This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

An Access Control Model for Online Social Networks Using User-to-User Relationships

Yuan Cheng, Jaehong Park, and Ravi Sandhu

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 enabled to specify which access can be granted in terms of existing relationships. In this paper, 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. Access control policies on users and resources are composed in terms of requested action, multiple relationship types, the starting point of the evaluation, and the number of hops on the path. We present two path checking algorithms to determine whether the required relationship path between users for a given access request exists. We validate the feasibility of our approach by implementing a prototype system and evaluating the performance of these two algorithms.

Index Terms—Social network, access control, security model, policy specification

1 INTRODUCTION

Online social networks (OSNs) have become ubiquitous in daily life and have tremendously changed how people connect, interact and share information with each other. Users share an enormous amount of content with other users in OSNs for a variety of purposes. The sharing and communications are based on social connections among users, namely relationships. Since most users join OSNs to keep in touch with people they already know, they often share a large amount of sensitive or private information about themselves. Given the rising popularity of OSNs and the explosive growth of information shared on them, OSN users are exposed to potential threats to security and privacy of their data. Security and privacy incidents in OSNs have increasingly gained attention from both media and research community [3], [21]. These incidents highlight the need for effective access control that can protect data from unauthorized access in OSNs.

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 expect 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 content, 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 collective control decision.

In OSN, access to resources is typically controlled based on the relationships between the accessing user and the controlling user of the target found on the social graph. This type of relationship-based access control (to which we refer as ReBAC) [22] 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.

Most existing OSN systems enforce a rudimentary and limited relationship-based access control mechanism, offering users the ability to choose from a predefined policy vocabulary, such as "public", "private", "friend" or "friend of friend". Google+ and Facebook introduced customized relationships, namely "circle" and "friend list", providing users richer options to differentiate distinctly privileged user groups. Meanwhile, researchers have proposed more advanced relationship-based access control models, such as [5], [8]–[12], [18]–[20], [28]. Policies in [5], [8]–[12], [18], [20] can be composed of multiple types of relationships. [10]–[12] also adopt the depth and the trust value of relationship to control the spread of informa-

- *Y. Cheng, J. Park (**Corresponding author**) and R. Sandhu are with the University of Texas at San Antonio, USA; E-mail: {yuan.cheng, jae.park, ravi.sandhu}@utsa.edu.*
- *A preliminary version of this article was presented at DBSec12 [14].*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

tion. Although only having the "friend" relationship type, [19] 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, allowing users the ability to express more sophisticated and fine-grained access control policies in terms of type pattern and depth of relationships among users in the network. Type pattern captures the pattern of relationship types along the relationship path from the accessing user to the target user. We adopt a regular expression-based approach for policy specification. Sequence of characters and quantification notations are employed to denote relationship paths, which express indirect relationships among users, such as $f*$, $f+$, $cf?$, etc. The use of regular expression and multiple relationship types gives the policy language the ability to specify more succinct policies than previous models did. To the best of our knowledge, this is the first relationship-based access control model for OSNs with such capability.

The rest of this paper is organized as follows. Section 2 provides motivation and context for our work, discusses related work, and identifies our contributions. In section 3, we present the fundamental structure of our UURAC model. A policy language for expressing access control policies is articulated in section 4. In section 5, we introduce path checking algorithms to evaluate a given access control policy. Section 6 describes prototype implementation and experimental results. Section 7 concludes the paper and outlines some future work.

2 MOTIVATION

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

2.1 Characteristics of Access Control for OSNs

OSN is becoming the most prevalent manifestation of user-generated content platforms. Photos, videos, blogs, web links and other kinds of information are posted, shared and commented by OSN users. Various types of user interactions, including chatting, private messaging, poking, social games, etc., are also embedded into these systems. Below, we discuss some essential characteristics [31], [32] 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 Policies for Outgoing and Incoming Actions. Notification of a particular friend's 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 all violent content or a parent may not want her child to invite her co-worker 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. Typically, the number of users in an OSN is very large and the amount of resources they own is usually even larger. Moreover, the relationships among users are changing frequently and dynamically. A user may not be able to know either the user name space of the entire network or all her possible direct or indirect contacts. Therefore, it is infeasible for her to specify access control policies for all of the possible accessing users. Even if she knows them all, it takes enormous amount of time for her to explicitly specify policies for all of them one by one as in discretionary access control. Role-based access control does not fit well in this situation either, because privileged user groups are different for each user. Thus different users' privileged user groups cannot be assigned to a unified set of roles. Overall using traditional access control approaches is cumbersome and inadequate for OSN systems.

Instead, 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

The large and complex collections of user data in OSNs require usable and fine-grained access control solutions to protect them [22], [23]. Gates [22]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

discusses the access control requirements for OSN environments, where she argues that one of the key requirements is relationship-based access control.

A formal model for access control in Facebook-like systems was developed by Fong et al [19], which treats access control as a two-stage process, namely, reaching the search listing of the resource owner and accessing the resource, respectively. Reachability of the search listings is a necessary condition for access. Although lacking support for directed relationships, multiple relationship types and trust metric of relationships, this model allows expression of arbitrary topology-based properties, such as "k-common friends" and "k-clique", which are beyond what Facebook and other commercial OSNs offer.

In [18], Fong proposed a formal model for social computing applications, in which authorization decisions are based on 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 expressive power [20]. In contrast to [19], 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 [5] later improved [18], [20] by using hybrid logic to enable better efficiency in policy evaluation and greater flexibility of atomic formulas. [5], [20] also support policies such as "k-common friends" and "k-clique" in addition to path policies.

