ACCESS CONTROL FOR ONLINE SOCIAL NETWORKS USING RELATIONSHIP TYPE PATTERNS

APPROVED BY SUPERVISING COMMITTEE:

Ravi Sandhu, Ph. D., Co-Chair

Jaehong Park, Ph. D., Co-Chair

Rajendra Boppana, Ph. D.

Jianwei Niu, Ph. D.

Shouhuai Xu, Ph. D.

Accepted:

Dean, Graduate School

DEDICATION

This dissertation is dedicated to my parents, Mr. Diansheng Cheng and Ms. Peiqin Shou, who support and inspire me each step of the way with their unconditional love.

ACCESS CONTROL FOR ONLINE SOCIAL NETWORKS USING RELATIONSHIP TYPE PATTERNS

by

YUAN CHENG, M. Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2014

ACKNOWLEDGEMENTS

I would like to express my sincerest appreciation and profoundest gratitude to my advisors Dr. Ravi Sandhu and Dr. Jaehong Park for their guidance and support throughout my doctoral studies. This dissertation would not be accomplished without their inspiring ideas, critical comments, and constant encouragement.

I would like to extend my gratitude to other members of my committee, Dr. Rajendra V. Boppana, Dr. Jianwei Niu, and Dr. Shouhuai Xu, for their insightful comments and the time they devoted to reading the dissertation. Special thanks go to my best friends and colleagues, Mr. Dang Nguyen and Mr. Khalid Zaman Bijon. Words cannot express my gratefulness for their incessant help, humor, and encouragement.

Finally, I would like to thank many other colleagues and friends at the University of Texas at San Antonio for their help and support during all these years, which made my stay very enjoyable.

This work has been graciously supported by the National Science Foundation grant CNS-0831452 and CNS-1111925 and by the State of Texas Emerging Technology Fund. Chapter 3 and part of Chapter 4 have appeared as [17]. The rest of Chapter 4 has not been previously published. Chapter 5 has been accepted and will appear in DBSec 2014. Chapter 6 has appeared as [16] and won the Best Paper Award in PASSAT 2012.

May 2014

ACCESS CONTROL FOR ONLINE SOCIAL NETWORKS USING RELATIONSHIP TYPE PATTERNS

Yuan Cheng, Ph. D.
The University of Texas at San Antonio, 2014

Supervising Professors: Ravi Sandhu, Ph. D. and Jaehong Park, Ph. D.

Users and resources in online social networks (OSNs) are interconnected via various types of relationships. User-to-user (U2U) relationships form the basis of the OSN structure, the social graph, and play a significant role in specifying and enforcing access control. In fact, U2U relationship-based access control (ReBAC) has been adopted as the most prevalent approach for access control in OSNs, where authorization is typically made by tracking the existence of U2U relationships of certain types and/or depth between the access requester and the resource owner. We propose a novel ReBAC model for OSNs that incorporates different types of relationships and utilizes regular expression notation for policy specification, namely UURAC (User-to-User Relationship-based Access Control). Authorization policies are defined in terms of the patterns of relationship path on social graph and the hopcount limit of the path. In addition, two path checking algorithms are developed to determine whether the required relationship path for a given access request exists, and proofs of correctness and complexity analysis for the algorithms are provided. The UURAC model is implemented and evaluated to validate our approach.

We subsequently 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 otherwise available in ReBAC.

Today's OSN applications allow various user activities that cannot be controlled by using U2U relationships alone. To enable a comprehensive ReBAC mechanism, we develop the URRAC (User-to-Resource Relationship-based Access Control) model to exploit user-to-resource (U2R) and resource-to-resource (R2R) relationships in addition to U2U relationships for authorization decision. While most of today's access control solutions for OSNs only focus on controlling user's normal usage activities, URRAC model also captures controls on user's administrative activities. Simple specifications of conflict resolution policies are provided to resolve potential conflicts among authorization policies.

The objective of this research is to demonstrate that greater generality and flexibility in policy specification and effective access evaluation can be achieved in OSNs using relationship type patterns and attributes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

Online social networks (OSNs) have emerged and thrived rapidly over the past several years and now have billions of users worldwide. A recent survey found that 73% of online adults use a social network of some kind [21]. Many existing OSNs facilitate convenient environments and various kinds of services for participating users to regularly make new connections, interact and share information with each other 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, including demographic information, contact information, education information, blog posts, pictures, videos, comments, and so on. A majority of the information is made public without user's careful consideration. 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 [8, 26, 30]. 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 is responsible for defining 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.

Moreover, 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 and attributes of subjects and objects, where attributes may include group or role memberships, access control lists, capability lists, and security labels, etc. However, identity and attribute-based approaches fail to cope with the scalability and dynamicity of OSNs. In OSNs, it is not practical for users to specify all of the authorized users who can access their information in a traditional way.

Instead, access control in OSNs is typically based on the relationships among users on the social graph. This type of relationship-based access control (ReBAC) [23, 25] 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 contacts. Accordingly, relationship-based access control has been recognized as a key requirement for security and privacy in OSNs [27], and has been commonly adopted in real world OSN systems since it keeps the balance between ease-of-use and flexibility.

1.1 Motivation

Despite its popularity in both theory and practice, current ReBAC is still far from perfect. Most existing OSN systems enforce a rudimentary and limited ReBAC mechanism, offering users the ability to choose from a pre-defined 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 ReBAC models [9,11–15,23–25,39]. These proposals explore more flexible and expressive solutions than provided by current commercial OSNs. Policies in [9, 11–15, 23, 25] can be composed of multiple types of relationships. [13–15] also adopt the depth and the trust value of relationship to control the spread of information. Although only having the “friend” relationship type, [24] provides additional topology-based policies, such as known quantity, common friends and stranger of more than k distance.

