Attribute-Aware Relationship-Based Access Control for Online Social Networks

Yuan Cheng, Jaehong Park, Ravi Sandhu

Attribute-aware Relationship-based Access Control for Online Social Networks*

Yuan Cheng, Jaehong Park, and Ravi Sandhu

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

Abstract. Relationship-based access control (ReBAC) has been adopted as the most prominent approach for access control in online social networks (OSNs), where authorization policies are typically specified in terms of relationships of certain types and/or depth between the access requester and the target. However, using relationships alone is often not sufficient to enforce various security and privacy requirements that meet the expectation from today's OSN users. In this work, we integrate attribute-based policies into relationship-based access control. The proposed attribute-aware ReBAC enhances access control capability and allows finer-grained controls that are not available in ReBAC. The policy specification language for the user-to-user relationship-based access control (UURAC) model proposed in [6] is extended to enable such attribute-aware access control. We also present an enhanced path-checking algorithm to determine the existence of the required attributes and relationships in order to grant access.

Keywords: Access Control, Attribute, Social Networks

1 Introduction

Authorization decisions in traditional access control models (e.g., discretionary access control, mandatory access control, role-based access control, etc.) are primarily based on identities, group or role memberships, and security labels, etc. However, they fail to cope with the scalability and dynamicity of online social networks (OSNs). In OSNs, it is not practical for users to specify all the users who can access their information in a traditional way. Instead, Relationship-based Access Control (ReBAC) [7, 9] has emerged as the most prevalent access control mechanism for OSNs. With ReBAC, resource owners can specify access control of their information based on their relationships with others, without knowing the user name space of the entire network or all their possible direct or indirect

* This work is partially supported by grant CNS-1111925 from the US National Science Foundation.

contacts. Accordingly, relationship-based access control has been recognized as a key requirement for security and privacy in OSNs [10], and has been commonly adopted in real world OSN systems since it keeps the balance between ease-of-use and flexibility.

Despite its popularity in both theory and practice, current ReBAC is still far from perfect. Most ReBAC systems merely focus on type, depth or strength of the relationships, lacking support for some topology-based and history-based access control policies that are of rich social significance. For example, they cannot express policies such as "at least five common friends" or "friendship request pending" that require global or contextual information of the social graph. In addition to relationships, attributes of users (such as age, location, identity) also need to be taken into account when determining access. Without introducing attributes, policies like "a common friend named Tom" cannot be described in current ReBAC languages. We suggest that combining these attributes of users and relationships with ReBAC would provide users more versatile and flexible access control on their data.

In this work, we discuss the benefits of incorporating attribute-based access control (ABAC) into an existing ReBAC model. We formalize a new policy specification based on the previous UURAC policy language [6], addressing the access requirements in terms of the attributes of users, relationships and social graphs. Several examples are provided to show the usage of the proposed attribute-aware ReBAC policy language. The path-checking algorithm for finding a qualified path in [6] is also extended to simultaneously check whether the attribute-based requirements are satisfied. We briefly compare the complexity of the algorithm with the original UURAC algorithm, and discuss the changes.

2 Background and Motivation

In this section, we review existing ReBAC literature followed by discussion on potential benefits of adding attribute-based policies to ReBAC.

2.1 ReBAC

Access control based on interpersonal relationships has become the de facto standard solution for OSNs in practice. This is called relationship-based access control (ReBAC) [10]. A number of ReBAC solutions have been proposed in the literature. From these solutions, we can identify at least three decision factors of relationship type, relationship depth, and relationship strength (e.g., trust) that are used in ReBAC.

Some solutions proposed to associate trust with ReBAC, allowing people to specify the strength of their connections by assigning trust values to relationships. Carminati et al. proposed a series of ReBAC models for OSNs, where trust level, type and depth of the user-to-user relationships are identified as decision factors for authorization [3, 4]. In their work, trust values of multiple relationships of the same type on a path can be calculated to form an indirect trust between users that are not directly connected. In [2], Carminati et al. introduced a semantic web based ReBAC solution, which defines authorization, administration and filtering policies to control the access. Another semantic web-based approach proposed by Masoumzadeh et al. allows both users and the system to express policies based on access control ontologies [15].

Fong et al. presented a Facebook-like access control model, featuring four types of policies that cover four different aspects of access in OSNs [8]. The four policies can regulate user search, traversal of the social graph, communication between users, and normal access to objects owned by users. The model mimics the privacy setting used in Facebook at the time of the paper was written. As such it lacks support for multiple types and depth of relationships. Fong et al. also built a formal ReBAC model for social computing applications on top of a modal logic language specification [7]. This model enables support for multiple types and direction of relationships. The model has been subsequently extended for improved flexibility and efficiency [1, 9].

