v \models \overline{\pi_1}$ and $G, u, w \models \overline{\pi_2}$. That is $G, u, v \models \overline{\pi_2} ; \overline{\pi_1}$. $\square$

DEFINITION 6. *Given a set of relationships R, we define a* simple path condition *recursively:*

- $\diamond$, r *and $\overline{r}$, where $r \in R$, are simple path conditions;*

- *if $\pi \neq \diamond$ and $\pi' \neq \diamond$ are simple path conditions, then $\pi ; \pi'$ and π^+ are simple path conditions.*

In other words, $\overline{\star}$ occurs in a simple path condition if and only if $\star$ is an element of R. It follows from Proposition 1 that every path condition may be reduced to a simple path condition. The path condition $\overline{r_1 ; r_2} ; (r_1 ; r_3)^+$, for example, can be transformed into the equivalent path condition $(\overline{r_3} ; \overline{r_1})^+ ; r_1 ; r_2$ using the equivalences in Proposition 1. Henceforth, we assume all path conditions are simple.

2.3 Policy Specification

Subjects within a system request authorization to perform actions on objects. The policies of a system define the authorized and unauthorized actions and the rules for determining the principals to which these actions are assigned. Principals are mapped to paths within the system graph, where these paths exist between the subject and object of an authorization request. The potential paths are described by path conditions, which are defined using relationships.

DEFINITION 7. *Let P be a set of authorization principals and let R be a set of relationship labels. A principal-matching rule has the form (π, p), where p is an authorization principal and π is either a path condition defined on R or the special symbol $\top$. The path condition π is called a* principal-matching condition. *A principal-matching policy ρ is a list of principal-matching rules.*

Informally, a principal-matching rule (π, p) is applicable to a request (s, o, a) if there is a path from s to o in the system graph that satisfies π.

In order to support scenarios where a default principal should apply, much like the concept of 'world' in the Unix access control system, we allow the definition of a special principal-matching rule with the principal-matching condition set to $\top$. This default principal-matching rule is, if present, always the last rule in the principal-matching policy and (whenever it is evaluated) is applicable to every request. This rule's associated principal, therefore, matches whenever the rule is evaluated.

DEFINITION 8. *An* authorization rule *has the form $(p, \star, a, b)$ or (p, o, a, b), where a is an action, p is a principal, o is an object and $b \in \{0, 1\}$. An* authorization policy *is a list of authorization rules.*

A rule of the form $(p, o, a, 0)$ asserts that p is explicitly *unauthorized* (or *prohibited*) to perform action a on object o, while the rule $(p, o, a, 1)$ explicitly authorizes p. Rules of this form allow us to specify on a per-object basis the actions for which a principal p is (un)authorized. A rule of the form $(p, \star, a, 0)$ asserts that the principal is unauthorized for all objects, while $(p, \star, a, 1)$ asserts that the principal is authorized for all objects. Rules of this form allow us to specify the actions for which a principal is (un)authorized, irrespective of the object to which access is requested. In this case, the authorization policy is concentrated in the principal-matching rule. Note also that we can combine rules $(p, \star, a, 0)$ and $(p, o, a, 1)$, for example, to specify that action a is generally unauthorized for principal p, but is, as an exception, authorized for object o.

Table 2 (in the appendix) lists the principal-matching rules for our example whilst Table 3 lists the authorization rules. A combination of authorization rules has been used in Table 3 to ensure that the Project Resource User is specifically unable to write to Func.Spec.#1 whilst other objects are writable by that principal.

A principal may be explicitly authorized or unauthorized for particular actions. The absence of any explicit authorization rules may itself be considered an implicit authorization depending on the default behaviour of the system. A default access control decision (allow or deny) needs to be specified in the event that no authorization rules apply to a request. Systems may need to support allow-by-default when

the system enters an emergency state, such as the opening of fire exit doors when there is a fire. Other circumstances will commonly require fail-safe handling, where a deny-by-default strategy is implemented in order to ensure no unauthorised access is allowed. Some systems may be deemed so sensitive that there may be no conditions under which allow-by-default would be enabled. In Section 2.4.3 we discuss the specification of default strategies in our model.

2.4 Request Evaluation

Our model for request evaluation is inspired by the Unix access control model and relationship-based access control models and is summarized in Figure 3. From the Unix model, we take the idea of binding a request to a principal before computing an access control decision, which we combine with the idea of specifying authorization policies in terms of relationships. Firstly, we use the subject and object specified in the request to compute a set of applicable principals. Then we compute the actions for which those principals are authorized. Finally a decision is made to allow or deny the request based on those authorizations.
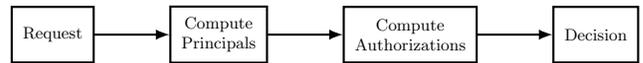

Figure 3: Processing overview

We now describe request evaluation, which has two main stages and is depicted schematically in Figure 4, in more detail. The first stage determines a list of matched principals for the request: in Figure 4 this stage is represented by the horizontal row of steps from 'START'. The second stage determines the authorizations explicitly defined for those matched principals identified in the first stage. This second stage is represented in Figure 4 by the vertical column of steps beginning at the 'MP list empty?' decision point. A conflict resolution process is employed to resolve any conflicting authorization rules and from this a decision is made.

2.4.1 Principal Matching

The list of matched principals is determined by the evaluation of principal-matching rules within the principal-matching policy. Thus, we first specify what it means for a principal-matching rule to be matched.

DEFINITION 9. *Let $q = (s, o, a)$ be a request and $G = (V, E)$ be a system graph. Then request q matches principal-matching rule (π, p) if $G, s, o \models \pi$. Given a principal-matching policy and a system graph, we write $G, q \xrightarrow{\pi} p$ if there exists a principal-matching rule (π, p) and request q matches (π, p).*