One common characteristic found in most of these commercial and academic solutions is that they mainly focus on user-to-user (U2U) relationships between the accessing user and the resource owner, and at least implicitly assume ownership is the only manifestation of user-to-resource (U2R) relationships. However, this is not sufficient to capture many user activities found in today’s OSN applications, where users can perform actions that create relationships between users and resources other than ownership. For example, tagging a friend on a photo will create U2R relationship between the photo and the tagged user which consequently may allow friends of the tagged user to access the photo. Hence the tagged user may want to control other related users’ access to the photo. Likewise, users’ actions can establish resource-to-resource (R2R) relationships such as photos under the same album, comments to a blog post, etc. To enable a fully expressive relationship-based access control, it is necessary to exploit U2R and R2R relationships in addition to U2U relationships for authorization policies and decisions.

Moreover, 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.

To show motivation for our work, let us start with some examples that cannot be properly addressed by current access control solutions.

1.1.1 Related User's Control

Due to the various functionality offered by today's OSNs, there exist several different types of relationships between users and resources in addition to ownership. Consider an example where Bob posts a photo that contains Alice and Carol's image in it and tags them. OSNs usually allow only the owner Bob to choose the audience for that photo, regardless of whether or not Alice and Carol may wish to release their images. To enable Alice and Carol some control capability on the photo, their relationships with the photo, which is not ownership, should be considered for authorization purposes. After the photo has been shared by Bob's friends several times, more and more users from different neighborhoods in the network come to view the photo and comment on it. When Dave reads through all the comments in Bob's photo and becomes curious about another user Eve who has commented recently, he decides to poke her to say hello. In this case, Dave and Eve are connected through the photo, not through another user (such as the owner of the photo Bob). Also, users may share or like the blog posts or videos posted by others, and gain the ability to determine how the shared/liked copy of the original content or the fact of sharing and liking activities can be seen by others. Consider another scenario where Betty finds a weblink originally posted by Ed interesting and then shares it with her friends. From this activity, she acquires the ability to decide how the weblink can be available to others.

1.1.2 Administrative Control

In OSN, users are allowed to configure access control policies for their own content and activities. Allowing U2R relationship-based access control further enables users to specify policies for contents related to them and activities of other related users. Hence, policy administration becomes very important since allowing individual users to specify policies requires the OSN to ensure that only the right users are authorized to do it. Moreover, since a change of relationships may result in a change of authorization, the creation and termination of relationships needs to be treated differently from usage activities to normal resources. Thus, access control in OSNs has to address the management of access control policies and relationships in addition to normal usage activities by means of U2U, U2R and R2R relationships. Carminati et al [11, 12] introduced a framework that allows system administrators to specify administrative policies in ontology-based representations. However, they did not provide a policy management model for managing policies and resolving policy conflicts. Most of the other relationship-based access control models do not incorporate users' administrative activities.

We address administrative activities, such as policy specifying, relationship invitation and relationship recommendation, separate from normal activities. Let us consider an example where parental control, such as allowing parents to configure their children's policies, is featured in the system. Bob's mother Carol may not want Bob to become friend with her colleagues, to access any violent content or to share personal information with others. Both Bob and Carol can specify policies for these activities, thus the system needs a strategy to resolve any potential conflicts among policies from different users.

1.1.3 Attribute-aware Relationship-based Access Control

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 polices 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 [28, 29, 39], 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.

1.2 Problem Statement

OSNs offer users various types of user interaction services, including chatting, private messaging, poking and social games. As OSN systems mature, various types of resources need to be protected, such as user sessions, relationships among user and resources, access control policies and events of users. However, as we identified earlier, traditional access control models are not adequate for today's OSN environment, because OSN systems access control is based on relationships found in social graph.

Many of the existing OSN systems enforce a coarse-grained and limited relationship-based access control, mainly based on U2U relationships between the accessing user and the target user. When a user requests access to a resource, current OSNs rely on an implicit ownership relationship, between the resource and its owner, hence the authorization of such U2R access is still based on the underlying U2U relationships. In today's fast evolving OSNs, we have identified various types of relationships between users and resources in addition to ownership. There are many scenarios where related users other than the owner want to exert their control capability on the resource they share certain types of U2R relationships with. Thus there exhibits a huge gap between users' mental model and the control offered by the systems.

As a consequence of allowing U2R relationships, users are able to specify policies for others, hence policy administration has to be properly managed to ensure only the right users can do so. Furthermore, since multiple users can express access control policies for a user or a resource, it is expected that there will be several policies applicable to the same access request which will inevitably raise conflicts. For example, Bob sets his policy so that he can get friendship request from anyone in the system, while at the same time policies defined by his parents may only allow him to receive such request from his friends of friends. To resolve such conflicts, it is necessary to introduce conflict resolution policies, which are meta-policies about how authorization policies are to be interpreted and how policy conflicts are resolved.

Using relationships alone is often not sufficient to enforce various security and privacy requirements that meet the expectation from today's OSN users. In addition to relationships, we believe that it is beneficial to incorporate attribute-based access control into an existing ReBAC model. A new policy specification language needs to be formalized to address the access requirements in terms of the attributes of users, relationships and social graphs.

1.3 Scope and Assumptions

Protection of security and privacy in OSNs can be divided into several aspects, including user's identity anonymity, user's content privacy, user's communication privacy, and so on [56]. Many efforts have been made by the community to solve each of these concerns. The aspect of user's content privacy implies the need for access control. Access control is of paramount importance in OSNs, since unauthorized access to user's contents may cause undesirable or damaging consequences to users. It is also the primary focus in this dissertation. In particular, we aim to improve the relationship-based access control mechanism for OSNs.

To offer richer functionality, many OSNs have launched social networking platforms that enable third-party developers to contribute applications to the social network through the use of APIs (application programming interfaces). With the support of OSN platforms, third-party applications (TPAs) have become highly popular in a very short period of time. The emergence of TPAs also poses severe privacy risks to users. For example, TPAs usually receive privileges equal to the TPA users with respect to social graph traversal, and thereby gain access to an abundance of users' information regardless of the actual legitimate needs. Moreover, these applications are available via OSNs but are running on external servers outside the OSN's control. Once they acquire the data, they can use or dispose it in whatever way they want without user or OSN consent. There is no control regarding the usage of user data once it is released to the TPA. The developers of TPAs can aggregate such data and accrue benefit by using or selling the data. We recognize the importance of the security and privacy issues regarding TPAs, and have developed an access control framework that provides flexible and fine-grained controls on how TPAs can access OSN user's data [18]. However, like most other research on access control in OSNs, this dissertation concentrates on access between regular users in the system, leaving the issues of TPAs out of scope.