Cheng et al. proposed a user-to-user relationship-based access control (UURAC) model with a regular expression-based policy specification language [6]. User-to-resource and resource-to-resource relationships are added to the model in their subsequent work on the URRAC model [5]. Both models are founded on the characteristics identified in Activity Control (ACON) [16, 17].

2.2 Beyond Relationships

ReBAC takes advantage of the structure of OSN systems and offers users a simple and effective way for configuring their access control policies. However, ReBAC suffers from two shortcomings. First, most of the ReBAC models rely on the type, depth, or strength of relationships, but cannot exploit more complicated topological information contained in the social graph. For example, many ReBAC proposals can determine whether there exists a qualified relationship path on the graph, but their policy languages cannot express requirements on multiple occurrences of such paths. Second, ReBAC generally lacks support for various contextual information of users and relationships available in OSNs. Such contextual

information also called attributes can be utilized for finer-grained access control. In addition to the normal relationship information, some of the attribute examples are user's name, age, role, location, trust in other users, duration of relationships, and so on. Let us consider two examples of attribute-based policies that are applied on ReBAC.

Common friends. The very nature of OSNs encourages new connections. To help users expand their connections, OSNs normally suggest some new connection candidates to a user based on number of common friends they share, for example. This is typically used as a tool for social promotion, but it can be applied to access control as well. Alice can allow a user who is not currently her friend but shares a certain number of common friends with her to access some of her contents. She can also specify that Bob must be a common friend of her and an access requester in order for the requester to get access. In regular ReBAC system, these policies cannot be expressed as they only check the existence of certain relationships but do not count the number of such relationships. Also, due to lack of support for node attributes, ReBAC policies are not able to distinguish particular users on the relationship paths.

Transitive trust. Consider a scenario where relationships are associated with an attribute of trust value, which denotes the strength of connection between two users. Since each user only knows a few direct friends, it is expected that more users are likely to be connected indirectly through their existing connections. Trust values between direct connections are used to compute the transitive trust, indicating the strength of such indirect relationships. A line of research has been addressing the area of trust in OSNs [11, 12, 14], where trust is combined with relationship type and depth as parameters to determine access. However, many limitations can be found in these works. Some only consider a single relationship type, others consider the case of multiple types but lack support for trust comparison and calculation between different types or paths. If we treat trust as an attribute of relationships, it can be mixed with other attributes of users and thus enable finer-grained access control policies, despite the multiple relationship types or paths.

3 Preliminaries

The UURAC model introduced by Cheng et al. [6] captures user-to-user relationships in OSNs for authorization purpose, and defines a regular expression-based policy specification language. We briefly summarize UURAC model as our proposed model is based on it.

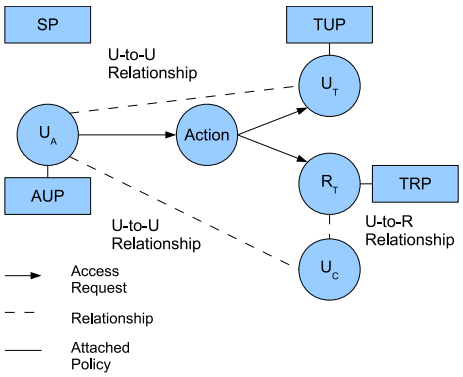

Fig. 1. Model components

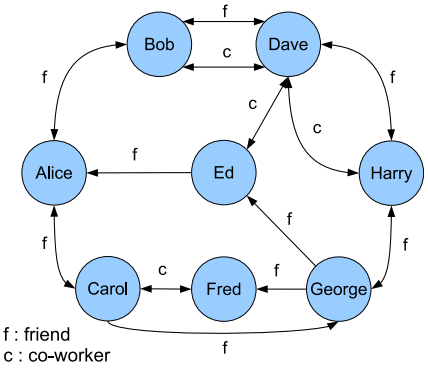

Fig. 2. A sample social graph

3.1 Basic Notations

In UURAC, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_n^{-1}\}$ denotes the set of relationship types. Given a relationship type $\sigma_i \in \Sigma$, the inverse of the relationship is $\sigma_i^{-1} \in \Sigma$. An action has an active form and a passive form, 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 UURAC Model Components

The UURAC model components are illustrated in Figure 1. **Accessing user** (u_a) represents an acting user who requests an access to a target and carries **accessing user policies** (AUP) which controls the accessing user's access to targets. Each **action** represents an operation initiated by an accessing user against a target. An action is denoted as *action* for the accessing user but $action^{-1}$ for the target. The targets can be either **target user** (u_t) (e.g., a user pokes another user) or **target resource** (r_t). There is also a **controlling user** (u_c) who controls an access to a resource that has user-to-resource (U2R) relationship with her such as the owner of the resource. An access to a target user is controlled based on the **target user policies** (TUP) that are configured using user-to-user (U2U) relationships between the target user and the access requesters, while an access to a target resource is controlled based on **target resource policies** (TRP) that are configured using U2U relationships between the controlling user and the access requester.