Informally, a principal-matching rule maps a (complex) relationship between entities in a graph to a principal; in other words, a principal-matching rule enables us, conceptually, to replace a path between two entities with a single edge labelled by a principal. Figure 5 illustrates such a matching, where request $q = (s, o, a)$ matches a principal-matching rule $(r_1; \overline{r_2}; r_3; r_4, p)$. It is worth noting that, based on the relationships shown in Figure 5, where r_4 is a symmetric label (identified by the lack of arrows on the edge), the principal-matching rule would also have been matched if the path condition had been $r_1; \overline{r_2}; r_3; \overline{r_4}$.
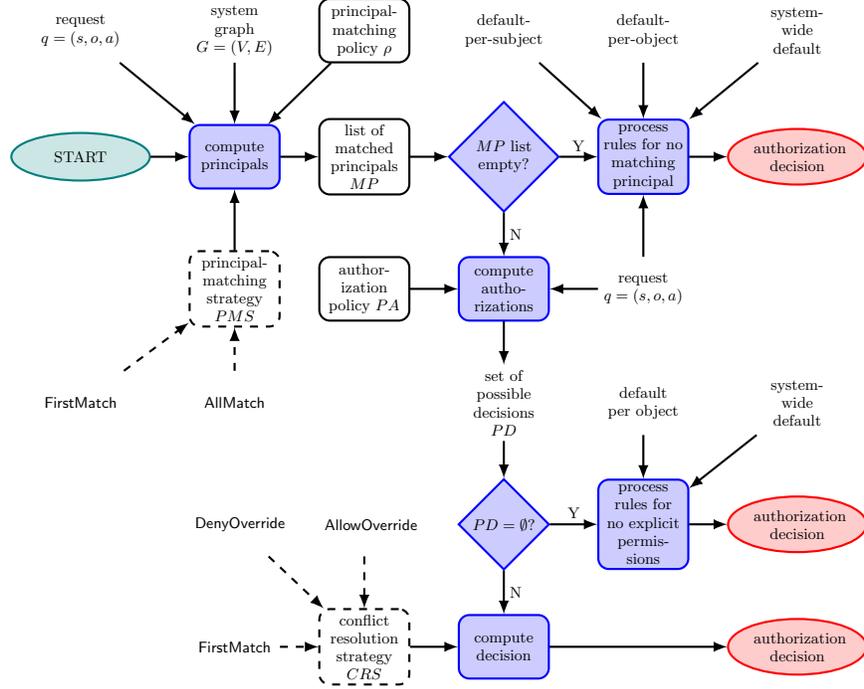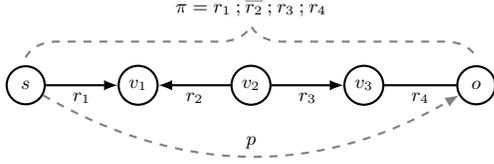
Figure 4: Detailed architecture

Figure 5: Principal-matching rule

A request may match more than one rule in the principal-matching policy. A *principal-matching strategy* (PMS) defines how the principals in matched rules should be combined (if at all). We consider two very natural PMSs: FirstMatch and AllMatch, but other options may be appropriate in some circumstances. The former evaluates the list of principal-matching rules in order and terminates when a path condition is matched, returning the corresponding principal. The latter evaluates the entire list of rules in the policy and returns a list of the principals in rules for which the request matches the path condition.

If used in conjunction with the FirstMatch PMS, the default principal rule $(\top, p)$, when present, would only be triggered, and so only apply, if no other rule matches. When used with AllMatch this rule would always apply, resulting in the default principal always being added to the list of matched principals.

An *authorization system* comprises a principal-matching policy ρ, a principal-matching strategy PMS, an authorization policy PA, and a conflict resolution strategy CRS (described in the next section). Given an authorization system, a system graph G and a request q, the list of *matched principals MP* includes those principals resulting from successful matches made in accordance with the specified principal-

matching strategy. We write $G, q \xrightarrow{\rho} MP$ to indicate that the list of matched principals for q (with respect to policy ρ and system graph G) is MP. If MP is empty then an authorization decision must be made based on pre-defined defaults. This process is described in Section 2.4.3.

2.4.2 Computing Authorizations and Decisions

The second stage of request authorization identifies whether the requested action (on the object) is explicitly authorized or unauthorized for one or more of the matched principals. Subsequently any conflicting assignments are resolved and we determine whether the requested action should, therefore, be allowed or denied.

DEFINITION 10. *Given a policy ρ, a request $q = (s, o, a)$ and a system graph G such that $G, q \xrightarrow{\rho} MP$, we define the set of* possible decisions, *denoted PD, to be* $\{b \in \{0, 1\} : (p, o, a, b) \in PA, p \in MP\}$.

PD can take one of four values: $\{0\}$, $\{1\}$, $\{0, 1\}$ and $\emptyset$.

- If $PD = \{b\}$, $b \in \{0, 1\}$, a decision can unambiguously be made (as a deny or allow, respectively).

- If $PD = \{0, 1\}$, we must employ a *conflict resolution strategy* (CRS) to determine the decision. We define three conflict resolution strategies: FirstMatch, DenyOverride and AllowOverride. The use of one of these strategies allows a single decision to be made from the conflicting assignments. In order to support the first of these, we require that the set of possible decisions be determined by considering each authorization, from the list PA, in turn.

The FirstMatch CRS takes the first element to be added to PD as the decision. In this way, if a positive authorization is identified first, then the request is

allowed. If a negative authorization is identified first, however, then the request is denied.

The DenyOverride and AllowOverride CRSs allow their respective elements, 0 and 1, to take precedence over the alternative, no matter which is identified first.

- The final case is $PD = \emptyset$. In this case, the authorization decision must once again be made using predefined defaults, as explained in Section 2.4.3 below.

2.4.3 Defaults

There are two circumstances when default decision making applies. The first is when no matched principals are identified, whilst the second is, as just described, when the set of possible decisions is empty.

To accommodate varying needs in these circumstances, we allow for default allow or deny of a request to be determined at one of the following levels: default-per-subject, default-per-object or system-wide default. We only support the default-per-subject when there are no matched principals, and not later, when there are no explicit authorizations. At the time when the set of possible decisions is determined, the subject is no longer directly relevant, having already been used to identify the appropriate matched principals. It is therefore unnecessary to reconsider the subject in order to evaluate the authorization decision.

The three defaults are evaluated in order, where specified, with the first applicable default determining the authorization decision. In this way, if there is a default specified for the subject s of the request $q = (s, o, a)$, the subject's default (allow or deny) applies. If no subject default is defined for s, then the default for the object o of the request shall apply, if specified. If there is no subject default for s and no object default for o, then the system-wide default shall apply. Whilst defaults for the subject and object are optional and may not be specified for the entities involved in the request, a system-wide default must be specified so as to ensure authorization decisions can be made in all circumstances.