Relationships in OSNs are usually one-to-one, and require an explicit procedure of initiation and termination. However, as more and more functions and services become available in OSNs, some unconventional relationships are being introduced to OSNs, such as temporary relationships (e.g., vicinity) and one-to-many relationships (i.e., network, group). These new relationships can be potentially exploited for the purpose of access control as well. But we decided to only focus on the conventional relationships in OSNs, and leave the unconventional ones as a possible future direction of this work.

This work is based on the following assumptions.

- Unlike some privacy preservation techniques proposed in [31, 40, 41], our threat model does not include OSN providers. Since all of the information provided by users is stored in OSN servers and processed by OSN supported services, OSN providers can easily monitor and analyze these data for their own purposes, such as targeted advertising, surveillance, data mining, user experience improvement, etc. Although we have witnessed the ongoing debates about ethics of the OSN sites, we still believe that totally preventing OSN providers from user's real data is not realistic and is conflicting with the business model of OSN providers.

- We assume that user's computers are not compromised by malicious intruders or malwares. We also do not consider the case when an attacker gains unauthorized access to a site's code and logic. The enormous amounts of information OSNs process each and every day end up making it much easier to exploit a single flaw in the system. However, in this work we only aim to overcome the limitations of the access control mechanisms provided by current OSN systems.

1.4 Thesis

The central thesis of this dissertation is given below.

Users and all of the resources related to users are interconnected through U2U, U2R and R2R relationships, which form the basis of an OSN system, the social graph. By utilizing regular expression notation for policy specification, it is efficient and effective to regulate access in OSNs in terms of the pattern of relationship path on the social graph and the hopcount limit on the path. Integrating attribute-based policies further enables finer-grained controls that are not available in

ReBAC alone.

1.5 Summary of Contributions

The major contributions of this research are stated as follows:

- Develop an access control framework for OSNs based on relationships, supporting the essential characteristics that need to be addressed by OSN access control.
- Further build two ReBAC models that utilize different kinds of relationships, using regular expression notation. They incorporate greater generality and flexibility of path patterns in policy specifications, including the incorporation of inverse relationships.
- Integrate attribute-based polices into relationship-based access control. The attribute-aware ReBAC enhances access control capability and allows finer-grained controls that are not available in ReBAC.
- Provide two effective path-checking algorithms for UURAC policy evaluation, along with proofs of correctness and complexity analysis. An enhanced path-checking algorithm for attribute-aware ReBAC is also presented to determine the existence of the required attributes and relationships in order to grant access.
- Implement the proposed path-checking algorithms.
- Conduct experiments to evaluate the performance of the access control procedure.

1.6 Organization of the Dissertation

Chapter 2 gives a brief background on ReBAC in OSNs and reviews related work that focus on access control and privacy preservation mechanisms for OSNs. Chapter 3 presents the model definition and policy specifications of the proposed UURAC model. In Chapter 4, we introduce two path-checking algorithms with proofs of correctness and complexity analysis, and evaluate the performance of the algorithms. Chapter 5 identifies the value of attribute-based policies for ReBAC, and describes an attribute-aware ReBAC model that incorporates such attribute-based policies. In Chapter 6, we propose the URRAC model that captures U2R and R2R relationships for authorization. Chapter 7 summarizes the completed research and discusses future work.

Chapter 2: BACKGROUND AND LITERATURE REVIEW

In this chapter, we provide a brief overview of necessary background on the security and privacy issues found in OSNs, and examine existing access control and privacy preservation solutions for OSNs.

2.1 Security and Privacy Issues in OSNs

Online social networks have emerged and thrived rapidly over the past several years now having billions of users worldwide. OSNs have become the most popular portal for users to regularly connect, interact and share information with each other for a multitude of purposes.

Although the nature and nomenclature found in OSNs may vary from one site to another, OSNs basically allow individual users to construct a public or semi-public profile within a bounded system, articulate a list of other users with whom they share a connection, and view and traverse their list of connections and those made by others within the system [8].

The explosive growth of sensitive or private user data that are readily available in OSNs has made security and privacy of information one of the most critical issues to be addressed. According to the survey by Gao et al [26], security issues in OSNs can be at least generalized into four categories: privacy breaches, spam and phishing attacks, sybil attacks and malware attacks.

Users share a vast amount of content with other users in OSNs using various services. Such an abundance of information makes privacy breach very easy to happen from OSN providers, other users, and third party applications. OSN systems keep all the information users have uploaded. Thus users have to trust OSN providers to protect and not to misuse the data. However, OSNs can benefit from analyzing and sharing user data and activity history for many purposes, such as targeted advertising and service improvement.

Many OSNs also allow third party applications to run on their platforms and offer users additional functionalities. Users grant permission to third party applications during installation, but the control mechanism is simply all-or-nothing. Hence, those applications may often get access to more information than they actually need for proper functioning [22], or may be able to post on user's profile or access user's data without user's knowledge.

Another major kind of threats are from peer users in OSNs, since many OSN users are not aware of who they share the data with and how much. Moreover, in a recent report [42], half of OSN users say they have some difficulty in managing privacy controls offered by OSN sites. This implies that a suitable and effective access control mechanism is needed for protecting user's data from unwanted access, which is the main focus of this research.

2.2 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 [45, 46] 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 friends' activities could be bothersome and a user may want to block it. This type of policy is captured as incoming action policy. Also, a user may want to control her own or other users' activities. For example, a user may restrict her own access from 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").

2.3 Relationships as Basis of Authorization

Typically, an OSN can be modeled as a graph, where nodes correspond to users and edges denote relationships between users. Moreover, since in many OSNs there exist non-mutual relationships of different types (e.g., follow, parent, etc.), it is more general to associate label and direction with the edges on the graph. More precisely, OSN can be abstracted as a directed labeled simple graph, an example of which is shown in Figure 3.3.