An access request $\langle u_a, act, target \rangle$ denotes the initiation of an access, where $act \in Act$ specifies the type of access requested by the accessing

user on the target. If u_a requests to interact with another user, *target* is $u_t \in U$. 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.

A policy defines the rules that determine how authorization is regulated. Policies can be either system-specified or user-specified. **System-specified policies** (SP) are system-wide rules enforced by the OSN system while user-specified policies are applied to specific users or resources. User-specified policies include AUP, TUP, and TRP.

3.3 Social Graph

As shown in Figure 2, OSN is abstracted as a directed labeled simple graph, where each node indicates a user and each edge corresponds to a U2U relationship. The social graph of an OSN is modeled as a triple $G = \langle U, E, \Sigma \rangle$ where

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

For every $\sigma_i \in \Sigma$, there is $\sigma_i^{-1} \in \Sigma$ representing the inverse of relationship type σ_i. We require that the original relationship and its inverse twin always exist on the social graph 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}).

4 $UURAC_A$ Model

In this section, we extend the UURAC model to facilitate attribute-aware ReBAC policy specification and enforcement.[1]

4.1 Attributes in OSNs

OSNs maintain a massive amount of data about attributes of users and resources. Users keep profile information as required by the OSNs, such as name, age, gender, etc. When a piece of resource is uploaded to the

[1] We reiterate that in UURAC only user-to-user relationships are considered so resources can only occur as the target of a relationship path. The relationship path itself can only include users.

OSN, the resource provider is also able to attach some metadata about the resource. We can define policies based on this attribute information associated with users and resources. However, the majority of ReBAC systems have focussed on some particular aspects of relationships, such as type, depth, and strength. This makes ReBAC relatively simple and efficient, but also limits the use of ReBAC in terms of control capability. In recent years, studies on attribute-based access control (ABAC) have shown that various contextual information of user, resource, and computing environment could be utilized for highly flexible and finer-grained controls [13, 18, 19]. However, current ABAC solutions are not likely to be readily usable on top of a ReBAC in OSNs. While typical ABAC models only consider the attributes of accessing user, target resource and sometimes computing environment, attribute-aware ReBAC needs to specify which attributes and whose attributes (i.e., user attributes, relationship attributes) on the relationship path between accessing users and target/controlling users should be examined.

For attribute-aware ReBAC, we identify three types of attributes: node (user/resource) attribute, edge (relationship) attribute and count attribute, as follows.

Node attributes. Users and resources are represented as nodes on the social graph. Users carry attributes that define their identities and characteristics, such as name, age, gender, etc. Resource attributes may include title, owner, date, etc.

Edge attributes. Each edge is associated with attributes that describe the characteristics of the edge. Such attributes may include relationship weights, types, and so on.

Both edge attributes and node attributes can apply to a single object or multiple objects. An example of attributes for multiple edges is the transitive trust between two nodes that are not directly connected. For instance, trust values of two or more edges need to be considered to calculate overall trust between accessing user and target/controlling user. Attributes describing multiple nodes are more commonly seen in OSNs, such as average age, common location, or common alma mater between people. Relevant node and edge attributes can be also assembled to enable policy combinations. For instance, Alice may specify a policy saying that "only users who have more than 0.5 trust with Bob can access."

Count attribute. Count attribute neither describes nor is associated with any node or edge. It depicts the occurrence requirement for the attribute-based path specification, specifying the lower bound of the occurrences of such path.

4.2 Attribute-based Policy Formulation

Attribute-based policy specifies access control requirements that are related to the attributes of users and their relationships. Here, we formally define the basic attribute-based policy language.

- N and E are nodes and edges, respectively;
- $NA_k (1 \leq k \leq K)$ and $EA_l (1 \leq l \leq L)$ are the pre-defined attributes for nodes and edges, respectively, where K is the number of node attributes and L is the number of edge attributes;
- $ATTR(n)$ and $ATTR(e)$ are attribute assignments for node n and edge e, respectively, where $ATTR(n) \in NA_1 \times NA_2 \times \cdots \times NA_K$, and $ATTR(e) \in EA_1 \times EA_2 \times \cdots \times EA_L$. Each attribute has only single value for its domain.

