A User-to-User Relationship-based Access Control Model for Online Social Networks*

Yuan Cheng, Jaehong Park, and Ravi Sandhu

Institute for Cyber Security, University of Texas at San Antonio
ycheng@cs.utsa.edu, {jae.park, ravi.sandhu}@utsa.edu

Abstract. Users and resources in online social networks (OSNs) are interconnected via various types of relationships. In particular, user-to-user relationships form the basis of the OSN structure, and play a significant role in specifying and enforcing access control. Individual users and the OSN provider should be allowed to specify which access can be granted in terms of existing relationships. We propose a novel user-to-user relationship-based access control (UURAC) model for OSN systems that utilizes regular expression notation for such policy specification. We develop a path checking algorithm to determine whether the required relationship path between users for a given access request exists, and provide proofs of correctness and complexity analysis for this algorithm.

Keywords: Access Control, Security, Social Networks

1 Introduction

Access control in OSNs presents several unique characteristics different from traditional access control. In mandatory and role-based access control, a system-wide access control policy is typically specified by the security administrator. In discretionary access control, the resource owner defines access control policy. However, in OSN systems, users may want to regulate access to their resources and activities related to themselves, thus access in OSNs is subject to user-specified policies. Other than the resource owner, some related users (e.g., user tagged in a photo owned by another user, parent of a user) may also expect some control on how the resource or user can be exposed. To prevent users from accessing unwanted or inappropriate contents, user-specified policies that regulate how a user accesses information need to be considered in authorization as well. Thus, the system needs to collect these individualized partial policies, from both the accessing users and the target users, along with the system-specified policies and fuse them for the overall control decision.

In OSN, access to resources is typically controlled based on the relationships between the accessing user and the controlling user of the

* This work is supported by grants from the US National Science Foundation.

target found on the social graph. This type of relationship-based access control [10] takes into account the existence of a particular relationship or a particular sequence of relationships between users and expresses access control policies in terms of such user-to-user (U2U) relationships.

Facebook-like systems allow users to specify access control policy to related resources based on topology of the social graph, by choosing options such as “public”, “private”, “friend” or “friend of friend”. Circles in Google+ allow users to create customized relationships. In recent years, researchers have proposed more advanced relationship-based access control models, such as [1–9, 11]. Policies in [1–6, 8, 9] can be composed of multiple types of relationships. [4–6] also adopt the depth and the trust value of relationship to control the spread of information. Although only having the “friend” relationship type, [7] provides additional topology-based policies, such as known quantity, common friends and stranger of more than k distance. While these works have their own advantages, one of the common drawbacks they share is that they do not allow different relationship types and multiple possible types on each hop.

In this paper, we propose a novel user-to-user relationship-based access control (UURAC) model and a regular expression-based policy specification language which enable more sophisticated and fine-grained access control in OSNs. To the best of our knowledge, this is the first relationship-based access control model for OSNs with such capability.

2 Motivation and Related Work

This section discusses characteristics of access control in OSNs, related works, our approach and shows our contributions.

2.1 Characteristics of Access Control for OSNs

Below, we discuss some essential characteristics [13, 14] that need to be supported in access control solutions for OSN systems.

Policy Individualization. OSN users may want to express their own preferences on how their own or related contents should be exposed. A system-wide access control policy such as we find in mandatory and role-based access control, does not meet this need. Access control in OSNs further differs from discretionary access control in that users other than the resource owner are also allowed to configure the policies of the related resource. In addition, users who are related to the accessing user, e.g. parent to child, may want to control the accessing user’s actions. Therefore, the OSN system needs to collectively utilize these individualized policies from users related to the accessing user or the target, along with the system-specified policies for control decisions.

User and Resource as a Target. Unlike traditional user access where the access is against target resource, activities such as poking and friend recommendations are performed against other users. User as a target is particularly crucial for access control in OSNs since policies for users can specify rules for incoming actions as well as outgoing actions.

User Policies for Outgoing and Incoming Actions. Notification of a particular friends' activities could be bothersome and a user may want to block it. This type of policy is captured as incoming action policy. Also, a user may want to control her own or other users' activities. For example, a user may restrict her own access from any violent contents or a parent may not want her child to invite her coworker as a friend. This type of policy is captured as an outgoing action policy. In OSN, it is necessary to support policies for both types of actions.

Necessity for Relationship-based Access Control. Access control in OSNs is mainly based on relationships among users and resources. For example, only Alice's direct friends can access her blogs, or only user who owns the photo or tagged users can modify the caption of the photo. Depth is another significant parameter, since people tend to share resources with closer users (e.g., "friend", or "friend of friend").

2.2 Prior Access Control Models for OSNs

Fong et al [7] developed a formal algebraic model for access control in Facebook-like systems. This model generalizes the Facebook-style access control mechanism into two stages: reaching the search listing of the resource owner and accessing the resource. The model formalizes policies for accessing resources as well as policies for search, traversal and communications. The policy vocabulary supports expressing arbitrary topology-based properties, such as "k common friends" and "k clique", which are beyond what Facebook offers.

In [8], Fong proposed a formal model for social computing applications, in which authorization decisions are based on the user-to-user relationships. This model employs a modal logic language for policy specification. Fong et al extended the policy language and formally characterized its expressiveness power [9]. In contrast to [7], this model allows multiple relationship types and directional relationships. Relationships and authorizations are articulated in access contexts and context hierarchy to support sharing of relationships among contexts. Bruns et al [1] later improved [8, 9] by using hybrid logic to enable better efficiency in policy evaluation and greater flexibility of atomic formulas.