Social relationships connect different social entities (e.g., people, organizations, etc.) and form the basis of the social network structure. The impact of the Internet on social relationships has been a major research topic in social science in the past decade. The majority of the research has been conducted on how the Internet affects the formation and maintenance of new online relationships with strangers, or on how the Internet helps to sustain existing offline relationships. In computer science community, on the other hand, researchers have been aware of the significant role online social relationships can play in protecting security and privacy of data in OSNs.

An OSN is basically a set of social entities connected by a set of relationships. Systems usually provide users various services for both the maintenance of existing social ties and the formation of new connections with other users. Based on such relationships, for example, users can identify contacts of their contacts, or get notification about the updates from their contacts.

As an OSN is built upon social graph, a new paradigm of access control, called Relationship-based Access Control (ReBAC), has been developed based on relationships between users on the social graph. This type of access control 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 relationships.

2.4 Existing Models for Relationship-based Access Control

The large and complex collections of user data in OSNs require usable and fine-grained access control solutions to protect them [27, 33]. Gates [27] discusses the access control requirements for OSN environments, where he argues that one of the key requirements is relationship-based access control.

Fong et al [23] proposed a formal ReBAC model for social computing applications, which employs a modal logic language for policy specification and composition. In [25], Fong et al later extended the policy language and studied its expressive power. These two models allow multiple relationship types and directional relationships. Authorization are based on U2U relationships between the accessing user and the resource owner, and relationships are articulated in contexts.

Inspired by research in trust and reputation systems, some early solutions proposed by Kruk et al [39] and Carminati et al [13, 14] 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 [15]. The system features a centralized certificate authority to assert the validity of relationship paths, while access control enforcement is carried out on the decentralized user side.

A formal model for access control in Facebook-like systems was developed by Fong et al [24], 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 [11, 12], Carminati et al proposed an access control framework which utilizes relationships among users and resources as the basis for access control and employs the Semantic Web Rule Language (SWRL) to define authorization, administration and filtering policies. Our URRAC model proposed in this work offers more complete policy administration by addressing policy management and conflict resolution. Another semantic web-based approach proposed in [43] allows both users and the system to express policies based on access control ontologies.

The first four columns of Table 2.1 summarize the salient characteristics of the models discussed above. The fifth column gives these characteristics for the UURAC model [17] and the URRAC model [16].

All the models deal only with U2U relationships, except [11,12,16] also recognize U2R (user-to-resource) relationships explicitly. U2R relationships can be captured implicitly via U2U with the last hop being U2R.

Table 2.1: Comparison of Access Control Models for OSNs

	Fong [24]	Fong [23, 25]	Carminati [15]	Carminati [11, 12]	UURAC, URRAC & UURAC$_A$
Relationship Category					
Multiple Relationship Types		✓	✓	✓	✓
Directional Relationship		✓	✓		✓
U2U Relationship	✓	✓	✓	✓	✓
U2R Relationship				✓	✓(only URRAC)
Model Characteristics					
Policy Individualization	✓	✓	✓	✓	✓
User & Resource as a Target				(partial)	✓
Outgoing/Incoming Action Policy				(partial)	✓
Relationship Composition					
Relationship Depth	0 to 2	0 to n	1 to n	1 to n	0 to n
Relationship Composition	f, f of f	exact type sequence	path of same type	exact type sequence	path pattern of different types
Attribute-aware Access Control					
Common-friends$_k$	✓				✓(only UURAC$_A$)
User Attributes		(partial)			✓(only UURAC$_A$)
Relationship Attributes			(partial)		✓(only UURAC$_A$)

Having seen the various notations used for expressing policies in the above works, Aktoudianakis et al [1] later on demonstrated that their proposed policy templates using set theoretic notation can provide a general framework for defining relationship-based access control policies.

2.5 Other OSN Privacy Preservation Solutions

NOYB [31] and Facecloak [41] provided extra access control mechanism to users' real data in addition to the existing access control provided by the OSN sites, since they don't trust OSN sites at all. User specifies who will be granted permissions, and disseminates the rights out-of-band. Any other users without the rights as well as the OSN sites can only see the fake data. The slight difference between the two approaches is the former one only addresses the privacy of profile information.

Lucas et al [40] proposed a Facebook application, flyByNight, that offers cryptographic protection on messages exchanged between Facebook users. It supports one-to-one and one-to-many communication. Because of the nature of a Facebook application, its fate is entirely at the discretion of Facebook.

Privacy-by-proxy [22] is a privacy preserving API that deals with the privacy risks between users and the third party applications. It replaces the real user data with tags before sending to the applications. The real data behind the tag is only visible to users who could see this data anyhow at the owner's page. It also assumes the trustworthiness of OSN sites. Because the API can handle the output transformation itself, no major changes need to be applied to either the OSN architecture or the applications.

Chapter 3: THE UURAC MODEL

In this chapter, we develop a novel user-to-user relationship based access control (UURAC) model for OSN systems that utilizes regular expression notations for such policy specification.

3.1 The UURAC Model Foundation

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

3.1.1 Basic Notations

We write Σ to denote the set of relationship type specifiers, where Σ = $\{\sigma_1$, σ_2, $\ldots$, σ_n, σ_1^{-1}, σ_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 σ_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.

3.1.2 Access Control Model Components

The model comprises five categories of components as shown in Figure 3.1.

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 $controlling$

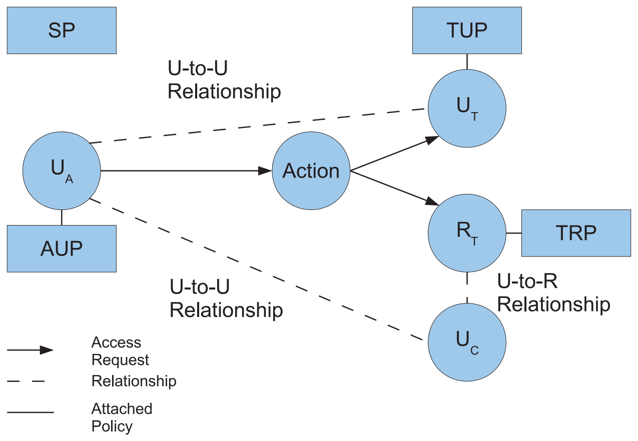


Figure 3.1: The UURAC Model Components

$users$ (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 $\langle u_a, action, target\rangle$, 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.