Inspired by research in trust and reputation systems, some early solutions proposed by Kruk et al [28] and Carminati et al [10], [11] identified aggregated trust value, denoting the level of relationship, along with relationship type and depth on a path from the resource owner to the accessing user as parameters for authorization. While Kruk's work only considers one relationship type, Carminati's work allows multiple relationship types but only supports trust computation of a relationship path of a single type at a time. Carminati et al also proposed a semi-decentralized architecture, where access rules are specified in terms of relationship type, depth and trust metrics by individual users in a discretionary way [12]. The system features a centralized certificate authority to assert the validity of relationship paths, while access control enforcement is running on the decentralized user side.

In [8], [9], an access control model for OSNs utilizes semantic web technologies. Unlike 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 [8], [9], Masoumzadeh et al [29] 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.

It is worth noting that Crampton et al [17] recently proposed a variant of ReBAC model for applications beyond OSNs that specifies policies in terms of path conditions, which are similar to regular expressions.

Several works resort to attribute information to address access control for OSNs. Persona [2], EASiER [27] and the DBRA framework [4] are three representatives of attribute-based encryption (ABE) schemes for protecting shared data in OSNs. Basically, users define relationships or groups by assigning attributes to them, and resources are encrypted with attribute-based policies. Keys are distributed to groups or relationships so that only users with necessary attributes will be able to decrypt the data. In addition to attributes, [4] also allows constraints on distance between users on the social graph. These ABE schemes usually work as a third-party application and require shared content to be stored in an encrypted form, which consequently limits the functionality of OSNs. Key distribution and revocation is also an issue in practice, since relationships between users in OSNs are changing dynamically.

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.

All the models deal only with U2U relationships, except [8], [9] also recognize U2R (user-to-resource) relationships explicitly. U2R relationships can be captured implicitly via U2U with the last hop being U2R. While we believe that explicit treatment of U2R and R2R (resource-to-resource) relationships is important, this is beyond the scope of this paper. Fong et al's [5], [19], [20] allow users to express policies such as "k-common friends" and "k-clique". While the proposed model in this paper only permits specification of paths, the model can be extended to capture this type of policies by utilizing attribute information of users and relationships as shown in [15].

In terms of expressive power, the regular expression path policy with hopcount proposed in this work is equal to the above logic based approaches. However, it is relatively easier and more efficient to use. A single regular expression path pattern can express multiple paths without enumerating every possible path. For instance, $(f*, 3)$ can cover three enumeration paths f, ff, and fff. Details about the policy specifications are provided later in the paper.

3 UURAC Model Foundation

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

TABLE 1: Comparison of Access Control Models for OSNs

	Fong [19]	Fong [5], [18], [20]	Carminati [12]	Carminati [8], [9]	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 n	0 to n	1 to n	1 to n	0 to n
Relationship Composition	f, f of f; common friends, clique	exact type sequence; common friends, clique (except [18])	path of same type	exact type sequence	exact type sequence, path pattern of different types

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 the 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

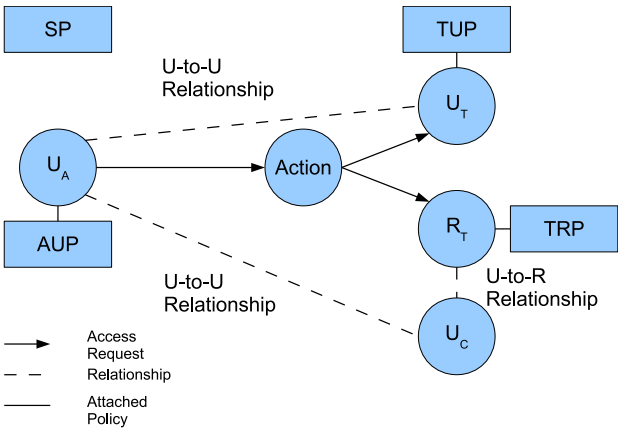


Fig. 1: Model Components

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

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.

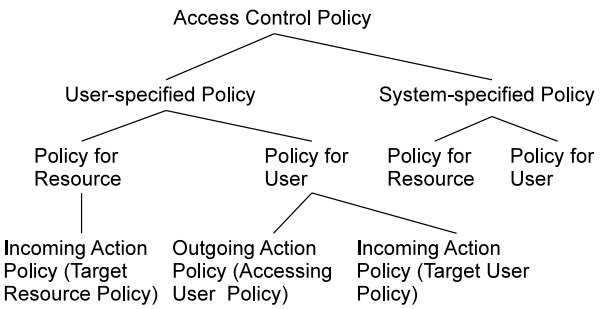


Fig. 2: Access Control Policy Taxonomy

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

policies include policies for resources and policies for users. Policies for resources are used to specify who can access the resources, 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.

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 $G =< U, E, \Sigma >$:

- 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.

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.

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.

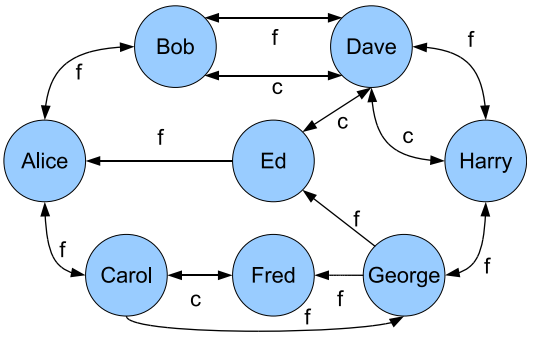


Fig. 3: A Sample Social Graph

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 the 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 quantification 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. The asterisk is commonly known as the Kleene star.