Carminati et al [4–6] proposed a series of access control solutions for OSNs where the access rules are specified by the users at their discretion. The access requirements that the accessing user must satisfy are specified

as type, depth, and trust metrics of the user-to-user relationships between the accessing user and the resource owner. The system features a centralized certificate authority that asserts the validity of the relationship path, while access control enforcement is carried out on decentralized user side.

In [2, 3], an access control model for OSNs is proposed by Carminati et al by utilizing semantic web technologies. Unlike many other works, this model exhibits different relationships between users and resources. It defines three kinds of access policies with the Web Ontology Language (OWL) and the Semantic Web Rule Language (SWRL), namely authorization, administration and filtering policies. Similar to [2, 3], Masoumzadeh et al [12] proposed ontology-based social network access control. Their model captures delegation of authority and empowers both users and the system to express finer-grained access control policies.

2.3 Comparison of Access Control Models for OSNs

The first four columns of Table 1 summarize the salient characteristics of the models discussed above. The fifth column gives these characteristics for the new UURAC model to be defined in this paper.

Table 1. Comparison of Access Control Models for OSNs

	Fong [7]	Fong [8, 9]	Carminati [6]	Carminati [2, 3]	UURAC
Relationship Category					
Multiple Relationship Types		✓	✓	✓	✓
Directional Relationship		✓	✓		✓
U2U Relationship	✓	✓	✓	✓	✓
U2R Relationship				✓	
Model Characteristics					
Policy Individualization	✓	✓	✓	✓	✓
User & Resource as a Target				(partial)	✓
Outgoing/Incoming Action Policy				(partial)	✓
Relationship Composition					
Relationship Depth	0 to 2	0 to n	1 to n	1 to n	0 to n
Relationship Composition	f, f of f	exact type sequence	path of same type	exact type sequence	path pattern of different types

All the models deal only with U2U relationships, except [2, 3] also recognize U2R (user-to-resource) relationships explicitly. U2R relationships can be captured implicitly via U2U with the last hop being U2R. Nevertheless, we believe that explicit treatment of U2R and R2R (resource-to-resource) relationships is important but leave it for future work.

2.4 Our Contributions

This paper develops a novel UURAC model for OSNs, using regular expression notation. UURAC supports policy individualization, user and resource as a target, distinction of user policies for outgoing and incoming actions, and relationship-based access control. It incorporates greater

generality of path patterns in its policy specifications than prior models, including the incorporation of inverse relationships. We also provide an effective path checking algorithm for access control policy evaluation, along with proofs of correctness and complexity analysis.

3 UURAC Model Foundation

In this section, we develop the foundation of UURAC including basic notations, access control model components and social graph model.

3.1 Basic Notations

We write Σ to denote the set of relationship type specifiers, where $\Sigma = \{\sigma_1,\sigma_2,\ldots,\sigma_n,\sigma_1^{-1},\sigma_2^{-1},\ldots,\sigma_n^{-1}\}$. Each relationship type specifier σ is represented by a character recognizable by regular expression parser. Given a relationship type $\sigma_i \in \Sigma$, the inverse of the relationship is $\sigma_i^{-1} \in \Sigma$.

We differentiate the active and passive forms of an action, denoted *action* and $action^{-1}$, respectively. If Alice pokes Bob, the action is *poke* from Alice's viewpoint, whereas it is $poke^{-1}$ from Bob's viewpoint.

3.2 Access Control Model Components

The model comprises five categories of components as shown in Figure 1.

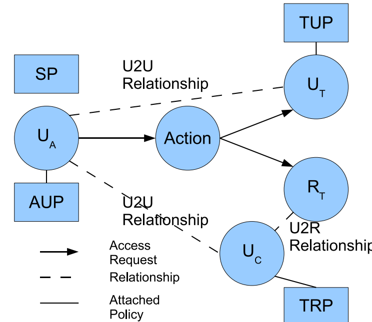

Fig. 1. Model Components

Accessing User (u_a) represents a human being who performs activities. An accessing user carries access control policies and U2U relationships with other users.

Each **Action** is an abstract function initiated by accessing user against target. Given an action, we say it is *action* for the accessing user, but $action^{-1}$ for the recipient user or resource.

Target is the recipient of an action. It can be either *target user* (u_t) or *target resource* (r_t). Target user has her own policies and U2U relationship information, both of which are used for authorization decisions. Target resource has U2R relationship (i.e., ownership) with *controlling users* (u_c). An accessing user must have the required U2U relationships with the controlling user in order to access the target resource.

Access Request denotes an accessing user's request of a certain type of action against a target. It is modeled as a tuple $< u_a, action, target >$, where $u_a \in U$ is the accessing user, *target* is the user or resource that

u_a tries to access, whereas $action \in Act$ specifies from a finite set of supported functions in the system the type of access the user wants to have with *target*. If u_a requests to interact with another user, $target = u_t$, where $u_t \in U$ is the target user. If u_a tries to access a resource owned by another user u_c, *target* is resource $r_t \in R$ where R is a finite set of resources in OSN.