On the relationship path between two users in OSNs, there may exist many other users connected with different relationships. Each user or relationship carries attributes, which can be utilized for specifying access control rules. In some cases, the attributes of all users or relationships on the path need to be considered. Sometimes, attributes of only certain users or relationships are used. As shown in Table 1, we use the universal quantifier $\forall$ and the existential quantifier $\exists$ to denote "all" and "at least one" user(s) or relationship(s), respectively. The notation [ ] is used to represent ranges on the relationship path while { } denotes a set of users/relationships located at a specific distance on the path between accessing user and target/controlling user. In order to express a range or exact position on the path, we use plus and minus signs to indicate the forward (from the start) and backward directions (from the end), followed by a number that denotes the position from the front or the back. Note that indicator for users starts from 0 while indicator for relationships begins from 1. For example, for users, +0 means the starting user and -1 represents the second last user on the path; while for relationships, +1 indicates the first relationship on the path and -2 means the second last. The plus-minus sign in the last two rows denotes the forward or backward direction rather than its normal mathematical meaning.

Table 1. Attribute quantifiers

$\forall$ [+m, -n]	All entities from the m^{th} to the n^{th} last, $m+n \leq h$ where m and n are non-negative integers and h is a hopcount limit
$\forall$ [+m, +n]	All entities from the m^{th} to the n^{th}, $m \leq n \leq h$
$\forall$ [-m, -n]	All entities from the m^{th} last to the n^{th} last, $h \geq m \geq n$
$\exists$ [+m, -n]	One entity from the m^{th} to the n^{th} last, $m+n \leq h$
$\exists$ [+m, +n]	One entity from the m^{th} to the n^{th}, $m \leq n \leq h$
$\exists$ [-m, -n]	One entity from the m^{th} last to the n^{th} last, $h \geq m \geq n$
$\forall$ $\{2^{\{\pm N\}}\}$	All entities in this set
$\exists$ $\{2^{\{\pm N\}}\}$	One entity in this set

An attribute-base policy rule is composed of a $quantifier$ specifying the quantity of certain node/edge attributes, a boolean function of these node/edge attributes $f(ATTR(N), ATTR(E))$, and a count attribute predicate $count \geq i$, as follows.

$$\langle quantifier, f(ATTR(N), ATTR(E)), count \geq i \rangle$$

Note that the quantifier is applied to a node/edge function, but not to the count attribute predicate. For instance, R1 specifies a rule saying that "there must be at least five common connections between the requester and the owner, whose occupation is student". In R2 and R3, the count attribute predicate is not used and this is shown as '_', which indicates $count \geq 1$ in default. Here, the policy contains a rule indicating that "a user who is connected through adults whose addresses are 'Texas' can access". R3 requires that on a path between the accessing user and target/controlling user, users in three specific distances must be adults.

- $R1 : \langle \exists[+1, -1], occupation(u) = \text{“}student\text{”}, count \geq 5 \rangle$
- $R2 : \langle \forall[+1, -1], (age(u) \geq 18) \wedge (address(u) = \text{“}Texas\text{”}), \_ \rangle$
- $R3 : \langle \forall\{+1, +2, -1\}, (age(u) \geq 18), \_ \rangle$

4.3 Policy Specifications

Attribute-based policies are applied on certain relationship paths between accessing user and target/controlling user. For this, we extend the regular expression-based policy specification language proposed in [6]. Table 2 defines a list of notations used in the policy specification language.

Attribute-aware UURAC policies include two parts: a requested action, and a graph rule that conditions the access based on the social graph. As shown in Table 3, we identify several different types of policies.

Table 2. Policy specification notations

Concatenation ($\cdot$)	Joins multiple characters $\sigma \in \Sigma$ or Σ itself end-to-end, denoting a series of occurrences of relationship types.
Asterisk (*)	Represents the union of the concatenation of σ with itself zero or more times.
Plus (+)	Denotes concatenating σ one or more times.
Question Mark (?)	Represents occurrences of σ zero or one time.
Disjunctive Connective ($\vee$)	Indicates the disjunction of multiple path specs.
Conjunctive Connective ($\wedge$)	Denotes the conjunction of multiple path specs.
Negation ($\neg$)	Implies the absence of the specified pair of relationship type sequence and hopcount.
Colon(:)	Separates relationship pattern and attribute-based policies

Table 3. Access control policy representations

Accessing User Policy	$\langle act, graphrule \rangle$
Target User Policy	$\langle act^{-1}, graphrule \rangle$
Target Resource Policy	$\langle act^{-1}, u_c, graphrule \rangle$
System Policy for User	$\langle act, graphrule \rangle$
System Policy for Resource	$\langle act, (r.typename, r.typevalue), graphrule \rangle$

Actions are denoted in the passive form act^{-1} in target user policy and target resource policy, since target user/resource is always the recipient of the action. Target resource policy has an extra parameter u_c, indicating the controlling user of the resource. The differentiation of active and passive form of an action does not apply to system-specified policies, as these policies are not associated with any particular entity in action. However, when specifying a system policy for a resource, we can optionally refine the resource in terms of resource type $(r.typename, r.typevalue)$.