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* denotes 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. Policies defined by different users for the same action against same target are considered as separate policies. The *path rule* in each policy is composed of one or more *path specs*, in which multiple *path specs* are 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 co-worker, contains disjunction of three different pieces of path specs, of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

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.

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 this 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", which is the only allowed case for empty $path$. $Quantifier$ captures the three quantification characters, which facilitate specifying path expressions more efficiently and effectively. Given a graph rule from the access control policy, this grammar specifies how to parse the expression 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 resource types (see Table 3).

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 u_c, representing the controlling user of the resource.

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 specify policies for the corresponding target resource. To access the resource, the accessing user must have the required relationships with the controlling users. The policies associated with the target resources are defined on the basis of per action per controlling user. For instance, when querying $read$ access request on r_t, all of r_t's target resource policies need to be considered in evaluation. Each policy specifies a controlling user, with whom the accessing user must have the required relationship. 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* has the same format as accessing user policy. 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 ($r.typename, r.typevalue$).[2] Examples of types are ($filetype, photo$), ($filetype, statusupdate$), ($location, Texas$), etc. Thus, $<read$, ($filetype$, $photo$), (u_c, $f*$, 4)$>$ is a system policy applied to all $read$ access to photos in the system. When dealing with system policy for resources, we can determine the controlling user of the resource through some U2R relationships, such as ownership (as shown in Figure 1).

4.4 Access Evaluation Procedure

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

```
(Policy Collecting Phase)
if target = u_t then
    AUP ← u_a's policy for action, TUP ← u_t's policy for action^-1, SP ← system's policy for action
else
    AUP ← u_a's policy for action, TRP ← r_t's policy for action^-1, SP ← system's policy for action, (r.typename, r.typevalue)
(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 from path rule
        Path-check each path spec using Algorithm 2
        Evaluate the combined result based on conjunctive or disjunctive connectives between path specs and negation on individual path specs
Compose the final result from the result of each policy
```

2. There could be combinations of multiple resource types in one policy, but for illustration, we only show one resource type per policy.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

TABLE 2: Grammar for graph rules

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

TABLE 3: Access Control Policy Representations

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

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 retrieve all the corresponding policies of r_t. 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}: $<$ $poke$, $(u_a, (f*, 3))>$ $<$ $poke^{-1}$, $(u_t, (f, 1))>$ $<$ $read$, $(u_a, (\Sigma*, 5))>$
- Harry's policy P_{Harry}: $<$ $poke$, $(u_a, (cf*, 5) \vee (f*, 5))>$ $<$ $poke^{-1}$, $(u_t, (f*, 2))>$
- Policy of file2 P_{file2}: $<$ $read^{-1}$, $Harry$, $(u_c, \neg(p+, 2)>$
- System's policy P_{Sys}: $<$ $poke$, $(u_a, (\Sigma*, 5))>$ $<$ $read$, $(filetype, photo)$, $(u_a, (\Sigma*, 5))>$

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}, $<$ $read^{-1}$, $Harry$, $(u_c, \neg(p+, 2)>$ from P_{file2}, 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 $path\ rule$). 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

4.5.1 Policy Conflict Resolution

In OSN systems, if multiple users are allowed to specify their own policies on a same object or user, policy conflicts become inevitable. There are substantial prior works on conflict resolution of access control policies, especially in distributed systems, database systems and collaborative environments. Most conflicts discussed in these works are conflicts between positive and negative authorizations (permission vs. prohibitions) typically arising due to generality or specificity of the applicable policy in a hierarchy. However, in OSNs possible policy conflicts arise as policies specified by distinct users may carry contrasting authorizations.

With regards to multi-user policy conflicts in OSNs, there are several interesting proposals as well. [34] leveraged a game theoretic approach to address collective policy management in OSNs. [24] formulated a multi-party access control model for OSNs that measures the tradeoff between privacy and sharing with a policy conflict resolution mechanism based on voting scheme. [24] has been extended to express other threshold-based and strategy-based conflict resolution in [26]. Similar idea can be found in another piece of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

their work [25], where conflict detection is also addressed in addition to conflict resolution. Carminati et al introduced collaborative security policies to express privacy concerns from multiple users, which explicitly state either of the three strategies, namely "All, One, and Majority", to reach a collaborative decision [6].

In the proposed work, we consider three simple and intuitive approaches to resolve conflicts: disjunctive, conjunctive or prioritized. 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 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 unambiguous system level policies are available to resolve conflicts in user-specified authorization policies and do not consider user-specified conflict resolution policies.

4.5.2 Syntax

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 $action$ is done by u_a while $action^{-1}$ is from u_t's perspective. 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 co-workers to know about it. Note that the two distinct paths denoted by $fffc$ and fc may co-exist between a pair of users. Simply specifying $fffc$ in the policy does not avoid someone who also has fc relationship with the owner from accessing the resume. In contrast, $fffc \wedge \neg(fc)$ allows the co-workers of the user's distant friends to see the resume, while the co-workers 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. 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, but it is still useful to restrict future addition of positive policies. Nonetheless it is possible for the co-worker of a direct friend to have a second persona that meets the criteria for co-worker of a distant friend and thereby acquires access to the resume. Without strong identities we can only provide persona level control in such policies.