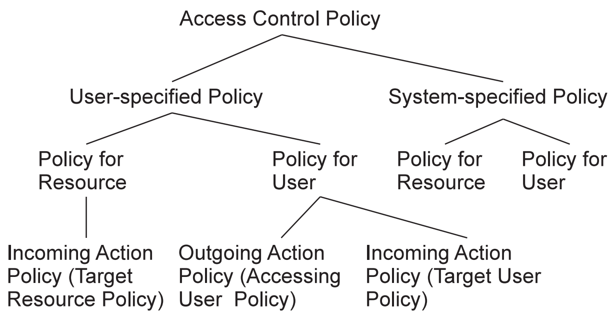

Access Control Policy
User-specified Policy
System-specified Policy
Policy for Resource
Policy for User
Policy for Resource
Policy for User
Incoming Action Policy (Target Resource Policy)
Outgoing Action Policy (Accessing User Policy)
Incoming Action Policy (Target User Policy)

Figure 3.2: Access Control Policy Taxonomy

Policy defines the rules according to which authorization is regulated. As shown in Figure 3.2, policies can be categorized into user-specified and system-specified policies, with respect to who defines the policies. System-specified policies (SP) are system-wide general rules enforced by the

OSN system; while user-specified policies are applied to specific users and resources. Both user- and system-specified policies include policies for resources and policies for users. Policies for resources are used to control who can access a resource, while policies for users regulate how users can behave regarding an action. User-specified policies for a resource are called $target\ resource\ policies$ (TRP), which are policies for $incoming\ actions$. User-specified policies for users can be further divided into $accessing\ user\ policies$ (AUP) and $target\ user\ policies$ (TUP), which correspond to user's outgoing and incoming access (see examples in Section 2.2), 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.

3.1.3 Modeling Social Graph

As shown in Figure 3.3, 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 $G = \langle U, E, \Sigma \rangle$:

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

[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.

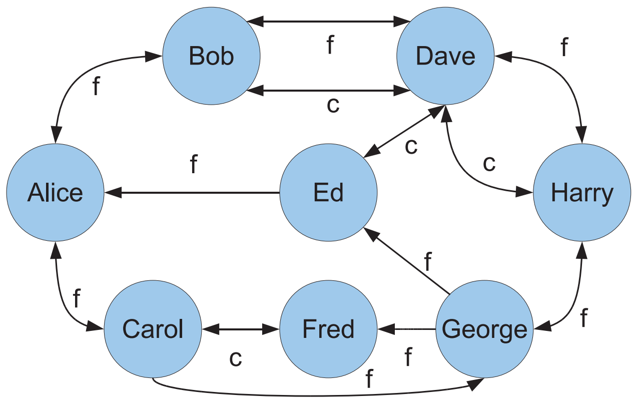


Figure 3.3: A UURAC Sample Social Graph

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

3.2 The UURAC Policy Specifications

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

3.2.1 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.

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

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 3.1. 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

Table 3.1: UURAC Grammar for Graph Rules

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

Table 3.2: UURAC Access Control Policy Representations

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

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.

3.2.3 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 3.2).

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 model 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 chapter 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, $\langle read$, ($filetype$, $photo$), (u_c, $f*$, 4)$\rangle$ 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 3.1).

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

3.2.4 Access Evaluation Procedure

Algorithm 3.1 $AccessEvaluation(u_a, action, target)$

```
1: (Policy Collecting Phase)
2: if $target = u_t$ then
3:     $AUP \leftarrow u_a$'s policy for $action$, $TUP \leftarrow u_t$'s policy for $action^{-1}$, $SP \leftarrow$ system's policy for $action$
4: else
5:     $AUP \leftarrow u_a$'s policy for $action$, $TRP \leftarrow r_t$'s policy for $action^{-1}$, $SP \leftarrow$ system's policy for $action, (r.typename, r.typevalue)$
6: (Policy Evaluation Phase)
7: for all policy in $AUP$, $TUP/TRP$ and $SP$ do
8:     Extract graph rules ($start$, $path\ rule$) from policy
9:     for all graph rule extracted do
10:        Determine the starting node, specified by $start$, where the path evaluation starts
11:        Determine the evaluating node which is the other user involved in access
12:        Extract path rules $path\ rule$ from graph rule
13:        Extract each path spec $path$, $hopcount$ from path rule
14:        Path-check each path spec using Algorithm 4.1 or 4.3
15:        Evaluate the combined result based on conjunctive or disjunctive connectives between path specs and negation on individual path specs
16: Compose the final result from the result of each policy
```

Algorithm 3.1 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 3.3:

- Alice's policy P_{Alice}: $\langle poke, (u_a, (f*, 3))\rangle$ $\langle poke^{-1}, (u_t, (f, 1))\rangle$ $\langle read, (u_a, (\Sigma*, 5))\rangle$

- Harry's policy P_{Harry}: $\langle poke, (u_a, (cf*, 5) \vee (f*, 5))\rangle$ $\langle poke^{-1}, (u_t, (f*, 2))\rangle$

- Policy of file2 P_{file2}: $\langle read^{-1}, Harry, (u_c, \neg(p+, 2))\rangle$

- System's policy P_{Sys}: $\langle poke, (u_a, (\Sigma*, 5))\rangle$ $\langle read, (filetype, photo), (u_a, (\Sigma*, 5))\rangle$

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

3.2.5 Discussion

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

For simplicity we assume system level policies are available to resolve conflicts in user-specified authorization policies and do not consider user-specified conflict resolution policies.

One observation from user-specified policies is that $action$ policy starts from u_a whereas $action^{-1}$ policy starts from u_t. This is because $action$ 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, $\langle poke$, (u_a, ($\emptyset$, 0))$\rangle$ says that u_a can only poke herself, and $\langle poke^{-1}$, (u_t, ($\emptyset$, 0))$\rangle$ 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.

Chapter 4: PATH-CHECKING ALGORITHMS AND EVALUATION

In this chapter, we first introduce the path-checking algorithms for UURAC based on two different traversal strategies. We then describe some experiments to evaluate the performance of the algorithms, and analyze the results of our experiments.