Table 4 defines the syntax for the graph rules using Backus-Naur Form (BNF). Each graph rule specifies a *startingnode* and a *pathrule*. Starting node denotes the user where the policy evaluation starts. A path rule represents a collection of path specs. Each path specification consists of a pair $(path, hopcount)$ that specifies the relationship path pattern between two users and the maximum number of edges on the path, which need to be satisfied in order to get access. Multiple path specifications can be connected with conjunctive "$\wedge$" and disjunctive "$\vee$" connectives. "$\neg$" over path specifications denotes absence of the specified pair of relationship pattern and hopcount limit. The pattern of relationship path *path* represents a sequence of type specifiers from the starting node to the evaluating node.

Unlike in UURAC, we add a new term $AttPolicy$ to the grammar to facilitate attribute-based policies. It can be found either after the whole path specification $(path, hopcount)$ or a segment of the path pattern *path*. The one that applies to $(path, hopcount)$ is called global attribute-based policy. When it follows a segment of *path*, it is a local attribute-based policy that only applicable for this segment. For simplicity, the examples hereafter only use global attribute-based policies.

We now show how attribute-based rules can be applied to some examples within UURAC$_A$.

Example 1: Node attribute and count attribute policy. Alice wants to reveal her profile to users who share at least five common student friends. She can specify the following policy for her friends of friends:

- P1: $\langle profile\_access, (u_a, ((ff, 2)\text{: } \exists[+1, -1], occupation(u) = \text{“}student\text{”}, count \geq 5))\rangle$

Table 4. Grammar for graph rules

$GraphRule \rightarrow$ "("$StartingNode$","$PathRule$")"
$PathRule \rightarrow AttPathSpecExp \mid AttPathSpecExp\ Connective\ PathRule$
$AttPathSpecExp \rightarrow PathSpecExp \mid PathSpecExp$" : "$AttPolicy$
$Connective \rightarrow \vee \mid \wedge$
$PathSpecExp \rightarrow PathSpec \mid$"$\neg$"$PathSpec$
$PathSpec \rightarrow$ "("$AttPath$","$HopCount$") "|"("$EmptySet$","$HopCount$")"
$HopCount \rightarrow Number$
$AttPath \rightarrow Path \mid Path$" : "$AttPolicy$
$Path \rightarrow TypeSeq \mid TypeSeq\ Path$
$EmptySet \rightarrow \emptyset$
$TypeSeq \rightarrow AttTypeExp \mid AttTypeExp$"·"$TypeSeq$
$AttTypeExp \rightarrow TypeExp \mid TypeExp$" : "$AttPolicy$
$TypeExp \rightarrow TypeSpecifier \mid TypeSpecifier\ Wildcard$
$AttPolicy \rightarrow$ use dedicated parser to process
$StartingNode \rightarrow u_a|u_t|u_c$
$TypeSpecifier \rightarrow \sigma_1|\sigma_2|\ldots|\sigma_n|\sigma_1^{-1}|\sigma_2^{-1}|\ldots|\sigma_n^{-1}|\Sigma$
$Wildcard \rightarrow$ " $*$ "|"?"|" $+$ "
$Number \rightarrow [0-9]+$

If she wants to allow someone who shares a common friend Bob with her to see her profile, the policy can be represented as follows:

– P2: $\langle profile\_access, (u_a, ((ff, 2)\text{: } \exists[+1, -1],\ name(u) =$ "Bob", _$))\rangle$

For P1, the system needs to find paths that match $(ff, 2)$ and check the occupation attribute of users on the paths. If there exist at least five such paths, u_a is allowed to see the profile information of the target. For P2, once a $(ff, 2)$ path is found and the name of the user on the path equals to Bob, the system would grant access.

Example 2: Edge attribute policy. Alice grants users to access $Photo1$ if the user is within 3 hops away and can reach her on a path with a minimum 0.5 trust value of friend relationships on each hop. Such policy is specified as follows:

– P3: $\langle read, Photo1, (u_a, ((f*, 3) : \forall[+1, -1], trust(r) \geq 0.5, \_))\rangle$

The system will check each edge on the path to ensure its trust value meets the requirement, before granting access.