Policy defines the rules according to which authorization is regulated. As shown in Figure 2, policies can be categorized into user-specified and system-specified policies, with respect to who defines the policies. System-specified policies (SP) are system-wide general rules enforced by the OSN system; while user-specified policies are applied to specific users and resources. Both user- and system-specified policies include policies for resources and policies for users. Policies for resources are used to control who can access a resource, while policies for users regulate how users can behave regarding an action. User-specified policies for a resource are called *target resource policies* (TRP), which are policies for *incoming actions*. User-specified policies for users can be further divided into *accessing user policies* (AUP) and *target user policies* (TUP), which correspond to user's outgoing and incoming access (see examples in Section 2.1), respectively. *Accessing user policies*, also called *outgoing action policies*, are associated with the accessing user and regulate this user's outbound access. *Target user policies*, also called *incoming action policies*, control how other users can access the target user. Note that system-specified policies do not have separate policies for incoming and outgoing actions, since the accessor and target are explicitly identified.

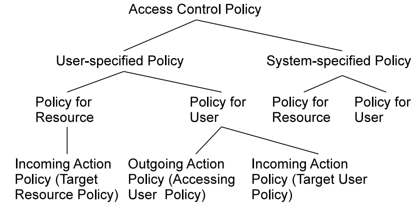

Fig. 2. Access Control Policy Taxonomy

3.3 Modeling Social Graph

As shown in Figure 3, an OSN forms a directed labeled simple graph[1] with nodes (or vertices) representing users and edges representing user-to-user relationships. We assume every user owns a finite set of resources and specifies access control policies for the resources and activities related to her. If an accessing user has the U2U relationship required in the policy, the accessing user will be granted permission to perform the requested action against the corresponding resource or user.

We model the social graph of an OSN as a triple $G =< U, E, \Sigma >$:

[1] A simple graph has no loops (i.e., edges which start and end on the same vertex) and no more than one edge of a given type between any two different vertices.

- U is a finite set of registered users in the system, represented as nodes (or vertices) on the graph. We use the terms user and node interchangeably from now on.
- $\Sigma = \{\sigma_1, \sigma_2, .., \sigma_n\, \sigma_1^{-1}, \sigma_2^{-1}, .., \sigma_n^{-1}\}$ denotes a finite set of relationship types, where each type specifier σ denotes a relationship type supported in the system.
- $E \subseteq U \times U \times \Sigma$, denoting social graph edges, is a set of existing user relationships.

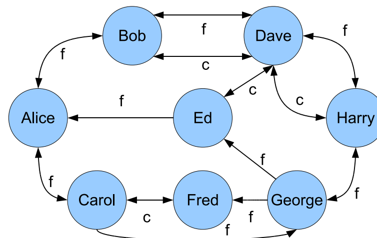

Fig. 3. A Sample Social Graph

Since not all the U2U relationships in OSNs are mutual, we define the relationships E in the system as directed. For every $\sigma_i \in \Sigma$, there is $\sigma_i^{-1} \in \Sigma$ representing the inverse of relationship type σ_i. We do not explicitly show the inverse relationships on the social graph, but assume the original relationship and its inverse twin always exist simultaneously. Given a user $u \in U$, a user $v \in U$ and a relationship type $\sigma \in \Sigma$, a relationship (u, v, σ) expresses that there exists a relationship of type σ starting from user u and terminating at v. It always has an equivalent form (v, u, σ^{-1}). $G =< U, E, \Sigma >$ is required to be a simple graph.

4 UURAC Policy Specifications

This section defines a regular expression based policy specification language, to represent various patterns of multiple relationship types.

4.1 Path Expression Based Policy

The user relationship path in access control policies is represented by regular expressions. The formulas are based on a set Σ of relationship type specifiers. Each specification in this language describes a pattern of required relationship types between the accessing user and the target/controlling user. We use three kinds of wildcard notations that represent different occurrences of relationship types: asterisk (*) for 0 or more, plus (+) for 1 or more and question mark(?) for 0 or 1.

4.2 Graph Rule Specification and Grammar

An access control *policy* consists of a requested action, optional target resource and a required *graph rule*. In particular, *graph rule* is defined as (*start*, *path rule*), where *start* denotes the starting node of relationship path evaluation, whereas *path rule* represents a collection of *path specs*. Each path spec consists of a pair (*path*, *hopcount*), where *path* is a sequence of characters, denoting the pattern of relationship path between

two users that must be satisfied, while *hopcount* limits the maximum number of edges on the path.

Typically, a user can specify one piece of policy for each action regarding a user or a resource in the system, and the *path rule* in the policy is composed of one or more *path specs*. Policies defined by different users for the same action against same target are considered as separate policies. Multiple *path specs* can be connected by disjunction or conjunction. For instance, a path rule $(f*, 3) \vee (\Sigma*, 5) \vee (fc, 2)$, where f is friend and c is coworker, contains disjunction of three different pieces of path specs, of which one must be satisfied in order to grant access. Note that, there might be a case where only users who do not have particular types of relationships with the target are allowed to access. To allow such negative relationship-based access control, a boolean negation operator over *path specs* is allowed, which implies the non-existence of the specified pair of relationship type pattern *path* and hopcount limit *hopcount* following $\neg$. For example, $\neg\ (fc+, 5)$ means the involved users should not have relationship of pattern $fc+$ within depth of 5 in order to get access.