2.5 Special Cases

The Unix access control mechanism employs a similar, albeit far simpler, mapping technique as that used above to identify principals from path conditions [6]. It can, therefore, be trivially represented using our model. In particular, the system model contains a set of three types: users, groups and objects, and a set of three relationships (none of which are symmetric): User-object, User-group and Group-object, which we will label uo, ug and go, respectively. The permissible relationship graph links the users to the objects and to the groups, as well as linking the groups to the objects (this is as our relationship naming suggests). There are three principal-matching rules: (uo, owner), (ug ; go, group) and, the default, ($\top$, world). Finally, we use the FirstMatch PMS and evaluate the rules in the above order.

Note also that we can configure our model to implement core RBAC [2]. We assume the set of entities is the disjoint union of users, roles, permissions and objects. Then there are two types of relationship, the User-role relationship, referred to as user assignment and abbreviated ua, along with the Role-permission relationship, referred to as permission assignment and abbreviated pa. At its simplest we then define a principal-matching policy where each rule has the form

(ua ; pa, p) where the principal p has the same name as the permission identified by the pa edge. The authorization policy contains elements $(p, ob, op, 1)$ which map the principals to objects, allowing them operations (as per the permission binary relation in RBAC).

Additionally, we can introduce the Role-role relationship (abbreviated rr) in order to extend this configuration to implement a role hierarchy. Finally, we could also introduce the User-permission relationship (abbreviated up), in order to articulate exceptions to the basic RBAC model by directly associating permissions with users.

Our model does not directly support the concept of sessions. However, if we were to introduce support for changing the system graph, we could employ a User-session-role relationship. The User-session-role relationship may only connect users and roles who are already joined by a User-role relationship. We then modify the original principal-matching rules to have the form (usr ; pa, p). Supporting (constrained) updates to the system graph in real time will be an important aspect of our future work.

3. PATH MATCHING

Principal matching, the first stage of request evaluation, described in Section 2.4.1, is the most complex part of request evaluation. (The second stage amounts to a sequence of simple lookups and comparisons.) Principal matching requires us to determine whether there exists a path in the graph from subject to object that matches a path condition. In this section, we describe the MatchPrincipal algorithm, which takes a path condition, two nodes (the subject and object of a request), the set of symmetric relationship labels and a system graph as inputs and returns a Boolean value indicating whether there exists a matching path in the graph.

The algorithm uses a (modified) breadth-first search to determine whether there exists a path in the system graph that begins at the subject and ends at the object such that concatenation of the relationship labels is equal to the path condition. It is employed iteratively to as many rules in the principal-matching policy as required, given the PMS in use: if FirstMatch is used then the algorithm is run on each principal-matching rule in turn, until a match is found; if the AllMatch PMS is used, the algorithm is run for every rule in the policy. In order to determine satisfaction of a simple path condition, we attempt to satisfy its component edge conditions one at a time. It is helpful to define the head and suffix of a path condition: the head is used to match edge labels in the graph, while the suffix determines the residual path condition.

DEFINITION 11. *Let $\pi \neq \diamond$ be a simple path condition. Then we define the* head *and* suffix *of π, denoted $\mathsf{H}(\pi)$ and $\mathsf{S}(\pi)$, respectively, as follows:*

- $\mathsf{H}(r) = r$ *and* $\mathsf{S}(r) = \diamond$;
- $\mathsf{H}(\bar{r}) = \bar{r}$ *and* $\mathsf{S}(\bar{r}) = \diamond$;
- $\mathsf{H}(\pi_1 \,;\, \pi_2) = \mathsf{H}(\pi_1)$ *and* $\mathsf{S}(\pi_1 \,;\, \pi_2) = \mathsf{S}(\pi_1) \,;\, \pi_2$;
- $\mathsf{H}(\pi^+) = \mathsf{H}(\pi)$ *and* $\mathsf{S}(\pi^+) = \mathsf{S}(\pi); \pi^*$, *where π^* denotes 0 or more occurrences of π.*

PROPOSITION 2. *Let π be a simple path condition. Then $\mathsf{H}(\pi)$ is equal to r or $\bar{r}$ for some $r \in R$. Moreover, $\mathsf{S}(\pi)$ is a simple path condition.*

PROOF. The results follow immediately by a simple induction on the structure of simple path conditions. □

PROPOSITION 3. *Let π be a simple path condition. Then $\pi \equiv \mathsf{H}(\pi) \mathbin{;} \mathsf{S}(\pi)$.*

PROOF. The proof proceeds by induction on the structure of π. Consider the (base) case $\pi = r$. Then

$$G, u, v \models \mathsf{H}(r) \mathbin{;} \mathsf{S}(r) \Leftrightarrow G, u, v \models r \mathbin{;} \diamond$$
$$\Leftrightarrow G, u, v \models r$$

Thus $\mathsf{H}(r) \mathbin{;} \mathsf{S}(r) \equiv r$, as required. We prove the case $\pi = \bar{r}$ in a similar fashion. Now consider $\pi = \pi_1 \mathbin{;} \pi_2$ and assume the result holds for π_1 and π_2. Then

$$G, u, v \models \mathsf{H}(\pi_1 \mathbin{;} \pi_2) \mathbin{;} \mathsf{S}(\pi_1 \mathbin{;} \pi_2) \Leftrightarrow G, u, v \models \mathsf{H}(\pi_1) \mathbin{;} \mathsf{S}(\pi_1) \mathbin{;} \pi_2$$
$$\Leftrightarrow G, u, v \models \pi_1 \mathbin{;} \pi_2$$

Finally, consider π^+ and assume the result holds for π. Then

$$G, u, v \models \mathsf{H}(\pi^+) \mathbin{;} \mathsf{S}(\pi^+) \Leftrightarrow G, u, v \models \mathsf{H}(\pi) \mathbin{;} \mathsf{S}(\pi) \mathbin{;} \pi^*$$
$$\Leftrightarrow G, u, v \models \pi \mathbin{;} \pi^*$$
$$\Leftrightarrow G, u, v \models \pi^+$$

concluding the proof. □

We now develop the path-matching algorithm in more detail.

3.1 The Path-Matching Algorithm

The algorithm takes a start node (the subject), a target node (the object) and a path condition as part of its input. The current node is initialized to be the start node. The path-matching algorithm traverses the provided system graph 'consuming' the head of the path condition as it matches it against (one or more of) the relationship labels associated with incident edges of the current node. It then considers each of the adjacent edges in turn replacing the path condition with the relevant suffix. The algorithm terminates if it 'consumes' the entire path condition with the adjacent node equal to the target node or if no further matches can be made.