The inclusion of conjunction and negation in grammar may add extra costs in processing, but it empowers users to define finer-grained or more strict policies. The above example path rule $fffc \wedge \neg(fc)$ shows the utility of the two notations. However, this is largely a design decision and we will let users decide how to use them efficiently in their implementation. If only disjunction exists in a path rule, path specs with the same hopcount can be composed into a single regular expression prior to evaluation to improve performance.

5 ALGORITHMS

In this section, we present two algorithms for determining if there exists a qualified path between two involved users in an access request, based on depth-first search (DFS) and breadth-first search (BFS) strategies. Then, we provide a complexity analysis for both algorithms.

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 $< start, path\ rule >$.

As shown in Algorithm 2 and 4, the path checking algorithm 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

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 an NFA (non-deterministic finite automata) then to a DFA, by using the well-known Thompson's Algorithm [35] and Subset Construction Algorithm (also known as Büchi's Algorithm) [33], respectively.

5.1 Depth-first Search

Using DFS to traverse the graph 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 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 take a long traversal to find a target node, even if the node is close to the starting node. If the hopcount is unlimited, a DFS traversal may pursue a lengthy useless exploration. However, as activities in OSNs typically occur among people with close relationships, DFS with limited hopcount can minimize such unnecessary traversals.

In Algorithm 2, 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 the current traversal depth d to 0 and launches the DFS traversal function $DFST()$ in Algorithm 3 from the starting node s.

In Algorithm 3, 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 (line 2) and goes back to the previous node (line 24). 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 six 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 the current for loop, and finds the next unchecked edges of u.

When v is not on $currentPath$, we check if the transition σ belongs to the set of valid transitions for DFA. If the transition is invalid for DFA, we try the next edge (case 2). If the transition is valid and v is the target node t, there are two cases depending on whether taking transition σ reaches an accepting state. If it reaches an accepting state, we find a path between s and t matching the pattern $Path$ (case 3). We increment d by one, concatenate edge (u, v, σ) to $currentPath$, and save the current DFA state to history. If it does not, we break out of the for loop and continue to check the next unchecked edge of u (case 4). If the transition is valid but v is not the target node t, 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 (case 5). If the recursive function call discovers a matching path, the previous call also returns true. Otherwise, the algorithm has to step back to the previous node of u, reset all variables to the previous values, and check the next edge of node u. However, if d = 0, all the outgoing edges of the starting node are checked, thus the whole execution completes without a matching path.

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

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 transition σ is invalid for DFA
9: break
10: **case 3** $v \notin currentPath$ and $v = t$ and DFA with transition σ is at accepting state
11: $d \leftarrow d + 1$; $currentPath \leftarrow currentPath.(u, v, \sigma)$
12: $currentState \leftarrow$ DFA takes transition σ
13: $stateHistory \leftarrow stateHistory.(currentState)$
14: **return** TRUE
15: **case 4** $v \notin currentPath$ and $v = t$ and transition σ is valid for DFA but DFA with transition σ is not at accepting state
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: $d \leftarrow d-1$; $currentPath \leftarrow currentPath \backslash (u, v, \sigma)$
25: $previousState \leftarrow$ last element in $stateHistory$
26: DFA backs off the last taken transition σ to $previousState$
27: $stateHistory \leftarrow stateHistory \backslash (previousState)$
28: **return** FALSE

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

5.2 Breadth-first Search

Starting from an initial node, a BFS traversal aims to expand and examine all nodes of a graph from inside out until it finds the goal. A FIFO (first in, first out) queue is created with the starting node as the first element. All the nodes of a level need to be added to the queue, and will be dequeued before the nodes of their child level. Similar to the DFS traversal, we need to create a running DFA and set up the corresponding variables for the search. However, to find a matching path, a BFS traversal has to maintain the DFA state and other variables for every possible path it examines, resulting in a multiple number of DFAs and variables simultaneously. Although BFS may naturally consume more computational resources, it has advantage over its DFS counterpart as it never wastes time on a lengthy unsuccessful exploration.

As shown in Algorithm 4, we create a DFA from the regular expression pattern, enqueue the starting node s, and initialize the variable $currentPath$, $stateHistory$ and d of s to NIL, the initial DFA state and 0, respectively. The algorithm continues when the queue is not empty, and dequeues the first node of the queue for further exploration. Given a node q, if $d+1$ does not exceed the hopcount limit, the algorithm moves on to examine the incident outgoing edges of q. All edges can be classified into the same five cases as in the abovementioned DFS algorithm. For an edge (u, v, σ), only when v is not on $currentPath$ and v is the target node t and DFA taking a valid transition σ reaches an accepting state, we find a path between q and t matching the pattern $Path$ (case 3). We then update the corresponding variables for node v and exit the algorithm with true. If v is not on $currentPath$ and is not the target node, we check the validity of the transition σ. If the transition is valid, we will take it, update the variables of v, and enqueue node v into the queue for later examination (case 5). In all other cases, a successful exploration will not possibly occur, thus the edges are dropped. After checking all edges within the hopcount limit, the algorithm terminates with false if no matching path is found.