Example 3: Capturing a UURAC policy. The following policy only contains relationship-based requirements $(f*, 3)$, where node/edge attributes and count attribute are both empty:

– P4: $\langle poke, (u_a, (f*, 3) : \exists[+0, -0], \_, \_))\rangle$

The UURAC$_A$ model is seamlessly compatible with the UURAC model. The example 3 shows how UURAC policy can be captured in UURAC$_A$.

Algorithm 1 *AccessEvaluation*$(u_a, act, target)$

(Policy Collecting Phase)
if $target = u_t$ **then**
    $AUP \leftarrow u_a$'s policy for act, $TUP \leftarrow u_t$'s policy for act^{-1}, $SP \leftarrow$ system's policy for act
else
    $u_c \leftarrow owner(r_t)$, $AUP \leftarrow u_a$'s policy for act, $TRP \leftarrow u_c$'s policy for act^{-1} on r_t, $SP \leftarrow$ system's policy for $act, r.type$
(Policy Evaluation Phase)
for all policy in AUP, TUP/TRP and SP **do**
    Extract graph rules (*start*, *path rule*) from policy
    for all graph rule extracted **do**
        Determine the starting node, specified by *start*, where the path evaluation starts
        Determine the evaluating node which is the other user involved in access
        Extract path rules *path rule* from graph rule
        Extract each path spec *path*, *hopcount* and/or attribute rule *attpolicy* from path rules
        Simultaneously path-check each path spec and evaluate the corresponding attribute rule using Algorithm 2
        Evaluate a combined result based on conjunctive or disjunctive connectives between path specs
Compose the final result from the result of each policy

5 Algorithm

This section addresses the access evaluation of UURAC$_A$. UURAC [6] provides a path-checking algorithm to find a qualified path between the access requester and the target (or the resource owner) that meets the ReBAC requirements. To enforce attribute-based policies, the access evaluation should incorporate attribute-based policies during path-checking. One may run attribute checking on the result paths found by the UURAC algorithm. However, this is likely to be inefficient. In this paper, we present a modified path-checking algorithm to incorporate an attribute-based policy evaluation on the fly during path finding process.

Access Evaluation Procedure Access requests can be evaluated as described in Algorithm 1. For an access request $(u_a, act, target)$, the system fetches u_a's policy about act, $target$'s act^{-1} policy and the system-specified policy for act. The decision module extracts path specification (*path*, *hopcount*) and attribute-based rules *attpolicy* from these policies. It runs the path-checking algorithm to determine the result for each policy. During path-checking, the decision module also needs to keep track of all of the involved attributes and make sure they satisfy the attribute-based policies. Finally, the results of all chosen policies in evaluation are composed into a single result. The existence of multi-user policies may raise policy conflicts. To resolve this, we can adopt the conflict resolution policy proposed in [5], which is based on a disjunctive, conjunctive, or prioritized strategy.

Algorithm 2 $DFSPathChecker(G, path, hopcount, s, t, globalattpol)$

$DFA \leftarrow REtoDFA(path)$; $currentPath \leftarrow NIL$; $d \leftarrow 0$
$stateHistory \leftarrow$ DFA starts at the initial state
Extract the quantifier symbol and interval/set information from *globalattpol*
Get the required rules for attributes of edges and nodes $f(ATTR(E), ATTR(N))$
Fetch the requirements of count attribute "$count \geq i$". If it is omitted, "$count \geq 1$".
Assign temporary space for attributes according to the size of the interval/set and the hopcount limit
Initialize counter $count \leftarrow 0$
if $hopcount \neq 0$ **then**
    return DFST(s)