If we consider, for example, the graph in Figure 5, the request (s, o, a) and path condition $r_1 \mathbin{;} r_2$, then $\mathsf{H}(r_1 \mathbin{;} r_2) = r_1$, which is the label on edge (s, v_1, r_1). Hence, the edge is traversed and we next consider the node v_1 with path condition $\mathsf{S}(r_1 \mathbin{;} r_2) = r_2$. The algorithm terminates at this point (returning false) because there is no outgoing edge from v_1 labelled r_2.

The MatchPrincipal algorithm (listed in Algorithm 1) is, essentially, a modified breadth-first search algorithm. However, there are some awkward aspects to the design of the algorithm. First, we have to allow for nodes to be revisited. Second, we have to allow matching of edge conditions of the form r and $\bar{r}$. Finally, our algorithm has to be able to handle path conditions of the form π^+ without entering an endless loop, in order for the algorithm to terminate.

The algorithm uses a queue Q to track nodes that we have to visit. Unlike a conventional breadth-first search, we allow those nodes to be revisited because path conditions may be satisfied by a cycle in the system graph. However, if we revisit a node then we require a different non-empty path condition on each visit. In this way we avoid infinite loops whilst processing the path condition.

Algorithm 1 MatchPrincipal

Require: Graph $G = (V, E)$, set of symmetric relationship labels S, nodes u and v, and path condition π
Ensure: Returns true if $G, u, v \models \pi$ and false if it does not
1: Initialize empty queue Q
2: Initialize empty set of visited nodes $SEEN$
3: **add** (u, π) **to** Q
4: $SEEN = SEEN \cup \{(u, \pi)\}$
5: **while** Q is not empty **do**
6:     **dequeue** next entry (h, ϕ) from Q
7:     Initialize empty list of (node, suffix) tuples Θ
8:     // consider edges directed away from h
9:     **for** each edge $(h, w, r) \in E$ **do**
10:         **if** $\phi = \pi_1^* \mathbin{;} \pi_2$ **then**
11:             $\Theta = \Theta \sqcup [(h, \pi_2)]$
12:             $\phi = \pi_1^+ \mathbin{;} \pi_2$
13:         **end if**
14:         **if** $\mathsf{H}(\phi) = r$ **then**
15:             $\Theta = \Theta \sqcup [(w, \mathsf{S}(\phi))]$
16:         **end if**
17:         **if** $(r \in S$ **and** $(w, h, r) \notin E)$ **then**
18:             **if** $\mathsf{H}(\phi) = \bar{r}$ **then**
19:                 $\Theta = \Theta \sqcup [(w, \mathsf{S}(\phi))]$
20:             **end if**
21:         **end if**
22:     **end for**
23:     // consider edges directed towards h
24:     **for** each edge $(w, h, r) \in E$ **do**
25:         **if** $\phi = \pi_1^* \mathbin{;} \pi_2$ **then**
26:             $\Theta = \Theta \sqcup [(h, \pi_2)]$
27:             $\phi = \pi_1^+ \mathbin{;} \pi_2$
28:         **end if**
29:         **if** $\mathsf{H}(\phi) = \bar{r}$ **then**
30:             $\Theta = \Theta \sqcup [(w, \mathsf{S}(\phi))]$
31:         **end if**
32:         **if** $(r \in S$ **and** $(h, w, r) \notin E)$ **then**
33:             **if** $\mathsf{H}(\phi) = r$ **then**
34:                 $\Theta = \Theta \sqcup [(w, \mathsf{S}(\phi))]$
35:             **end if**
36:         **end if**
37:     **end for**
38:     // determine match or other nodes to visit
39:     **for** each $(n, \phi_s) \in \Theta$ **do**
40:         **if** $(n, \phi_s) \notin SEEN$ **then**
41:             **if** $\phi_s = \diamond$ **then**
42:                 **if** $n = v$ **then**
43:                     **return true** // match
44:                 **end if**
45:             **else**
46:                 **add** (n, ϕ_s) **to** Q
47:                 $SEEN = SEEN \cup \{(n, \phi_s)\}$
48:             **end if**
49:         **end if**
50:     **end for**
51: **end while**
52: **return false** // no match

Previously visited nodes, and the path condition at the time of the visit, are tracked using the set $SEEN$. At each node h we visit, we identify incident edges from our system graph $G = (V, E)$, it is these edges that we attempt

to traverse by matching a label to the head of our path condition. Matched edges result in path condition suffixes relevant at specific adjacent nodes; we hold these in a list as node-suffix pairs. We use the notation $list_1 \sqcup list_2$ to indicate the concatenation of two lists, where the entries from $list_2$ are appended, in order, to the end of $list_1$.

The algorithm performs its edge condition matching in lines 8 to 22 and 23 to 37 for outgoing and incoming edges to the current node h respectively. However, the implementation is complicated by the handling of path conditions of the form π^+, whose suffix includes π^* (see Proposition 3), which may represent 0 occurrences of π or at least one occurrence of π. When processing path conditions, therefore, we first determine if it has the structure $\pi_1^* ; \pi_2$ (where π_2 may be $\diamond$) and, if so, we treat it as $\pi_1^+ ; \pi_2$; in addition we add π_2 (corresponding to 0 occurrences of π_1), along with the current node h, to our list of node-suffix pairs for consideration later (see Algorithm 1 lines 10 to 13 and 25 to 28).

After an edge condition is matched by the algorithm, each node-suffix pair is checked against *SEEN* and ignored if previously processed. The suffix ϕ_s of each unseen tuple is compared to $\diamond$, those matching indicate fully processed path conditions. If the node n associated with such a tuple is equal to the target node v then the path condition is considered to have been matched between u and v and the algorithm returns true. If the node isn't the target node, then the tuple is discarded as there is no remaining path condition to evaluate. Those unseen tuples, whose suffixes are not $\diamond$, are added to the queue of nodes to be visited (see Algorithm 1 lines 38 to 50).

Once all incident edges are considered for the current node, we move to the next node as indicated by the next entry in the queue Q. If the queue is empty and we have not already returned a value, then the path condition cannot be matched (because there are no further nodes to examine) and the algorithm returns false (see Algorithm 1 lines 5, 6 and 52).