Table 2. Grammar for graph rules

$GraphRule ::=$ "(" $< StartingNode >$ "," $< PathRule >$ ")"
$PathRule ::= < PathSpecExp > | < PathSpecExp >< Connective >< PathRule >$
$Connective ::= \vee | \wedge$
$PathSpecExp ::= < PathSpec > | \neg < PathSpec >$
$PathSpec ::=$ "(" $< Path >$ "," $< HopCount >$ ")" | "(" $< EmptySet >$ "," $< Hopcount >$ ")"
$HopCount ::= < Number >$
$Path ::= < TypeExp > | < TypeExp >< Path >$
$EmptySet ::= \emptyset$
$TypeExp ::= < TypeSpecifier > | < TypeSpecifier >< Wildcard >$
$StartingNode ::= u_a | u_t | u_c$
$TypeSpecifier ::= \sigma_1 | \sigma_2 | .. | \sigma_n | \sigma_1^{-1} | \sigma_2^{-1} | .. | \sigma_n^{-1} | \Sigma$ where $\Sigma = \{\sigma_1, \sigma_2, .., \sigma_n, \sigma_1^{-1}, \sigma_2^{-1}, .., \sigma_n^{-1}\}$
$Wildcard ::=$ " $*$ " | "?" | " $+$ "
$Number ::= [0 - 9]+$

Each graph rule usually specifies a starting node, the required types of relationships between the starting node and the evaluating node, and the hopcount limit of such relationship path. A grammar describing the syntax of such policy language is defined in Table 2. Here, *GraphRule* stands for the graph rule to be evaluated. *StartingNode* can be either the accessing user u_a, the target user u_t or the controlling user u_c, denoting the given node from which the required relationship path begins. *Path* represents a sequence of type specifiers from the starting node to the evaluating node. *Path* will typically be non-empty. If *path* is empty and *hopcount* $= 0$ we assign the special meaning of "only me". *Wildcard* captures the three wildcard characters, which facilitate specifying more powerful and expressive path expressions. Given a graph rule from the access control policy, this grammar specifies how to parse the expres-

sion and to extract the containing path pattern and hopcount from the expression.

4.3 User- and System-specified Policy Specifications

User-specified policies specify how individual users want their resources or services related to them to be released to other users in the system. These policies are specific to actions against a particular resource or user. System-specified policies allow the system to specify access control on users and resources. Different from user policies, the statements in system policies are not specific to particular accessing user or target, but rather focus on the entire set of users or resources (see Table 3).

Table 3. Access Control Policy Representations

Accessing User Policy	$< action, (start, path\ rule)>$
Target User Policy	$< action^{-1}, (start, path\ rule)>$
Target Resource Policy	$< action^{-1}, r_t, (start, path\ rule)>$
System Policy for User	$< action, (start, path\ rule)>$
System Policy for Resource	$< action, r.type, (start, path\ rule)>$

In *accessing user policy*, *action* denotes the requested action, whereas ($start$, $path\ rule$) expresses the graph rule. Similarly, $action^{-1}$ in *target user policy* and *target resource policy* is the passive form of the corresponding *action* applied to target user. Target resource policy contains an extra parameter r_t, representing the resource to be accessed.

This paper considers only U2U relationships in policy specification. In general, there could be one or more controlling users who have certain types of U2R relationships with the resource and possess policies for the corresponding target resource. For simplicity, we assume the only such U2R relationship is ownership. To access the resource, the accessing user must have the required relationships with the controlling user. The policies associated with the controlling users are defined on the basis of per action per resource. For instance, when querying *read* access request on r_t, $owner(r_t)$ returns the list of users who have ownership with r_t. Access to r_t is under the authority of all the controlling users who have *read* policies for r_t. Note that in this paper we are not introducing the policy administration model, so who can specify the policy is not discussed.

System-specified policies do not differentiate the active and passive forms of an action. *System policy for users* carries the same format as accessing user policy does. However, when specifying *system policy for resources*, one system-wide policy for one type of access to all resources may not be fine-grained and flexible enough. Sometimes we need to refine the scope of the resources that applied to the policies in terms of resource types *r.type*. Examples of resource type *r.type* are photo, blog post, status update, etc. Thus, $<read, photo, (u_c, f*, 4)>$ is a system policy applied to all *read* access to photos in the system.

4.4 Access Evaluation Procedure

Algorithm 1 $AccessEvaluation(u_a, action, target)$

1: (Policy Collecting Phase)
2: **if** $target = u_t$ **then**
3: $AUP \leftarrow u_a$'s policy for *action*, $TUP \leftarrow u_t$'s policy for $action^{-1}$, $SP \leftarrow$ system's policy for *action*
4: **else**
5: $u_c \leftarrow owner(r_t)$, $AUP \leftarrow u_a$'s policy for *action*, $TRP \leftarrow u_c$'s policy for $action^{-1}$ on r_t, $SP \leftarrow$ system's policy for $action, r.type$
6: (Policy Evaluation Phase)
7: **for all** policies in AUP, TUP/TRP and SP **do**
8: Extract graph rules (*start*, *path rule*) from policies
9: **for all** graph rules extracted **do**
10: Determine the starting node, specified by *start*, where the path evaluation starts
11: Determine the evaluating node which is the other user involved in access
12: Extract path rules *path rules* from graph rules
13: Extract each path spec *path*, *hopcount* from path rules
14: Path-check each path spec using Algorithm 2
15: Evaluate a combined result based on conjunctive or disjunctive connectives between path specs
16: Compose the final result from the result of each policy

Algorithm 1 specifies how the access evaluation procedure works. When an accessing user u_a requests an *action* against a target user u_t, the system will look up u_a's *action* policy, u_t's $action^{-1}$ policy and the system-specified policy corresponding to *action*. When u_a requests an *action* against a resource r_t, the system will first find out the controlling user u_c via $owner(r_t)$ and retrieve all the corresponding policies. Although each user can only specify one policy per action per target, there might be multiple users specifying policies for the same pair of action and target. Multiple policies might be collected in each of the three policy sets: AUP, TUP/TRP and SP.