Algorithm 3 $DFST(u)$

if $d + 1 > hopcount$ **then**
    return FALSE
else
    for all (v, σ) where (u, v, σ) *in* G **do**
        switch
        case 1 $v \in currentPath$
        break
        case 2 $v \notin currentPath$ and $v = t$ and DFA with transition σ is at accepting state
        if v and (u, v, σ) is within the range specified by quantifier **then**
            $attrList \leftarrow attrList.(ATTR(v), ATTR(u, v, \sigma))$
            if $f(ATTR(v), ATTR(u, v, \sigma)) = TRUE$ **then**
                $count \leftarrow count + 1$
                if $count \geq i$ **then**
                    $d \leftarrow d + 1; currentPath \leftarrow currentPath.(u, v, \sigma)$
                    $currentState \leftarrow$ DFA takes transition σ
                    $stateHistory \leftarrow stateHistory.(currentState)$
                    return TRUE
            else
                $attrList \leftarrow attrList \backslash (ATTR(v), ATTR(u, v, \sigma))$
        else
            $d \leftarrow d + 1; currentPath \leftarrow currentPath.(u, v, \sigma)$
            $currentState \leftarrow$ DFA takes transition σ
            $stateHistory \leftarrow stateHistory.(currentState)$
            return TRUE
        break
        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
        break
        case 4 $v \notin currentPath$ and $v = t$ and transition σ is invalid for DFA
        break
        case 5 $v \notin currentPath$ and $v \neq t$ and transition σ is invalid for DFA
        break
        case 6 $v \notin currentPath$ and $v \neq t$ and transition σ is valid for DFA
        $d \leftarrow d + 1; currentPath \leftarrow currentPath.(u, v, \sigma)$
        $currentState \leftarrow$ DFA takes transition σ
        $stateHistory \leftarrow stateHistory.(currentState)$
        if (DFST(v)) **then**
            return TRUE
        else
            $d \leftarrow d - 1; currentPath \leftarrow currentPath \backslash (u, v, \sigma)$
            $attrList \leftarrow attrList \backslash (ATTR(v), ATTR(u, v, \sigma)$
            $previousState \leftarrow$ last element in $stateHistory$
            DFA backs off the last taken transition σ to $previousState$
            $stateHistory \leftarrow stateHistory \backslash (previousState)$
    return FALSE

Attribute-aware Path Checking Algorithm The path-checking algorithm, as shown in Algorithm 2, uses a depth-first search (DFS) strategy to traverse the social graph G from a starting node s. The mission is to find relationship paths between the starting node s and the evaluating node t, that satisfy the policy. The pair of path pattern *path* and hopcount limit *hopcount* specifies the relationship-based requirements, whereas *globalattpol* indicates the attribute-based rules.

Let us consider the example policy P1 in Section 4.3: $\langle profile\_access$, $(u_a, ((ff, 2)$: $\exists[+1, -1]$, $occupation(u) =$ "*student*", $count \geq 5))\rangle$. The grammar extracts the starting node u_a and splits the relationship-based rules $(ff, 2)$ and the attribute-based rules "$\exists[+1, -1]$, $occupation(u) =$ "*student*", $count \geq 5$". Algorithm 2 then constructs a DFA (deterministic finite automata) from the regular expression ff. This is done by the function $REtoDFA()$. Variables $currentPath$ and $stateHistory$ are initialized to NIL and the initial DFA state, respectively. The attribute-based rule is divided into three parts: "$\exists[+1, -1]$", "$occupation(u) =$ '*student*'" and "$count \geq 5$". "$\exists[+1, -1]$" quantifies the whole path between the access requester and the target (or the resource owner) to which the following node attribute function applies. "$occupation(u) =$ '*student*'" is a function of node attributes that checks the occupation of the users on the path. The count attribute predicate "$count \geq 5$" specifies the required number of qualified relationship paths. To store the attribute values of nodes and edges during traversal in this example, we need space for attributes of 1 node and 2 edges. In general, if the interval is $[+a, -b]$ and the hopcount limit is c, we need to assign space for attributes of (c - a - b + 1) nodes and (c - a - b) edges.

After setting the hopcount indicator d to 0, Algorithm 2 launches the DFS traversal function $DFST()$, shown in Algorithm 3, from the starting node. Given the node u, the algorithm first makes sure taking one step forward does not violate the hopcount limit. Otherwise, it has to exit and return to the previous node. If further traversal is allowed, the algorithm starts to pick an edge (u, v, σ) from the collection of all incident edges leaving u one by one. According to the path pattern ff in the example, at the first step, the algorithm specifically looks for an unvisited edge of type f terminating at a node other than the evaluating node (case 6). If such edge is found, let's say (u_a, u_1, f), the algorithm increments d by 1, adds the edge to $currentPath$, moves the DFA from the initial state by taking transition f and updates the DFA state history accordingly. It also adds the corresponding attributes of edge (u_a, u_1, f) and node u_1 to the attribute list $attrList$ for later evaluation, since u_1 is 1 hop away from u_a and thus is within the range [+1, -1]. The algorithm then con-

tinues to run $DFST()$ on the new node u_1. From node u_1, it repeats the previous process again by checking the hopcount limit and picking new incident edges. Since the hopcount limit is 2, the algorithm has to find an unvisited edge of type f that terminates at t (case 2). Once the edge (u_1, t, f) is discovered, the algorithm goes on to find the corresponding attributes for evaluation. [+1, -1] indicates that we also need to check the attributes of the second last node on the path, which is u_1. Since we already added u_1's attributes to the list, the algorithm simply runs attribute function $f(ATTR(u_1))$ to see if it satisfies the requirements. If yes, we then check the count attribute, which is *count* in this case. The policy says it requires five qualified paths, thus the algorithm has to increment the counter and return to the previous node to search for another 4 paths. If $(u_a, u_1, f)(u_1, t, f)$ is the fifth path we found, $DFST(u_1)$ should return true and all its previous $DFST()$ calls as well. Eventually, it makes Algorithm 2 to return true, indicating we found the necessary amount of paths that satisfy the policy. If the node/edge attributes do not match the requirements, the algorithm removes the attributes from the list (line 18-19) and try the next edge. After finishing edge searching at this level and returning to the previous $DFST()$ call (line 38-43), it has to drop the edge and reset all variables to the previous values. Algorithm 2 returns false after all incident edges leaving u_a have been unsuccessfully searched.