4.1 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 proofs of correctness and complexity analysis for both algorithms.

As mentioned, in order to grant access, relationships between the accessing user 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 $\langle u_a$, $action$, $target\rangle$ 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 $\langle start$, $path\ rule\rangle$.

As shown in Algorithm 4.1 and 4.3, 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, 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 [52] and Subset Construction Algorithm (also known as Büchi's Algorithm) [47], respectively.

4.1.1 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 4.1, 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 4.2 from the starting node s.

In Algorithm 4.2, 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 26). 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 $currentPath$ and v is the target node t, we check if the transition σ belongs to the set of valid transitions for DFA. If the transition is valid and if DFA taking the transition σ reaches an accepting state, we find a path between s and t matching the pattern $Path$ (case 2). We increment d by one, concatenate edge (u, v, σ) to $currentPath$, and save the current DFA state to history. If DFA with transition σ is not at an accepting state, then the path from s to v does not match the pattern (case 3). If the transition is invalid for DFA, we try the next edge (case 4). When v is not on $currentPath$ and is not the target node, there are two cases depending on whether the transition type σ is a valid transition for DFA. If it is not, we break out of the for loop and continue to check the next unchecked edge of u (case 5). Otherwise, the algorithm increments d by one, concatenates e to $currentPath$, moves DFA to the next state via transition type σ, updates the DFA state history, and repeatedly executes $DFST()$ from node v (case 6). 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.

4.1.2 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 4.3, 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.

Algorithm 4.1 $DFSPathChecker(G, path, hopcount, s, t)$

```
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 4.2 $DFST(u)$

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

All edges can be classified into the same six cases as in the above mentioned 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 2). 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 the transition, update the variables of v, and enqueue node v into the queue for later examination (case 6). 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.

4.1.3 Proof of Correctness

Theorem 1. *Algorithm 4.2 and 4.3 will halt with true or false.*

Proof. Base case ($Hopcount$ = 1): d is initially set to 0. Each outgoing edge from the starting node s will be examined once and only once. If taking an edge reaches the target node t and its type matches the language $Path$ denotes (case 2), both algorithms return true. In Algorithm 4.2, if the edge type matches the prefix of an expression in $L(Path)$ (case 6), d increments to 1 followed by a recursive call to $DFST()$. The second level call will return false, since incremented d has exceeded $Hopcount$. In all other cases, the examined edge is discarded and d remains the same. In Algorithm 4.3, if the edge type matches the prefix of an expression in $L(Path)$ (case 6), the new node v is added to the queue with an incremented d. When v is dequeued later, the new edge will be dropped as the updated d has exceeded $Hopcount$. In all other cases, the examined edge is discarded and d remains the same. Eventually, if a matching edge is not found, both algorithms will go through every outgoing edge from s and exit with false thereafter.

Induction step: Assume when $Hopcount$ = k ($k \geq 1$), Theorem 1 is true.

In Algorithm 4.2, when $Hopcount$ is set to $k + 1$, all the $(k + 1)$th level recursive calls will examine every outgoing edge from the $(k+1)$th node on $currentPath$. If visiting an edge reaches t and the updated $currentPath$ matches $L(Path)$, the $(k + 1)$th level call returns true and exits

Algorithm 4.3 $BFSPathChecker(G, path, hopcount, s, t)$