Consider, for example, the system graph depicted in Figure 5 and the path condition $r_1^+ ; \overline{r_2} ; r_3 ; r_4$ with start node s and end node o. Then we are able to match edge condition r_1 and progress to node v_1 with path condition $r_1^* ; \overline{r_2} ; r_3 ; r_4$. We now attempt to match r_1 again, which fails. In addition, we add $(v_1, \overline{r_2} ; r_3 ; r_4)$ to the list of node-path condition pairs to consider. This will, eventually, lead to the node-suffix pair $(o, \diamond)$ being identified, at which point the algorithm will return a match (for path condition $r_1^+ ; \overline{r_2} ; r_3 ; r_4$ with start and end nodes s and o).

3.2 Correctness and Complexity

We first introduce the concept of the length of a simple path condition. Informally, it is equal to the number of edge conditions (r or $\overline{r}$) which it contains.

DEFINITION 12. *The* length $\ell(\pi)$ *of simple path condition* π *is defined as follows:*

- $\ell(r) = \ell(\overline{r}) = 1$;
- $\ell(\pi ; \pi') = \ell(\pi) + \ell(\pi')$;
- $\ell(\pi^+) = \ell(\pi)$.

The length $\ell(\rho)$ *of a principal-matching policy* ρ *is equal to the length of the longest path condition of the principal-matching rules within* ρ, $\ell(\rho) = \max_{\pi \in \rho} (\ell(\pi))$.

Our algorithm terminates because, with one exception discussed below, $\ell(\mathsf{S}(\pi)) = \ell(\pi) - 1$ because an edge is consumed in matching the head of the path condition. Thus, any node-suffix pair that is enqueued contains a shorter path condition. Eventually, the path condition will be reduced to $\diamond$ and we test whether the adjacent node is the target node. The exception arises when we consider a path condition of the form $\pi^* ; \pi'$. In this case, we enqueue a path condition of the form π' and also evaluate the path condition $\pi^+ ; \pi'$. Thus, we could, in the worst case visit every node in the system graph and evaluate the path condition $\pi^+ ; \pi'$. However, we do not enqueue a node-suffix pair if we have previously evaluated it (in the same way that a normal breadth-first search keeps track of visited nodes). Thus, for this exceptional case, we will eventually process the path condition π' (since $\pi^+ ; \pi'$ will either be discarded or fail to find a matching edge).

We can summarise our path-matching algorithm's processing as a breadth-first search through the graph, attempting to match edges to the remaining path condition. At each node the number of possible comparisons depends on the degree of that node. The path condition under consideration is re-written as each edge comparison is performed, with the head of the path condition removed if the edge satisfied the next element in the path condition. The path condition under consideration at adjacent nodes is, therefore, one element shorter than at the current one.

The time complexity of a standard breadth-first search is determined by the number of nodes and edges, since, in the worst case, each node and edge will be explored. For the PrincipalMatch algorithm, the number of "nodes" is determined by the number of nodes in the system graph and the length of the path condition. Specifically, the size of the queue is bounded by $|V| \cdot \ell(\phi)$. The number of edges in the system graph is $O(|V|^2 \cdot |R|)$. Thus, the total complexity of the algorithm is $O(|V| \cdot \ell(\phi) + |V|^2 \cdot |R|)$.

The MatchPrincipal algorithm determines whether a single path condition matches. In order to compute the list of matching principals in the worst case, every rule in the principal-matching policy ρ may need to be evaluated. The worst-case time complexity of principal matching is, therefore, determined by the complexity of matching one rule, the number of rules in the policy and $\ell(\rho)$.

3.3 Implementation

We have created a Python implementation of the MatchPrincipal algorithm which roughly follows the structure shown in Algorithm 1. We represent a path condition as a tree of nodes, where each node is a data structure containing (i) pointers to a left and a right node (ii) a relationship label if it is a leaf node (iii) a node type if the node is a non-leaf node (indicating the operation used to construct the path condition). Our implementation modifies the pseudo-code listed in Algorithm 1 in order to improve the processing of path conditions containing π^*. In particular, we process both possibilities for π^* when we meet it, rather than simply putting one aside for consideration later (as we do in lines 11 and 26 of Algorithm 1).

Using this implementation we evaluated the requests in our appendix example. The results are summarized in Table 1, which shows the number of nodes visited (n) and edges considered (e) during the evaluation of one specific principal-matching rule for each of these requests.

Path condition π	$\ell(\pi)$	Request	n	e	Found
$\mathsf{P}\,;\overline{\mathsf{R}}\,;\overline{\mathsf{M}}^{+}$	3	(Sales.#2, Func.Spec.#1, write)	5	19	Yes
$\mathsf{P}\,;\overline{\mathsf{R}}\,;\overline{\mathsf{M}}^{+}$	3	(Tech.#2, Test.Spec.#1, read)	7	24	Yes
$\mathsf{S}\,;\overline{\mathsf{R}}\,;\overline{\mathsf{M}}^{+}$	3	(Tech.#2, Func.Spec.#1, write)	4	15	Yes
$\mathsf{S}^{+}\,;\overline{\mathsf{M}}\,;\mathsf{S}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}$	5	(CTO, Proj.#1 Report#1, read)	17	58	Yes
$\mathsf{S}^{+}\,;\overline{\mathsf{M}}\,;\mathsf{S}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}$	5	(CEO, Proj.#1 Report#1, read)	7	24	No

Table 1: Running our implementation of MatchPrincipal using path conditions and requests from Tables 2 and 4

Notice that the algorithm may visit many more nodes than exist on the shortest path between the subject and object of the request. This is because we are using a breadth-first search. Notice also that two different subject nodes may be the same distance from the object node (as is the case for the subjects in the first and second rows) and yet one request is resolved with less computational effort. It would be interesting to see whether there is any advantage to be gained in using a depth-first search. This is certainly something we hope to investigate in future work.

4. RELATED WORK

We have already noted those aspects of the Unix access control model and role-based access control that have influenced the design of our model. Our work also takes inspiration from the formal model developed for Unix by Crampton [6], which suggested that the two-stage evaluation process used by Unix could provide inspiration for novel relationship-based access control models. We now compare our model with related work in the literature.