Algorithm 4 $BFSPathChecker(G, path, hopcount, s, t)$

```
1: DFA ← REtoDFA(path)
2: if hopcount ≠ 0 then
3:    create queue Q
4:    create node s: s.DFA ← DFA; s.currentPath ← NIL; s.d ← 0; s.stateHistory ← DFA starts at the initial state
5:    enqueue s onto Q
6:    while Q is not empty do
7:       dequeue a node from Q into q
8:       if q.d + 1 > hopcount then
9:          break
10:      else
11:         for all (v, σ) where (q, v, σ) in G do
12:            switch
13:            case 1 v ∈ currentPath
14:            break
15:            case 2 v ∉ currentPath and transition σ is invalid for DFA
16:            break
17:            case 3 v ∉ currentPath and v = t and DFA with transition σ is at accepting state
18:            create node v (clone from q)
19:            v.previousState ← v.currentState
20:            v.currentState ← DFA takes transition σ
21:            v.d++
22:            v.currentPath adds (q, v, σ)
23:            v.stateHistory adds currentState
24:            return TRUE
25:            case 4 v ∉ currentPath and v = t and transition σ is valid for DFA but DFA with transition σ is not at accepting state
26:            break
27:            case 5 v ∉ currentPath and v ≠ t and transition σ is valid for DFA
28:            create node v (clone from q)
29:            enqueue v onto Q
30:            v.previousState ← v.currentState
31:            v.currentState ← DFA takes transition σ
32:            v.d++
33:            v.currentPath adds (q, v, σ)
34:            v.stateHistory adds currentState
35:   return FALSE
```

5.3 Iterative Deepening Search

With hopcount, the DFS algorithm becomes a depth limited search. Hence, it avoids drawbacks in classical DFS regarding completeness. Iterative deepening search (IDS) algorithm executes depth limited search multiple times thus yields a worse result than our hopcount-enabled DFS algorithm. For this reason, we do not consider IDS further in this paper.

5.4 Proof of Correctness

The two algorithms are based on the classical DFS and BFS algorithms with a specific goal of finding qualified paths between nodes within a given hopcount limit. To establish the correctness, we need to prove from two aspects: (1) the algorithms will halt with true or false, and (2) if the algorithms return 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 $L(Path)$; if the algorithms return false, there is no simple path p of length less than or equal to $Hopcount$ such that the string representing p belongs to $L(Path)$.

All edges are classified into five categories using four rules: (1) is the current node on current traversed path, (2) is the transition σ valid, (3) is the edge's destination the target node, and (4) does taking transition σ reach an accepting state. Only edges that fall into case 3 indicate that a qualified path is found, and only edges that belong to case 5 require the algorithm to take one step further. The for loop guarantees edge will be visited once and only once, if a qualified path has been found yet. Rule (1) avoids cycles in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

traversal, and hopcount limit provides a cutoff to halt the algorithm. Other than that, the two algorithms are identical with the classical algorithms. Thus, we can use induction to prove the above properties easily.

5.5 Complexity Analysis

In the algorithms, 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 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 BFS algorithm and the recursive DFST() call terminate as soon as either a matching path is found or the hopcount limit is reached.

6 Implementation and Evaluation

In this section, we present some of the results obtained from our performance studies on the two path-checking algorithms. We implemented the algorithms in Java, and designed two sets of experiments to test the runtime execution of an access request evaluation using both algorithms. We deployed an access control decider with BFS and DFS path checkers on a virtual machine instance of an Ubuntu 12.04 image with 4GB memory and a 2.53 GHz quad-core CPU. The social graphs to be tested are stored in MySQL databases on the testing machine along with the sample access control policies. We designed sample policies and access requests that would require the access control decider to gather necessary information and crawl on the graph for access decisions. We then measured the time the algorithms take to complete a path checking over the graph and return a result to the decider.

6.1 Datasets

When designing the experiments, we consider two parameters of the graphs: hopcount (depth) and degree (width). Although the total number of nodes in the system may influence the performance and scalability of many graph systems, in our system the algorithms are not to explore the whole graph but the paths with limited hops stemming from one node. Therefore, the total number of nodes is not significant with respect to the performance. In fact, it is the hopcount limit and the number of edges to be explored at each hop that contribute most to the size of the problem, and hence the performance of our system.

A significant issue in this evaluation consists in the selection of representative datasets. There are some public available datasets collected from real-world OSN systems with large amount of real data. However, most of them only consider single relationship type or do not support relationship type at all. In a related analysis [7], the authors modified the original datasets to add type information, where relationship types are uniformly distributed. However, manually adding type information to the real datasets may not reflect the actual user behaviors, and thus ruins the integrity of the datasets and diminishes the value of having real data. Moreover, different real datasets possess various properties, making them incomparable with each other. Hence, synthetic data becomes an alternative for us, where we can configure different social graphs under our control, and analyze some specific properties of these graphs. To generate synthetic social graphs, we use neither the G(n, p) nor the G(n,m) variation of the Erdos-Renyi model, because both of them create graphs in which each node may have different number of edges. Instead, since our experiment is focused on the comparison on density, we set the outgoing degree of each node to a fixed number in each graph. The selection of the destination of each edge is random.

In the first set of experiments, we examine the performance of the BFS and DFS algorithms with respect to policies with different hopcount limit. In particular, we set the parameters to 1000 users and single relationship type for this set of experiments. Each user has the same number of neighbors, who are randomly selected among the rest 999 users. Two different kinds of path patterns, including enumeration and *-pattern, are used in the policies to investigate the impact of hopcount limit on the performance of the algorithms.

In the second set of experiments, we aim to study the performance of the algorithms against various number of edges that need to be traversed (i.e., the average degree of nodes in the graph) to show the scalability of our approach against dense graph. We keep the same 1000 users as in the previous experiments, but enable two types of relationships, namely "f(riend)" and "c(oworker)", and randomly assign each relationship between users with one of these types. The number of neighbors for each user is set in the quantities of 100, 200, 500 and 1000. Consider the fact that there are only two types of relationship and the social graph in reality is usually a sparse graph, 1000 neighbors for each of 1000 users makes a relatively "dense" social graph for evaluation. We then run different policies on these four graphs to compare their differences.