```
1: $DFA \leftarrow REtoDFA(path)$
2: **if** $hopcount \neq 0$ **then**
3:     create queue $Q$
4:     create node $s$: $s.DFA \leftarrow DFA$; $s.currentPath \leftarrow NIL$; $s.d \leftarrow 0$; $s.stateHistory \leftarrow$ 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** $(v, \sigma)$ where $(q, v, \sigma)\ in\ G$ **do**
12:                **switch**
13:                **case 1** $v \in currentPath$
14:                break
15:                **case 2** $v \notin currentPath$ and $v = t$ and DFA with transition $\sigma$ is at accepting state
16:                create node $v$ (clone from $q$)
17:                $v.previousState \leftarrow v.currentState$
18:                $v.currentState \leftarrow$ DFA takes transition $\sigma$
19:                $v.d++$
20:                $v.currentPath$ adds $(q, v, \sigma)$
21:                $v.stateHistory$ adds $currentState$
22:                **return** TRUE
23:                **case 3** $v \notin currentPath$ and $v = t$ and transition $\sigma$ is valid for DFA but DFA with transition
                   $\sigma$ is not at accepting state
24:                break
25:                **case 4** $v \notin currentPath$ and $v = t$ and transition $\sigma$ is invalid for DFA
26:                break
27:                **case 5** $v \notin currentPath$ and $v \neq t$ and transition $\sigma$ is invalid for DFA
28:                break
29:                **case 6** $v \notin currentPath$ and $v \neq t$ and transition $\sigma$ is valid for DFA
30:                create node $v$ (clone from $q$)
31:                enqueue $v$ onto $Q$
32:                $v.previousState \leftarrow v.currentState$
33:                $v.currentState \leftarrow$ DFA takes transition $\sigma$
34:                $v.d++$
35:                $v.currentPath$ adds $(q, v, \sigma)$
36:                $v.stateHistory$ adds $currentState$
37:    **return** FALSE
```

to the previous level, making all of the previous level calls all the way back to the first level exit with true as well. If an edge falls into case 6, d is incremented to $k + 2$ and a $(k + 2)$th level recursive call invokes, which will halt with false and return to the $(k+1)$th level as d has exceeded $Hopcount$. After all edges are examined without returning true, the algorithm will exit with false to the previous level. In the kth level, when $Hopcount$ = $k + 1$, edges without taking a recursive call are treated the same as they are when $Hopcount$ = k. Since when $Hopcount$ = k the theorem holds, the algorithm will terminate with true or false when $Hopcount$ = $k + 1$ as well.

In Algorithm 4.3, when $Hopcount$ increases to $k+1$, all of the edges leaving kth level nodes are examined in the same way as before. If there exists a qualifying path between s and $(k+1)$th level nodes, the algorithm returns true; otherwise, it enqueues $(k+1)$th level nodes whose $currentPath$ matches the prefix of pattern, and continues to examine edges leaving the next dequeued node, as right now the incremented d does not exceed $Hopcount$ any more. If visiting one of the edges from $(k + 1)$th nodes reaches t and the updated $currentPath$ matches $L(Path)$, the algorithm returns true. If $currentPath$ matches only the prefix of $L(Path)$, the corresponding $(k + 2)$th node will be added to the queue. However, when the new node is dequeued later, the algorithm will halt with false, since d exceeds $Hopcount$. The algorithm eventually exits false as all other edges leaving $(k + 1)$th level nodes are dropped during path checking. Therefore, the algorithm will terminate with true or false when $Hopcount$ = $k + 1$ as well. □

Lemma 1. *At the start and end of each call in Algorithm 4.2 and during the execution of Algorithm 4.3, the DFA corresponding to $Path$ is at $currentState$ reachable from the starting state π_0 by transitions corresponding to the sequence of symbols in $currentPath$.*

Proof. The proof is straightforward. New edge is added to $currentPath$ only when it reaches the target node (case 2) or it may possibly lead to the target node by examining the child of the current node (case 6). In both cases the DFA starting from π_0 will move to $currentState$ by taking the transition regarding the edge. Removing the last edge on $currentPath$ after all edges leaving the current node are checked always accompanies one step back-off of the DFA to its previous state in Algorithm 4.2 (lines 26-29), which can eventually take the DFA all the way back to the starting state π_0. □

Theorem 2. *If Algorithm 4.2 or 4.3 returns true, $currentPath$ gives a simple path of length less than or equal to $Hopcount$ and the string described by $currentPath$ belongs to the language described by $L(Path)$. If Algorithm 4.2 or 4.3 returns false, there is no simple path p of length less than or equal to $Hopcount$ such that the string representing p belongs to $L(Path)$.*

Proof. Base case ($Hopcount$ = 1): At first, $d = 0$, $currentPath$ = NIL, and the DFA is at the starting state π_0. When $d = 0$, case 1 requires that the edge being checked is a self loop which is not allowed in a simple graph. Both algorithms only return true in case 2, where edge (s, t, σ) to be added to $currentPath$ finds the target node t in one hop. The transition σ moves the DFA to an accepting state. Case 6 cannot return true, because incrementing d by one will exceed $Hopcount$ when the new node is being examined. When algorithms exit with true, due to Lemma 1, $currentPath$, which is (s, t, σ), can move the DFA from π_0 to an accepting state π_1, implying that $\sigma \in L(Path)$. If the algorithms return false, they have searched all the edges leaving node s. However, these examined edges either do not match the pattern specified by $L(Path)$ (case 3, 4 and 5), or may possibly match $L(Path)$ but require more than one hop (case 6). Hence, Theorem 2 is true when $Hopcount$ = 1.