The widespread use of social networks and restricting the access to resources within such networks has inspired the development of research into relationship-based access control. The early work of Kruk *et al.* used friend and friend-of-a-friend relationships to determine access to resources [12], while the work of Ali *et al.* was based on the trust relationships between users [1]. Carminati *et al.* synthesized these elements to create an access control model for social networks based on relationships [3]. They represent a social network as a graph in which the edges are labelled by relationships (such as friend) and all nodes represent users. Each edge is also labelled with a trust value, indicating the "strength" of the relationship. An access condition has the form (u, r, d, t), where u is a user, r is a relationship label, d is the depth and t is the trust threshold. An access rule has the form (o, C), where o is an object and C is a set of access conditions. A user v is authorized to access the resource o if v satisfies the access conditions specified in C. More recent work has built on this model to provide additional features, such as joint management of access policies, but only in the context of social networks [11].

If we ignore the trust threshold, access conditions are a special case of path conditions. Specifically a relationship r of depth d can be represented by the path condition $r\,;\ldots;r$ (repeated d times). Moreover, we can specify relationships of unbounded depth using the path condition r^{+}. However, access conditions certainly cannot represent arbitrary path conditions. In other words, our approach significantly extends the possibilities for policy specification. (Trust thresholds may be useful in social networks, but we feel their use for our intended applications is inappropriate. Of course,

our framework may be easily adapted to accommodate trust thresholds by having a path condition built from pairs of the form (r, t), where r is a relationship label and t is a threshold.)

Fong's recent work on relationship-based access control also concentrates on access control in social networks and models the social network as a graph in which the edges are labelled by relationships and all nodes are users [7, 8]. Fong's work specifies a policy for each resource, where a policy is specified using a multi-modal logic.[1] Thus Fong's work provides a richer policy language than that of Carminati *et al.* The policy syntax is specified by the grammar

$$\phi, \psi ::= \top \mid \mathsf{a} \mid \neg\phi \mid \phi \vee \psi \mid \langle i \rangle \phi$$

where i is a relationship identifier. Informally, $\top$ serves the same purpose as our default path condition $\top$; a is analogous to $\diamond$; $\langle i \rangle$ is equivalent to our path condition r. Fong's language can encode alternatives (using $\vee$); we would simply specify alternative principal-matching rules. Fong's language does support negation, which we do not. Conversely, our language does support unbounded path conditions, which are useful when traversing a sub-graph comprising similar types of elements that might have arbitrary diameter (as in a directory tree, for example). A limitation of Fong's language is that a policy has to be specified for every resource and admits no relationships, other than ownership, between users and resources. In our approach, we simply identify the principals that apply to a request, given the subject and object of the request, thereby leveraging some of the advantages of a role-based approach. Moreover, we allow for arbitrary relationships between the nodes (subject to the constraints in the permissible relationship graph) in the system graph.

Cheng *et al.* also focused on the use of relationship-based access control within Online Social Networks [4, 5]. Their work allows for the specification of user-to-resource relationships (other than ownership). However, our model is more general still in its support for entities of any kind (including logical ones) and policies not focused on, but still applicable to, social networks. Cheng *et al.* employ a path checking algorithm which is comparable to our concept of path matching. However, their approach directly assigns permissions, whereas we introduce some of the benefits of RBAC and Unix access control by abstracting that assignment to matched principals. Their path expressions are directly based on regular expressions, including wildcards,

---

[1]The rationale for using a modal logic is that each relationship specifies an accessibility relation between users, which is used to provide semantics for policies. We do this more directly by working with path conditions and specifying their semantics in terms of a graph.

although they constrain rules containing wildcards so that such rules could, in fact, be enumerated as different alternatives. (Thus paths of arbitrary length are not properly supported.) In contrast, we only provide direct support for π^+ in path conditions, but do not limit the number of edges across which it can match, something that is crucial when dealing with variable depth data structures such as directory trees. Moreover, as we have seen, we can encode alternation in a rule's path condition as two (or more rules): the rule $(\pi_1 \mid \pi_2, p)$ is simply defined as two principal-matching rules (π_1, p) and (π_2, p). Similarly, we can handle (π^*, p) by defining the rules (π^+, p) and $(\diamond, p)$.

5. CONCLUSION

We have formally defined a new graph-based model for access control based on two concepts: path conditions and principal matching. We believe that path conditions are a novel contribution to the literature on relationship-based access control and that these conditions allow us to specify a wide range of policies that are relevant to access control in a wide range of applications, not just in the usual context of social networks. Principal matching enables us to leverage the advantages of both Unix and RBAC and extend the capabilities of both models. We also believe our model provides significant advantages over existing models for relationship-based access control, both in terms of the expressive power of path conditions and the relatively straightforward request evaluation process. Additionally, our model is generic, thus able to describe systems of various forms be they social networks, IT systems (singularly or as networks) or entire businesses. We have illustrated how the model can be implemented by describing an algorithm to support principal matching and, thereby, enable request evaluation within our model.

There are many opportunities for further work. In particular, we would like to investigate alternative path-matching algorithms and compare their efficiency with the one described in Section 3. SPARQL is an RDF query language that may well be an suitable alternative. We would also like to extend the policy language to include more expressive matching as a means of directly supporting access constraints such as separation of duty, binding of duty and Chinese Wall. We believe that such constraints can be supported simply by introducing conjunction within, and negation of, path conditions. Extending this further we plan to consider the matching of subgraphs, rather than paths, and to investigate the trade-offs in increased expressive power with the more expensive request evaluation algorithms that will be required. We also intend to develop an administrative model to manage components such as the system graph. In this way, we should be able to handle dynamic concepts, such as sessions in RBAC. RT is a family of role-based trust management languages [13] that combine features of RBAC with distributed access control models. Many of the rules of RT can, like the assignment relations in RBAC, be encoded as a single type of relationship within a system graph in our model. However, the RT delegation rule $A.r \leftarrow A'.r'.r''$ cannot be directly encoded within our model. We would like to be able to provide support for distributed access control, in which different parts of the subgraph form different administrative domains. Then RT-like rules would specify the edges that link different subgraphs. Finally, we would also like to enrich the model with stateful objects, such as work-

flow tasks, for which the set of authorized individuals may change over time. We expect that this will result in the system graph being updated as the state of an object changes (for example to support task-based separation of duty).

6. REFERENCES

[1] ALI, B., VILLEGAS, W., AND MAHESWARAN, M. A trust based approach for protecting user data in social networks. In *CASCON* (2007), K. A. Lyons and C. Couturier, Eds., IBM, pp. 288–293.