Given an access control policy, we randomly pick 1000 different pairs of requester and target nodes from

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing

the graph, and run each algorithm 5 times on these 1000 pairs of nodes. Each measurement is the average results of these 5000 runs. To make fair comparison between true and false cases, we design different policies to get 5000 true cases and 5000 false cases. To evenly compare between true cases of different settings, we scale the number of selected users so that we can get results from the same amount of true cases.

6.2 Results

Figure 4 illustrates the results of the first set of experiments. We compare the BFS and DFS algorithms using policies with different hopcount limits in both the true-case and false-case scenarios. For true cases of *-pattern paths, Figure 4 (a) shows how the average running time changes with respect to increase in hopcount limit. To make a more comprehensive comparison, in this particular test, we apply the following values 10, 50 and 200 (which is close to 190, the average number of friends claimed by Facebook [37]) to the number of neighbors for each user. *-pattern paths are known to be more flexible than enumeration paths in path-checking. In fact, the results for *-pattern record the time elapse of finding one of the shortest qualified path. As we expected, when hopcount increments, the average execution time required for both algorithms increases as well, but the trends tend to flatten after the hopcount reaches 4. It indicates that a qualified path can be always found between two users within 4 hops in this setting. A probability calculation also verifies this finding. In the case of 10 neighbors per user, the aggregate probability of finding a qualified path is 1%, 10.5%, 67.3% for the first three hops, respectively, and eventually 100% at the fourth hop. The probability reaches 100% within 3 hops in the other two denser graphs. We also find that the BFS algorithm works slightly better than the DFS algorithm for large hopcount limit in sparse graphs, as DFS takes many lengthy probes before finding a qualified path while BFS does not suffer from much overhead in sparse graphs.

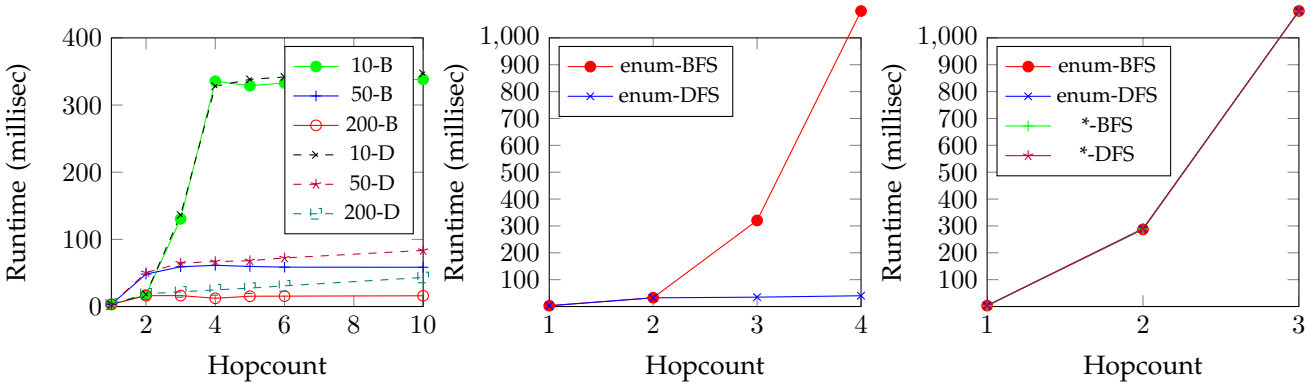
According to the classic idea of "six degrees of separation" and the findings of "small world experiment" [30], [36], any pair of people are distanced by no more than six intermediate connections on average. A recent study by Backstrom et al [1] further indicates that the average distance on the current social graph of Facebook is smaller than the commonly cited six degrees, and has shrunk to 4.74 as Facebook grows. Based on these findings, for true cases of enumeration paths, we restrain the hopcount limit up to 4, as our dataset is relatively much smaller than Facebook. As shown in Figure 4 (b), when hopcount limit increments, the time cost by the BFS algorithm increases significantly, due to the fact that it will not take the next hop without finishing search on all edges at the current level; whereas a greater hopcount does not worsen the performance of the DFS algorithm much.

Figure 4 (c) demonstrates the comparison between the two algorithms in false-case scenarios. The false-case scenarios actually represent the worst case scenario for path-checking, where both algorithms need to exhaustively search all possible paths within the hopcount limit from the starting node. Therefore, the two algorithms perform similarly in both enumeration and *-pattern settings. As hopcount increases, the time costs of the algorithms increase approximately in the magnitude of node degree, which match our expectation given in the complexity analysis.

Figure 5 represents a comparison of the performance of the two algorithms on graphs with different node degrees. In true-case scenarios, as shown in Figure 5 (a, b and c), we notice that incrementing hopcount limit increases the time for both algorithms to find a qualified path, since the search space expands accordingly. We also observe that when dealing with 2-hop policies, the time cost declines gradually with an increase in node degree. This is mainly because it is more possible to find a qualified path between two nodes at an earlier time in denser graphs than sparser graphs, although the worst possible time for denser graphs is way larger. For 3-hop policies, however, BFS algorithm has to explore all possible paths at the first 2 hops until attempting the 3rd hop, thus spending much more time to find a match when node degree increases. DFS algorithm, on the other hand, does not suffer from the greater search space brought by the increase of node degree. In general, both algorithms perform similarly on 1 and 2-hop policies, but DFS algorithm outperforms its BFS counterpart when dealing with 3-hop policies and larger. Similar to the first set of experiments, we obtain similar results for both algorithms in false-case scenarios (5 (d)), as both of them experienced an exhaustive search. Consistent with our previous analysis on complexity, the results we observed from the four different social graphs reveal an increase of time proportional to the node degrees as expected.