Example Given the following policies and social graph in Figure 3:

- Alice's policy P_{Alice}: $< poke, (u_a, (f*, 3))> < poke^{-1}, (u_t, (f, 1))> < read, (u_a, (\Sigma*, 5))> < read^{-1}, file1, (u_c, (cf*, 4))>$
- Harry's policy P_{Harry}: $< poke, (u_a, (cf*, 5) \vee (f*, 5))> < poke^{-1}, (u_t, (f*, 2))> < read^{-1}, file2, (u_c, \neg(p+, 2)>$
- System's policy P_{Sys}: $< poke, (u_a, (\Sigma*, 5))> < read, photo, (u_a, (\Sigma*, 5))>$

When Alice requests to poke Harry, the system will look up the following policies: $< poke, (u_a, (f*, 3))>$ from P_{Alice}, $< poke^{-1}, (u_t, (f*, 2))>$ from P_{Harry}, and $< poke, (u_a, (\Sigma*, 5))>$ from P_{Sys}. When Alice requests to read photo $file2$ owned by Harry, the policies $< read, (u_a, (\Sigma*, 5))>$ from P_{Alice}, $< read^{-1}, file2, (u_c, \neg(p+, 2)>$ from P_{Harry}, and $< read, photo, (u_a, (\Sigma*, 5))>$ from P_{Sys} will be used for authorization.

For all the policies in the policy sets, the algorithm first extracts the graph rule (*start*, *path rule*) from each policy. Once the graph rule is extracted, the system can determine where the path checking evaluation starts (using *start*), and then extracts every path spec *path*, *hopcount*

(from *path rules*). Then, it runs a path-checking algorithm (see the next section) for each path spec. The path-checking algorithm returns a boolean result for each path spec. To get the evaluation result of a particular policy, we combine the results of all path specs in the policy using conjunction, disjunction and negation. At last, the final evaluation result for the access request is made by composing all the evaluation results of the policies in the chosen policy sets.

4.5 Discussion

The existence of multi-user policies can result in decision conflicts. To resolve this, we can adopt a disjunctive, conjunctive, or prioritized approach. When a disjunctive approach is enabled, the satisfaction of any corresponding policy is sufficient for granting the requested access. In a conjunctive approach, the requirements of every involved policy should be satisfied in order that the access request would be granted. In a prioritized approach, if, for example, parents' policies get a priority over children's policies, the parents' policies overrule children's policies. While policy conflicts are inevitable in the proposed model, we do not discuss this issue in further detail here. For simplicity we assume system level policies are available to resolve conflicts in user-specified authorization policies and do not consider user-specified conflict resolution policies.

One observation from user-specified policies is that *action* policy starts from u_a whereas $action^{-1}$ policy starts from u_t. This is because at the time of policy configuration, users are not aware of who are the other participants in the action hence cannot specify graph rule starting from the other side. When $hopcount = 0$ and *path* equals to empty, it has special meaning of "only me". For instance, $<$ *poke*, (u_a, ($\emptyset$, 0))$>$ says that u_a can only poke herself, and $<$ $poke^{-1}$, (u_t, ($\emptyset$, 0))$>$ specifies u_t can only be poked by herself. The above two policies give a complementary expressive power that the regular policies do not cover, since regular policies are simply based on existing paths and limited hopcount.

As mentioned earlier, the social graph is modeled as a simple graph. Further we only allow simple path with no repeating nodes. Avoiding repeating nodes on the relationship path prevents unnecessary iterations among nodes that have been visited already and unnecessary hops on these repeating segments. On the other hand, this "no-repeating" could be quite useful when a user wants to expose her resource to farther users without granting access to nearer users. For example, in a professional OSN system such as LinkedIn, a user may want to promote her resume to users outside her current company, but does not want her coworkers to know about it. Note that the two distinct paths denoted by $(fffc)$ and (fc) may co-exist between a pair of users. The path specs $fffc \wedge \neg fc$

allows the coworkers of the user's distant friends to see the resume, while the coworkers of the user's direct friends (fc) are not authorized.

In general, conventional OSNs are susceptible to the multiple-persona problem, where users can always create a second persona to get default permissions. In a default-denial system, a new persona initially has no permission to access others, thus allowing multiple new personas from the same user is safe to the existing users. Our approach follows the default-denial design, which means if there is no explicit positive authorization policy specified, there is no access permitted at all. Based on the default-denial assumption, negative authorizations in our policy specifications are mainly used to further refine permissions allowed by the positive authorizations specified (e.g., $f * c \wedge \neg fc$). A single negative authorization without any positive authorization has the same effect as there is no policy specified at all. Nonetheless it is possible for the coworker of a direct friend to have a second persona that meets the criteria for coworker of a distant friend and thereby acquires access to the resume. Without strong identities we can only provide persona level control in such policies.

5 Path Checking Algorithm

In this section, we present the algorithms for determining if there exists a qualified path between two involved users in an access request.

As mentioned, in order to grant access, relationships between the accessor and the target/controlling user must satisfy the graph rules specified in access control policies regarding the given request. We formulate the problem as follows: given a social graph G, an access request $< u_a$, *action*, $target >$ and an access policy, the system decision module explores the graph and verifies the existence of a path between u_a and $target$ (or u_c of $target$) matching the graph rule $<$ *start*, *path rule* $>$.