[2] ANSI. *American National Standard for Information Technology - Role Based Access Control (359-2004)*. ANSI INCITIS, 2004.

[3] CARMINATI, B., FERRARI, E., AND PEREGO, A. Enforcing access control in web-based social networks. *ACM Trans. Inf. Syst. Secur. 13*, 1 (2009).

[4] CHENG, Y., PARK, J., AND SANDHU, R. S. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *SocialCom/PASSAT* (2012), IEEE, pp. 646–655.

[5] CHENG, Y., PARK, J., AND SANDHU, R. S. A user-to-user relationship-based access control model for online social networks. In *DBSec* (2012), N. Cuppens-Boulahia, F. Cuppens, and J. García-Alfaro, Eds., vol. 7371 of *Lecture Notes in Computer Science*, Springer, pp. 8–24.

[6] CRAMPTON, J. Why we should take a second look at access control in unix. In *13th Nordic Conference on Secure IT Systems* (2008), NORDSEC'08, ACM.

[7] FONG, P. W. L. Relationship-based access control: protection model and policy language. In *CODASPY* (2011), R. S. Sandhu and E. Bertino, Eds., ACM, pp. 191–202.

[8] FONG, P. W. L., ANWAR, M. M., AND ZHAO, Z. A privacy preservation model for facebook-style social network systems. In *ESORICS* (2009), M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 303–320.

[9] GE, M., AND OSBORN, S. L. A design for parameterized roles. In *DBSec* (2004), C. Farkas and P. Samarati, Eds., Kluwer, pp. 251–264.

[10] GIURI, L., AND IGLIO, P. Role templates for content-based access control. In *ACM Workshop on Role-Based Access Control* (1997), pp. 153–159.

[11] HU, H., AHN, G.-J., AND JORGENSEN, J. Multiparty access control for online social networks: Model and mechanisms. *IEEE Trans. Knowl. Data Eng. 25*, 7 (2013), 1614–1627.

[12] KRUK, S. R., GRZONKOWSKI, S., GZELLA, A., WORONIECKI, T., AND CHOI, H.-C. D-FOAF: Distributed identity management with access rights delegation. In *ASWC* (2006), R. Mizoguchi, Z. Shi, and F. Giunchiglia, Eds., vol. 4185 of *Lecture Notes in Computer Science*, Springer, pp. 140–154.

[13] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy* (2002), IEEE Computer Society, pp. 114–130.

[14] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *IEEE Computer 29*, 2 (1996), 38–47.

APPENDIX

A. CORPORATE EXAMPLE

The following example applies our model to the project environment within a fictional company. To support this specific system, we initially define the underlying system model $(T, R, S, G_{\mathrm{PR}})$ where the set of types is

$$T = \{\mathsf{File}, \mathsf{Folder}, \mathsf{Group}, \mathsf{Printer}, \mathsf{Project}, \mathsf{User}\}$$

The set of relationship labels used in the system model is

$$R = \{\mathbf{C}\mathsf{lient\text{-}of}, \mathbf{D}\mathsf{eliverable\text{-}for}, \mathbf{M}\mathsf{ember\text{-}of},$$
$$\mathbf{P}\mathsf{articipant\text{-}of}, \mathbf{R}\mathsf{esource\text{-}for}, \mathbf{S}\mathsf{upervises}\}$$

There are no symmetric relationship labels. Finally, the permissible relationship graph G_{PR}, defined using T and R, is shown in Figure 6.
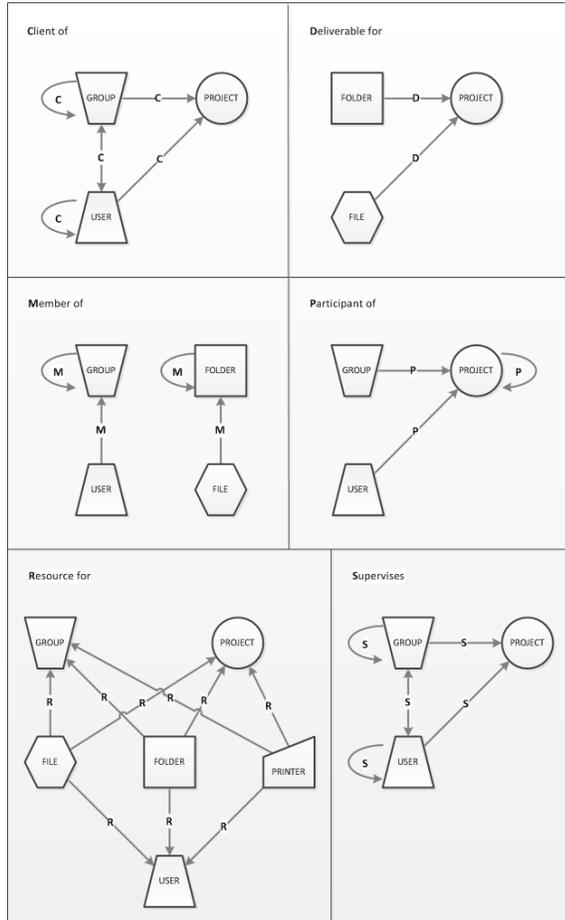

Figure 6: Permissible relationship graph

Using this system model we then describe the project environment using the system graph shown in Figure 7.

Within our authorization system we define the principal-matching rules shown in Table 2 and make use of the All-Match PMS.

Additionally, we define the authorization policy shown in Table 3 and whilst we define no per-subject or per-object defaults, we define the system-wide default as deny-by-default. We employ the FirstMatch conflict resolution strategy.