Our results indicate that both node degree and hopcount limit significantly affect the performance of the two algorithms. In some extreme cases (e.g., long enumeration paths, high density graph, etc.), searching a qualified path of 3 hops long may take very long time that the system and users cannot tolerate. However, social graphs in reality are often big and sparse, not many people will have thousands of contacts in the social network. Moreover, people tend to interact with other users within a close distance, so a large hopcount is rather uncommon in practice. If users specify policies with loose constraints (e.g., *-patterns) and small hopcount limit, the algorithms are able to return a result in a reasonably short time. We also suggest the system adds a time out for any access query in order to avoid waiting for those extreme scenarios. Another possible way of mitigating lengthy hops is to allow users to have a customized view

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing



(a) True-case scenarios: *-patterns (b) True-case scenarios: enum-patterns (c) False-case scenarios

Fig. 4: Experiment 1: BFS vs DFS on hopcount

of social graph and create shortcuts for frequently used relationship patterns. Caching might also be an alternative for improving performance [16]. Another important observation from our experiments is that although they have almost the same performance for 1 and 2-hop policies, DFS algorithm in general is likely to be more suitable for policies with intermediate hopcount values (e.g., 3, 4, 5, etc) than its BFS counterpart.

7 Conclusion

In this paper, we proposed a UURAC model and a regular expression based policy specification language. We provided DFS-based and BFS-based path checking algorithms and analyzed the complexity for the algorithms. We demonstrated the feasibility of our approach by discussing a proof-of-concept implementation of both algorithms, followed by the evaluation results.

We believe the proposed model in this paper provides a solid foundation for more advanced ReBAC solutions in the future. We have extended this work to a new model, namely URRAC, which exploits user-to-resource and resource-to-resource relationships as well [13]. We have also proposed an attribute-aware UURAC model that incorporates attribute-based policies to ReBAC [15].

Acknowledgment

This work is partially supported by grants CNS-0831452 and CNS-1111925 from the National Science Foundation.

References

[1] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. *CoRR*, abs/1111.4570, 2011.

[2] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. *ACM SIGCOMM Computer Communication Review*, 39(4):135–146, 2009.

[3] d. m. boyd and N. B. Ellison. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2007.

[4] S. Braghin, V. Iovino, G. Persiano, and A. Trombetta. Secure and policy-private resource sharing in an online social network. In *PASSAT 2011*, pages 872–875. IEEE, 2011.

[5] 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.

[6] B. Carminati and E. Ferrari. Collaborative access control in online social networks. In *CollaborateCom 2011*, pages 231–240. IEEE, 2011.

[7] B. Carminati, E. Ferrari, and J. Girardi. Performance analysis of relationship-based access control in osns. In *IEEE IRI 2012*, pages 449–456, 2012.

[8] 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 ACM SACMAT*, pages 177–186. ACM, 2009.

[9] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. Semantic web-based social network access control. *Computers and Security*, 30(2C3), 2011.

[10] B. Carminati, E. Ferrari, and A. Perego. Rule-based access control for social networks. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 1734–1744. Springer, 2006.

[11] 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.

[12] B. Carminati, E. Ferrari, and A. Perego. Enforcing access control in web-based social networks. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[13] 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.

[14] Y. Cheng, J. Park, and R. Sandhu. A user-to-user relationship-based access control model for online social networks. In *Data and Applications Security and Privacy XXVI*, pages 8–24. Springer, 2012.

[15] Y. Cheng, J. Park, and R. Sandhu. Attribute-aware relationship-based access control for online social networks. In *Data and Applications Security and Privacy XXVIII*, pages 292–306. Springer, 2014.

[16] J. Crampton and J. Sellwood. Caching and auditing in the RPPM model. In *Security and Trust Management*, pages 49–64. Springer, 2014.