Path checking is performed by Algorithm 2, which takes as input the social graph G, the path pattern *path* and the hopcount limit *hopcount* specified by *path spec* in the policy, the starting node s specified by *start* and the evaluating node t which is the other user involved, and returns a boolean value as output. Note that *path* is non-empty, so this algorithm only copes with cases where *hopcount* $\neq 0$. The starting node s and the evaluating node t can be either the accessing user or the target/controlling user, depending on the given policy. The algorithm starts by constructing a DFA (deterministic finite automata) from the regular expression *path*. The REtoDFA() function receives *path* as input, and converts it to a NFA (non-deterministic finite automata) then to a DFA, by using the well-known Thompson's Algorithm [16] and Subset Construction Algorithm (also known as Büchi's Algorithm) [15], respectively.

Algorithm 2 $PathChecker(G, path, hopcount, s, t)$

1: $DFA \leftarrow REtoDFA(path)$; $currentPath \leftarrow NIL$; $d \leftarrow 0$
2: $stateHistory \leftarrow$ DFA starts at the initial state
3: **if** $hopcount \neq 0$ **then**
4:    **return** DFST(s)

Algorithm 3 $DFST(u)$

1: **if** $d + 1 > hopcount$ **then**
2:    **return** FALSE
3: **else**
4:    **for all** (v, σ) where (u, v, σ) in G **do**
5:       **switch**
6:       **case 1** $v \in currentPath$
7:       break
8:       **case 2** $v \notin currentPath$ and $v = t$ and DFA with transition σ is at accepting state
9:       $d \leftarrow d + 1; currentPath \leftarrow currentPath.(u, v, \sigma)$
10:       $currentState \leftarrow$ DFA takes transition σ
11:       $stateHistory \leftarrow stateHistory.(currentState)$
12:       **return** TRUE
13:       **case 3** $v \notin currentPath$ and $v = t$ and transition σ is valid for DFA but DFA with transition σ is not at accepting state
14:       break
15:       **case 4** $v \notin currentPath$ and $v \neq t$ and transition σ is invalid for DFA
16:       break
17:       **case 5** $v \notin currentPath$ and $v \neq t$ and transition σ is valid for DFA
18:       $d \leftarrow d + 1; currentPath \leftarrow currentPath.(u, v, \sigma)$
19:       $currentState \leftarrow$ DFA takes transition σ
20:       $stateHistory \leftarrow stateHistory.(currentState)$
21:       **if** (DFST(v)) **then**
22:          **return** TRUE
23:       **else**
24:          break
25:    **if** $d = 0$ **then**
26:       **return** FALSE
27:    **else**
28:       $d \leftarrow d - 1; currentPath \leftarrow currentPath \backslash (u, v, \sigma)$
29:       $previousState \leftarrow$ last element in $stateHistory$
30:       DFA backs off the last taken transiton σ to $previousState$
31:       $stateHistory \leftarrow stateHistory \backslash (previousState)$
32:       **return** FALSE

The algorithm uses a depth-first search (DFS) to traverse the graph, because it requires only one running DFA and, correspondingly, one pair of variables keeping the current status and the history of exploration in a DFS traversal. Whereas, a breadth-first search (BFS) traversal has to maintain multiple DFAs and multiple variables simultaneously and switch between these DFAs back and forth constantly, which makes the costs of memory space and I/O operations proportional to the number of nodes visited during exploration. Note that DFS could avoid a target node for a longer time, even if the node is close to the starting node. If the hopcount is unlimited, a DFS traversal may pursue lengthy useless

exploration. However, activities in OSN typically occur among people with close relationships. Hence, DFS with limited hopcount fits our model.

The variable $currentPath$, initialized as NIL, holds the sequence of the traversed edges between the starting node and the current node. Variable $stateHistory$, initialized as the initial DFA state, keeps the history of DFA states during algorithm execution. The main procedure starts by setting d to 0 and launches the DFS traversal function $DFST()$, given in Algorithm 3, from the starting node s.

Given a node u, if $d + 1$ does not exceed the hopcount limit, it indicates that traversing one step further from u is allowed. Otherwise, the algorithm returns false and goes back to the previous node. If further traversal is allowed, then the algorithm picks up an edge (u, v, σ) from the list of the incident edges leaving u. If (u, v, σ) is unvisited, we get the node v on the opposite side of the edge (u, v, σ). Now we have five different cases. If v is on $currentPath$, we will never visit v again, because doing so creates a cycle on the path. Rather, the algorithm breaks out of for loop, and finds the next unchecked edges of u. When v is not on $currentPath$ and v is the target node t and DFA taking transition σ reaches an accepting state, we find a path between s and t matching the pattern $Path$. We increment d by one, concatenate edge (u, v, σ) to $currentPath$, and save the current DFA state to history. If v is the target node but DFA with transition σ is not at an accepting state, then the path from s to v does not match the pattern. When v is not on $currentPath$ and is not the target node, there are two cases depending on whether the transition type σ is a valid transition for DFA. If it is not, we break out of for loop and continue to check the next unchecked edge of u. Otherwise, the algorithm increments d by one, concatenates e to $currentPath$, moves DFA to the next state via transition type σ, updates the DFA state history, and repeatedly executes $DFST()$ from node v. If the recursive function call discovers a matching path, the previous call also returns true. Otherwise, it checks next edge of node u.

After all the outgoing edges of u have been checked, the algorithm has to step back to the previous node of u and reset all variables to the previous values. But if $d = 0$, all the outgoing edges of the starting node are checked, thus the whole execution completes without a matching path.