#	Principal-Matching Rule
1	$(\mathsf{C}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}, \text{Deliverable Client})$
2	$(\mathsf{S}^{+}\,;\overline{\mathsf{M}}\,;\mathsf{S}\,;\overline{\mathsf{D}}, \text{Deliverable Reviewer})$
3	$(\mathsf{S}^{+}\,;\overline{\mathsf{M}}\,;\mathsf{S}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}, \text{Deliverable Reviewer})$
4	$(\mathsf{S}\,;\overline{\mathsf{D}}, \text{Deliverable Supervisor})$
5	$(\mathsf{S}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}, \text{Deliverable Supervisor})$
6	$(\mathsf{P}\,;\overline{\mathsf{D}}, \text{Deliverable User})$
7	$(\mathsf{P}\,;\overline{\mathsf{D}}\,;\overline{\mathsf{M}}^{+}, \text{Deliverable User})$
8	$(\mathsf{S}\,;\overline{\mathsf{R}}, \text{Project Resource Supervisor})$
9	$(\mathsf{S}\,;\overline{\mathsf{R}}\,;\overline{\mathsf{M}}^{+}, \text{Project Resource Supervisor})$
10	$(\mathsf{P}\,;\overline{\mathsf{R}}, \text{Project Resource User})$
11	$(\mathsf{P}\,;\overline{\mathsf{R}}\,;\overline{\mathsf{M}}^{+}, \text{Project Resource User})$
12	$(\mathsf{M}\,;\overline{\mathsf{R}}, \text{Team Resource User})$

Table 2: Principal-matching policy

#	Authorization Rule
1	$(\text{Deliverable Client}, \star, \mathsf{read}, 1)$
2	$(\text{Deliverable Reviewer}, \star, \mathsf{read}, 1)$
3	$(\text{Deliverable Supervisor}, \star, \mathsf{read}, 1)$
4	$(\text{Deliverable Supervisor}, \star, \mathsf{write}, 1)$
5	$(\text{Deliverable User}, \star, \mathsf{read}, 1)$
6	$(\text{Project Resource Supervisor}, \star, \mathsf{read}, 1)$
7	$(\text{Project Resource Supervisor}, \star, \mathsf{write}, 1)$
8	$(\text{Project Resource User}, \star, \mathsf{read}, 1)$
9	$(\text{Project Resource User}, \mathsf{Func.Spec.}\#1, \mathsf{write}, 0)$
10	$(\text{Project Resource User}, \star, \mathsf{write}, 1)$
11	$(\text{Team Resource User}, \star, \mathsf{write}, 1)$

Table 3: Authorization policy

Table 4 lists some illustrative requests, together with the result of their evaluation. Requests 1 and 2 would result in the list of matched principals [Project Resource Supervisor, Project Resource User]. Requests 3 and 4 would result in the matched principal lists [Project Resource User] and [Deliverable reviewer], respectively, while the final request would match no principals.
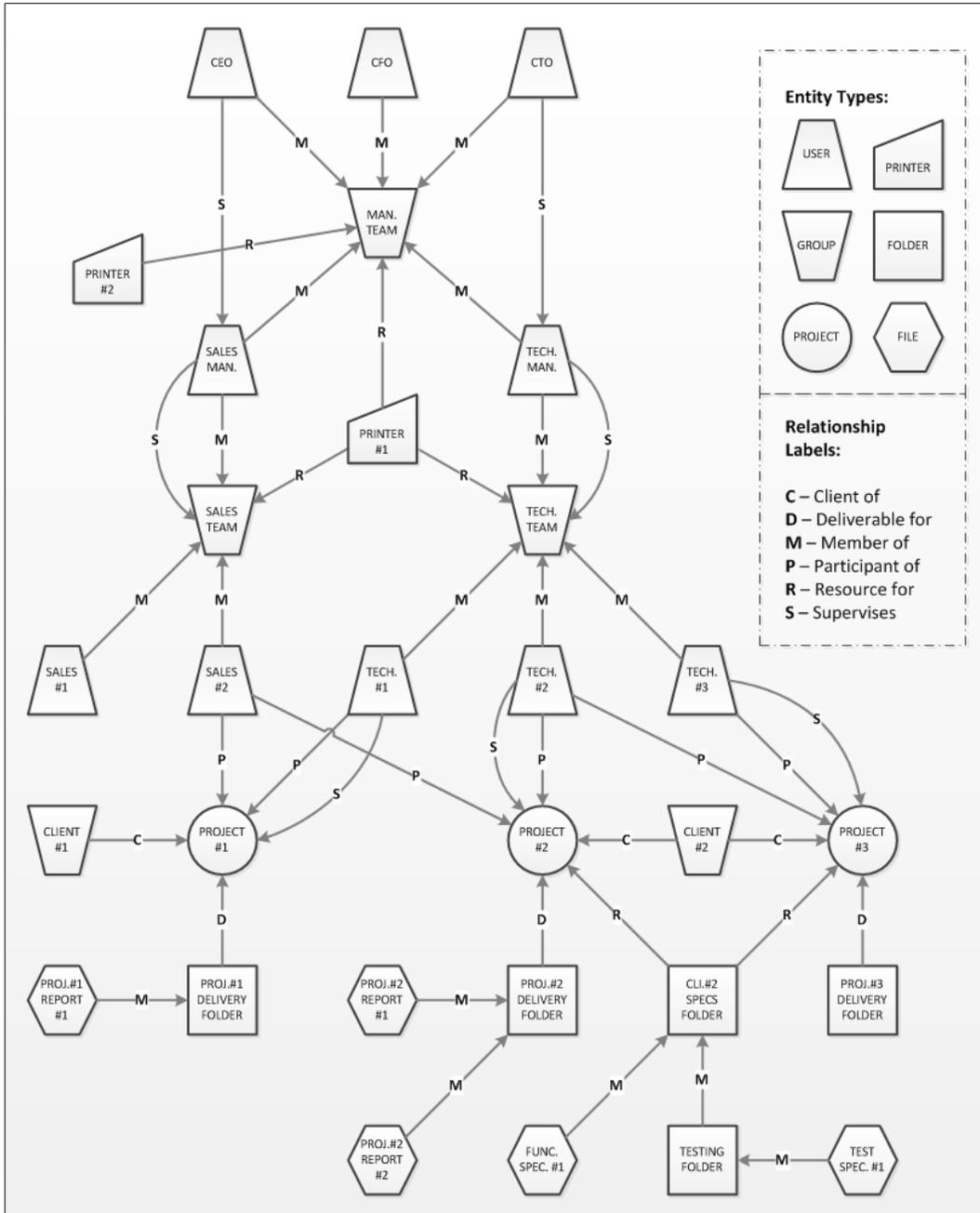
Figure 7: System graph

#	Request	Decision set	Outcome	Comment
1	(Tech.#2, Test.Spec.#1, read)	{1}	Allow	
2	(Tech.#2, Func.Spec.#1, write)	{1, 0}	Allow	First match
3	(Sales.#2, Func.Spec.#1, write)	{0}	Deny	
4	(CTO, Proj.#1 Report#1, read)	{1}	Allow	
5	(CEO, Proj.#1 Report#1, read)	{}	Deny	System default

Table 4: Sample requests