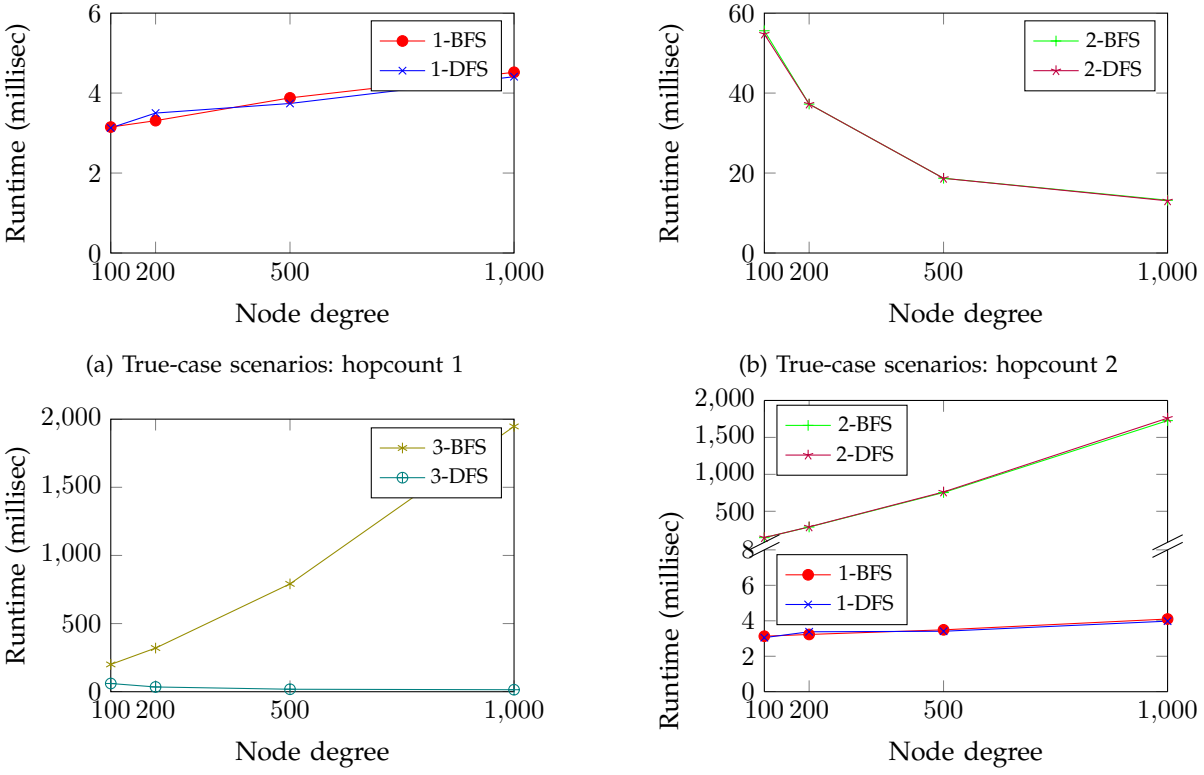
[17] J. Crampton and J. Sellwood. Path conditions and principal matching: a new approach to access control. In *Proceedings of the 19th ACM SACMAT*, pages 187–198. ACM, 2014.

[18] P. W. Fong. Relationship-based access control: protection model and policy language. In *Proceedings of the first CODASPY*, pages 191–202. ACM, 2011.

[19] P. W. Fong, M. Anwar, and Z. Zhao. A privacy preservation model for facebook-style social network systems. In *Computer Security–ESORICS 2009*, pages 303–320. Springer, 2009.

[20] 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2406705, IEEE Transactions on Dependable and Secure Computing



(a) True-case scenarios: hopcount 1

(b) True-case scenarios: hopcount 2

(c) True-case scenarios: hopcount 3

(d) False-case scenarios

Fig. 5: Experiment 2: BFS vs DFS on node degree

[21] H. Gao, J. Hu, T. Huang, J. Wang, and Y. Chen. Security issues in online social networks. *Internet Computing, IEEE*, 15(4):56–63, 2011.

[22] C. Gates. Access control requirements for Web 2.0 security and privacy. *IEEE Web 2.0*, 2007.

[23] M. Hart, R. Johnson, and A. Stent. More content-less control: Access control in the Web 2.0. *IEEE Web 2.0*, 2007.

[24] H. Hu and G.-J. Ahn. Multiparty authorization framework for data sharing in online social networks. In *Data and Applications Security and Privacy XXV*, pages 29–43. Springer, 2011.

[25] H. Hu, G.-J. Ahn, and J. Jorgensen. Detecting and resolving privacy conflicts for collaborative data sharing in online social networks. In *Proceedings of the 27th ACSAC*, pages 103–112. ACM, 2011.

[26] H. Hu, G.-J. Ahn, and J. Jorgensen. Multiparty access control for online social networks: model and mechanisms. *Knowledge and Data Engineering, IEEE Transactions on*, 25(7):1614–1627, 2013.

[27] S. Jahid, P. Mittal, and N. Borisov. Easier: Encryption-based access control in social networks with efficient revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 411–415. ACM, 2011.

[28] 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.

[29] A. Masoumzadeh and J. Joshi. OSNAC: An ontology-based access control model for social networking systems. In *SocialCom 2010*, pages 751–759. IEEE, 2010.

[30] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[31] J. Park, R. Sandhu, and Y. Cheng. ACON: Activity-centric access control for social computing. In *ARES 2011*, pages 242–247. IEEE, 2011.

[32] 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.

[33] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[34] A. C. Squicciarini, M. Shehab, and F. Paci. Collective privacy management in social networks. In *Proceedings of the 18th international conference on World wide web*, pages 521–530. ACM, 2009.

[35] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[36] J. Travers and S. Milgram. An experimental study of the small world problem. *Sociometry*, 32(4):425–443, 1969.

[37] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

Yuan Cheng received the PhD degree in Computer Science from the University of Texas at San Antonio in 2014. He is currently a postdoctoral researcher in the Institute for Cyber Security at the University of Texas at San Antonio. His main research interests include access control, security and privacy in social computing.

Jaehong Park received the PhD degree in Information Technology from George Mason University. He is currently a research associate professor at the Institute for Cyber Security, University of Texas at San Antonio. His research interests include data and application security and privacy, access and usage control, cloud computing security, secure provenance and social computing.

Ravi Sandhu is founding Executive Director of the Institute for Cyber Security at the University of Texas San Antonio, and holds an Endowed Chair. He is an ACM, IEEE and AAAS Fellow and inventor on 29 patents. He is past Editor-in-Chief of the IEEE Transactions on Dependable and Secure Computing, past founding Editor-in-Chief of ACM Transactions on Information and System Security and a past Chair of ACM SIGSAC. He founded ACM CCS, SACMAT and CODASPY, and has been a leader in numerous other security conferences. His research has focused on security models and architectures, including the seminal role-based access control model. His papers have accumulated over 26,000 Google Scholar citations, including over 6,400 citations for the RBAC96 paper.