In Figure 3, suppose user Harry owns a resource r_t and expresses the target resource policy as ($read^{-1}$, r_t, ($f * cf*$,3)), where $read$ is the permitted action, ($f * cf*$, 3) is the path pattern and hopcount limit. Path pattern $f*cf*$ means the accessing user and Harry must be either a pair of coworkers (c) or direct or indirect friend (f) of a pair of coworkers. Hopcount 3 constrains the distance between the two users to be within three hops. Figure 4 shows the DFA accepting the path pattern $f*cf*$. If

Alice requests read access to the resource owned by Harry, the algorithm starts exploration from node H (Harry) by checking all the edges leaving H. If it picks the edge (H, D, f) or (H, D, c) first, it will eventually find out that there exists a satisfiable path (H, D, f), (D, E, c), (E, A, f) or (H, D, c), (D, B, f), (B, A, f) that also moves the DFA from the starting state π_0 to the accepting state π_3 in three hops. (H, G, f), (G, F, f), (F, C, c), (C, A, f) also matches the path pattern, but it is invalid because it takes four hops to reach node A.

Suppose Harry specifies a target user policy for him as $(poke^{-1}, (f+, 2))$. This implies only his friends or indirect friends can poke him. Then, Bob, Dave, Ed, Fred and George can poke Harry because the paths between Harry and them contain relationship f and are within depth of two. Carol and Harry do not have friend relationship with Harry, while Alice is too far away from Harry.

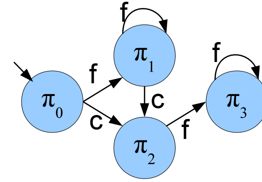

Fig. 4. DFA for $f * cf*$

6 Conclusions and Future Work

We proposed a UURAC model and a regular expression based policy specification language. We provided a DFS-based path checking algorithm and established its correctness and complexity. Correctness of the algorithm is proved by induction on hopcount. Due to the sparseness nature of social graph, given the constraints on relationship types and hopcount limit in policy, the complexity of the algorithm can be dramatically reduced. Proofs of correctness and complexity are given in appendix.

While this work only uses user-to-user relationships for authorization, we plan to extend our model to exploit user-to-resource and resource-to-resource relationships. To improve the expressiveness of the model, we also plan to incorporate some predicate expressions for attribute-based control and filtering users and relationships. Another future direction is to capture some unconventional relationships in OSNs, such as temporary relationships (i.e., vicinity) and one-to-many relationships (i.e., network, group). Last but not least, we will be working on implementing our approach into a prototype and doing some experiments to analyze the approach.

References

1. G. Bruns, P. W. Fong, I. Siahaan, and M. Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *ACM CODASPY*, 2012.
2. B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. A semantic web based framework for social network access control. In *ACM SACMAT*, 2009.

3. B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. Semantic web-based social network access control. *Computers and Security*, 30(2C3), 2011. Special Issue on Access Control Methods and Technologies.
4. B. Carminati, E. Ferrari, and A. Perego. Rule-based access control for social networks. In *OTM 2006 Workshops*, volume 4278 of *LNCS*. Springer, 2006.
5. B. Carminati, E. Ferrari, and A. Perego. A decentralized security framework for web-based social networks. *Int. Journal of Info. Security and Privacy*, 2(4), 2008.
6. B. Carminati, E. Ferrari, and A. Perego. Enforcing access control in web-based social networks. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.
7. P. Fong, M. Anwar, and Z. Zhao. A privacy preservation model for Facebook-style social network systems. In *ESORICS 2009*. Springer, 2009.
8. P. W. Fong. Relationship-based access control: protection model and policy language. In *ACM CODASPY*, 2011.
9. P. W. Fong and I. Siahaan. Relationship-based access control policies and their policy languages. In *ACM SACMAT*, 2011.
10. C. E. Gates. Access control requirements for web 2.0 security and privacy. In *Proc. of Workshop on Web 2.0 Security and Privacy (W2SP 2007)*, 2007.
11. S. Kruk, S. Grzonkowski, A. Gzella, T. Woroniecki, and H. Choi. D-FOAF: Distributed identity management with access rights delegation. *LNCS*, 4185, 2006.
12. A. Masoumzadeh and J. Joshi. Osnac: An ontology-based access control model for social networking systems. In *IEEE Social Computing (SocialCom)*, 2010.
13. J. Park, R. Sandhu, and Y. Cheng. Acon: Activity-centric access control for social computing. In *Int. Conf. on Availability, Reliability and Security (ARES)*, 2011.
14. J. Park, R. Sandhu, and Y. Cheng. A user-activity-centric framework for access control in online social networks. *Internet Computing, IEEE*, 15(5), sept.-oct. 2011.
15. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3, April 1959.
16. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11, June 1968.

A Proof of Correctness

Theorem 1. *Algorithm 2 will halt with true or false.*

Proof. Base case ($Hopcount = 1$): d is initially set to 0. Each outgoing edge from the starting node s will be examined once and only once. If taking an edge reaches the target node t and its type matches the language $Path$ denotes (case 2), the algorithm returns true. If the edge type matches the prefix of an expression in $L(Path)$ (lines 17-24), d increments to 1 followed by a recursive call to $DFST()$. The second level call will return false, since incremented d has exceeded $Hopcount$. In all other cases, the examined edge is discarded and d remains the same. Eventually, if a matching edge is not found, the algorithm will go through every outgoing edge from s and exit with false thereafter (lines 25-26).