The proof of correctness of this algorithm is fundamentally the same as the algorithm for UURAC [6]. The new algorithm neither brings in more edges to be considered nor increases the depth of recursive traversal to be taken. Hence, its complexity is still bounded between $O(dmin^{Hopcount})$ and $O(dmax^{Hopcount})$, where $dmin$ and $dmax$ stand for the minimum and maximum out-degree of node, and $Hopcount$ denotes the hopcount limit. Attribute-base check introduces additional overhead when the algorithm finds a possible qualified path. The overhead costs are proportional to the amount of attributes as well as the type of attribute functions considered in the policy, which is not related to the structure of the social graph.

6 Conclusion

This paper presents an extended UURAC model for OSNs that utilizes both relationship-based and attribute-based policies for determining access. Attribute information of users and their relationships are as important as the social graph in OSNs with respect to access control. We formalized the attribute-based policies and extended the grammar for policy specifications. The policy language supports expressing requirements on attributes of some or all of the users and relationships on the

path. While it could be possible to further extend the proposed model for even finer-grained attribute-based controls, the proposed model provides a solid foundational mechanism for ReBAC that also allows attribute-based access control.

References

1. G. Bruns, P. W. Fong, I. Siahaan, and M. Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *Proceedings of the second CODASPY*, pages 117–124. ACM, 2012.
2. B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. A semantic web based framework for social network access control. In *Proceedings of the 14th SACMAT*, pages 177–186. ACM, 2009.
3. B. Carminati, E. Ferrari, and A. Perego. Rule-based access control for social networks. In *OTM 2006 Workshops*, pages 1734–1744. Springer, 2006.
4. B. Carminati, E. Ferrari, and A. Perego. Enforcing access control in web-based social networks. *ACM TISSEC*, 13(1):6, 2009.
5. Y. Cheng, J. Park, and R. Sandhu. Relationship-based access control for online social networks: beyond user-to-user relationships. In *PASSAT 2012*, pages 646–655. IEEE, 2012.
6. Y. Cheng, J. Park, and R. Sandhu. A user-to-user relationship-based access control model for online social networks. In *DBSec 2012*, pages 8–24. Springer, 2012.
7. P. W. Fong. Relationship-based access control: protection model and policy language. In *Proceedings of the first CODASPY*, pages 191–202. ACM, 2011.
8. P. W. Fong, M. Anwar, and Z. Zhao. A privacy preservation model for facebook-style social network systems. In *ESORICS 2009*, pages 303–320. Springer, 2009.
9. P. W. Fong and I. Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th SACMAT*, pages 51–60. ACM, 2011.
10. C. Gates. Access control requirements for Web 2.0 security and privacy. *IEEE Web 2.0*, 2007.
11. J. Golbeck and J. Hendler. Inferring binary trust relationships in web-based social networks. *ACM Transactions on Internet Technology (TOIT)*, 6(4):497–529, 2006.
12. J. A. Golbeck. *Computing and Applying Trust in Web-based Social Networks*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2005.
13. X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *DBSec 2012*, pages 41–55. Springer, 2012.
14. S. R. Kruk, S. Grzonkowski, A. Gzella, T. Woroniecki, and H.-C. Choi. D-FOAF: distributed identity management with access rights delegation. In *The Semantic Web–ASWC 2006*, pages 140–154. Springer, 2006.
15. A. Masoumzadeh and J. Joshi. OSNAC: an ontology-based access control model for social networking systems. In *SocialCom 2010*, pages 751–759. IEEE, 2010.
16. J. Park, R. Sandhu, and Y. Cheng. ACON: activity-centric access control for social computing. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 242–247. IEEE, 2011.
17. J. Park, R. Sandhu, and Y. Cheng. A user-activity-centric framework for access control in online social networks. *Internet Computing, IEEE*, 15(5):62–65, 2011.
18. H. Shen and F. Hong. An attribute-based access control model for web services. In *PDCAT 2006*, pages 74–79. IEEE, 2006.
19. E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *Proceedings of the IEEE ICWS*, pages 561–569. IEEE, 2005.