Induction step: Assume when $Hopcount = k$ ($k \geq 1$), Theorem 1 is true. When $Hopcount$ is $k + 1$, all the $(k + 1)$th level recursive calls will examine every outgoing edge from the $(k + 1)$th node on $currentPath$. If visiting an edge reaches t and the updated $currentPath$ matches $L(Path)$,

the $(k+1)$th level call returns true and exits to the previous level, making all of the previous level calls all the way back to the first level exit with true as well. If an edge falls into case 5, d is incremented to $k+2$ and a $(k+2)$th level recursive call invokes, which will halt with false and return to the $(k+1)$th level as d has exceeded $Hopcount$. After all edges are examined without returning true, the algorithm will exit with false to the previous level. In the kth level, when $Hopcount = k+1$, edges without taking a recursive call are treated the same as they are when $Hopcount = k$. Since when $Hopcount = k$ the theorem holds, the algorithm will terminate with true or false when $Hopcount = k+1$ as well.

Lemma 1. *At the start and end of each $DFST()$ call, the DFA corresponding to Path is at currentState reachable from the starting state π_0 by transitions corresponding to the sequence of symbols in $currentPath$.*

Proof. The proof is straightforward. New edge is added to $currentPath$ only when it reaches the target node (lines 8-12) or it may possibly lead to the target node by taking a recursive $DFST()$ call (lines 17-24). In both cases the DFA starting from π_0 will move to *currentState* by taking the transition regarding the edge. Removing the last edge on $currentPath$ after all edges leaving the current node are checked always accompanies one step back-off of the DFA to its previous state (lines 28-32), which can eventually take the DFA all the way back to the starting state π_0.

Theorem 2. *If Algorithm 2 returns true, $currentPath$ gives a simple path of length less than or equal to $Hopcount$ and the string described by $currentPath$ belongs to the language described by $Path$ ($L(Path)$). If Algorithm 2 returns false, there is no simple path p of length less than or equal to $Hopcount$ such that the string representing p belongs to $L(Path)$.*

Proof. Base case ($Hopcount = 1$): At first, $d = 0$, $currentPath$ = NIL, and the DFA is at the starting state π_0. When $d = 0$, case 1 requires that the edge being checked is a self loop which is not allowed in a simple graph. $DFST()$ only returns true in case 2, where edge (s, t, σ) to be added to *currentPath* finds the target node t in one hop. The transition σ moves the DFA to an accepting state. Case 5 cannot return true, because incrementing d by one will exceed $Hopcount$ in the recursive $DFST()$ run. When DFST() exits with true, due to Lemma 1, $currentPath$, which is (s, t, σ), can move the DFA from π_0 to an accepting state π_1, implying that $\sigma \in L(Path)$. If the first $DFST()$ call returns false (lines 29-30), the algorithm has searched all the edges leaving node s. However, these examined edges either do not match the pattern specified by $L(Path)$ (case 2 and 3), or may possibly match $L(Path)$ but require more than one hop (case 5). Hence, Theorem 2 is true when $Hopcount = 1$.

Induction step: Assume when $Hopcount = k$ ($k \geq 1$), Theorem 2 is true. For the same G, $Path$, s and t, executions of $DFST()$ when $Hopcount = k$ and $k+1$ only differ after invoking the recursive $DFST()$ call in case 5. If an edge being checked can make the algorithm return true when $Hopcount = k$, $currentPath$ is a string of length $\leq k$ which is in $L(Path)$. When $Hopcount$ is $k+1$, the same $currentPath$ gives the same string and is of length $< k+1$, thus making the function exit with true as well. The only difference between $Hopcount = k$ and $Hopcount = k+1$ is that adding edges that lie in case 5 to $currentPath$ and incrementing d by one may not exceed the larger $Hopcount$ during the recursive call. If taking one of these edges leads to the target node and its corresponding transition moves the DFA to an accepting state, the algorithm will return true. The new $currentPath$ gives a simple path of length $k+1$ that connects node s and t. The algorithm only returns true in these two scenarios. In both scenarios, based on Lemma 1, the DFA can reach an accepting state by taking the transitions corresponding to $currentPath$, so the string corresponding to $currentPath$ is in $L(Path)$. If the algorithm returns false when $Hopcount = k$, there is no simple path p of length $\leq k$, where the string of symbols in p is in $L(Path)$. When $Hopcount$ is $k+1$, given the same G, such a path still does not exist. By taking a recursive $DFST()$ call in case 5, the algorithm will go through all 5 cases again to check all the edges leaving the new node. If the recursive call returns false, it means there is no simple path of length $k+1$ with its string of symbols in $L(Path)$. Combining the results from all $k+1$ level recursive calls, there exists no simple path of length $\leq k+1$ with its string of symbols in $L(Path)$. Hence, Theorem 2 is true when $Hopcount = k+1$.

B Complexity

In this algorithm, every possible path from s to t will be visited at most once until it fails to reach t, while every outgoing edge of a visited node may be checked multiple times during the search. In the extreme case, where every relationship type is acceptable and the graph is a complete directed graph, the overall complexity would be $O(|V|^{Hopcount})$. However, users in OSNs usually connect with a small group of users directly, thus the social graph is actually very sparse. We define the maximum and minimum out-degree of node on the graph as $dmax$ and $dmin$, respectively. Then, the time complexity can be bounded between $O(dmin^{Hopcount})$ and $O(dmax^{Hopcount})$. Given the constraints on the relationship types and hopcount limit in the policies, the size of graph to be explored can be dramatically reduced. The recursive DFST() call terminates as soon as either a matching path is found or the hopcount limit is reached.