Induction step: Assume when $Hopcount$ = k ($k \geq 1$), Theorem 2 is true. For the same G, $Path$, s and t, executions of $DFST()$ when $Hopcount$ = k and $k + 1$ only differ after invoking the recursive $DFST()$ call in case 6 of Algorithm 4.2 or new nodes are added to the queue in case 6 of Algorithm 4.3. If an edge being checked can make the algorithm return true when $Hopcount$ = k, $currentPath$ is a string of length $\leq k$ which is in $L(Path)$. When $Hopcount$ is $k + 1$, the same $currentPath$ gives the same string and is of length $< k + 1$, thus making the function exit with true as well. The only difference between $Hopcount$ = k and $Hopcount$ = $k + 1$ is that adding edges that lie in case 6 to $currentPath$ and incrementing d by one may not exceed the larger $Hopcount$ during the examination of the new node. If taking one of these edges leads to the target node and its corresponding transition moves the DFA to an accepting state, the algorithm will return true. The new $currentPath$ gives a simple path of length $k + 1$ that connects node s

and t. The algorithm only returns true in these two scenarios. In both scenarios, based on Lemma 1, the DFA can reach an accepting state by taking the transitions corresponding to $currentPath$, so the string corresponding to $currentPath$ is in $L(Path)$. If the algorithm returns false when $Hopcount$ = k, there is no simple path p of length $\leq k$, where the string of symbols in p is in $L(Path)$. When $Hopcount$ is $k+1$, given the same G, such a path still does not exist. By taking a recursive $DFST()$ call in case 6 of Algorithm 4.2 or adding a new node to the queue in case 6 of Algorithm 4.3, both algorithms will go through all 6 cases again to check all the edges leaving the new node. If they return false, it means there is no simple path of length $k+1$ with its string of symbols in $L(Path)$. Combining the results from all $k+1$ level recursive calls, there exists no simple path of length $\leq k+1$ with its string of symbols in $L(Path)$. Hence, Theorem 2 is true when $Hopcount$ = $k+1$. □

4.1.4 Complexity Analysis

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

4.2 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 the sample policies and sample access requests that would require the access control decider to gather necessary information and crawl on the graph for the 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.

4.2.1 Datasets

When designing the experiments, we take into account 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 [10], 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.

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(o-worker)", 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 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.

4.2.2 Results

Figure 4.1 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 4.1 (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 [54]) 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.

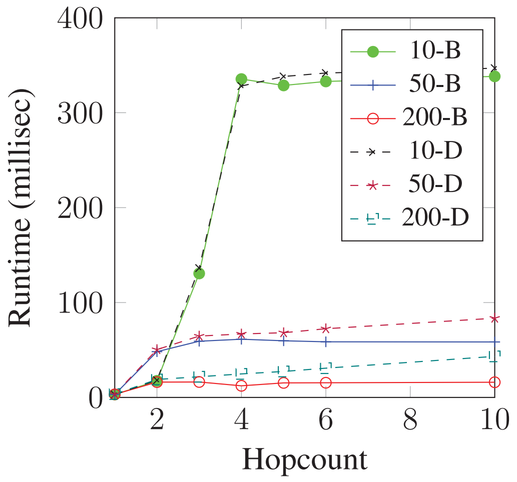
According to the classic idea of "six degrees of separation" and the findings of "small world experiment" [44,53], any pair of people are distanced by no more than six intermediate connections on average. A recent study by Backstrom et al [2] further indicates that the average distance on the current 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 4.1 (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 4.1 (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.

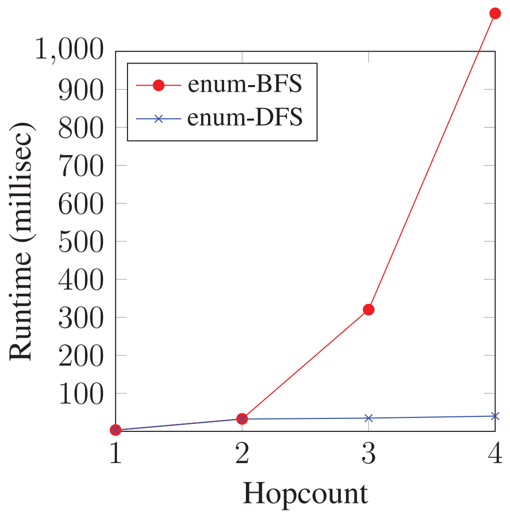
As hopcount increases, the time costs of the algorithms increase approximately in the magnitude of node degree, which match our expectation given in the complexity analysis.

Figure 4.2 represents a comparison of the performance of the two algorithms on graphs with different node degrees. In true-case scenarios, as shown in Figure 4.2 (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 (4.2 (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.

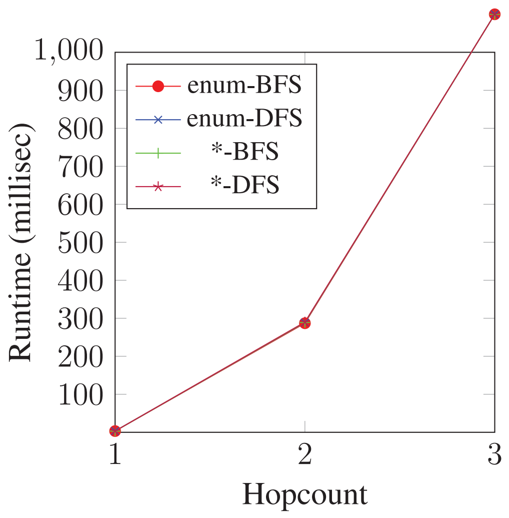
Our results indicate that both node degree and hopcount limit significantly affect the performance of the two algorithms. In some extreme cases (e.g., long enumeration paths, high density graph, etc.), searching a qualified path of 3 hops long may take very long time that the system and users cannot bear. 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 limit is barely seen 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 waitings for those extreme scenarios. Another important observation from our



(a) True-case scenarios: *-patterns



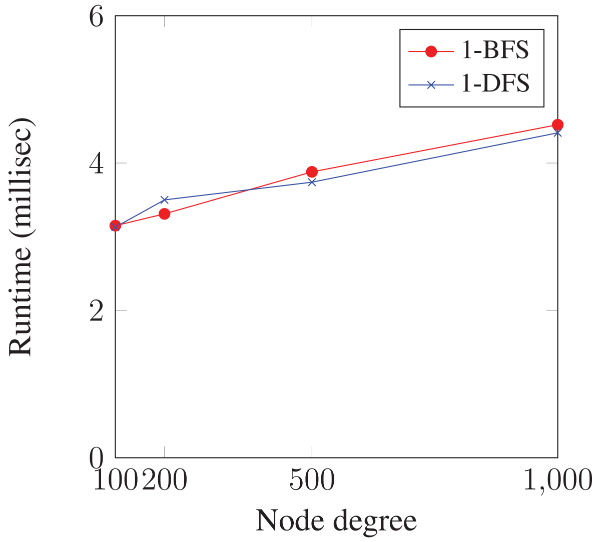
(b) True-case scenarios: enum-patterns



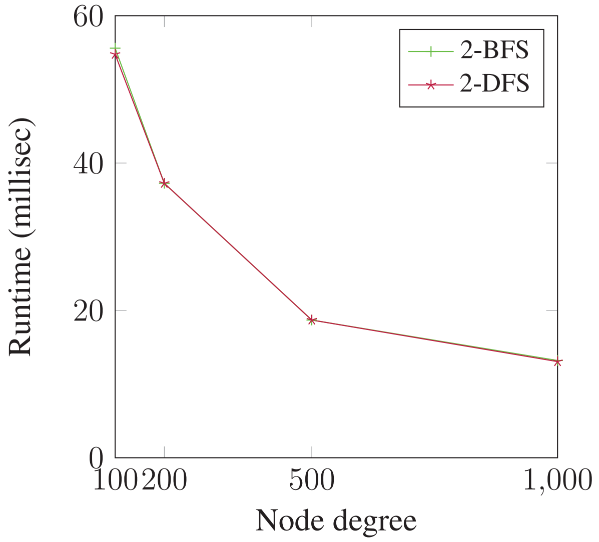
(c) False-case scenarios

Figure 4.1: Experiment 1: BFS vs DFS on hopcount

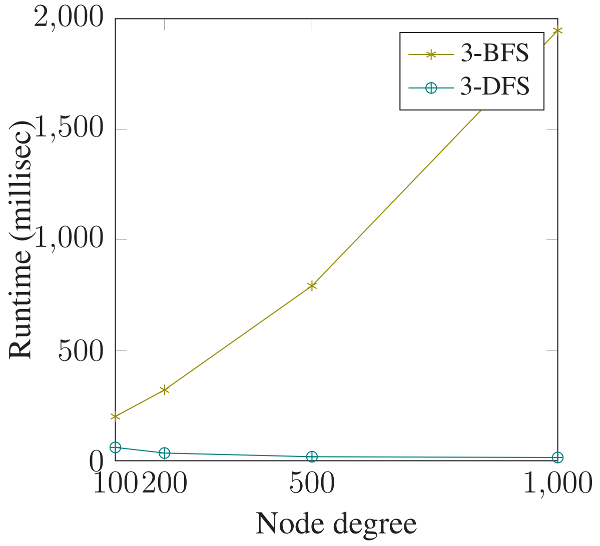
experiments is that although they have almost the same performance for 1 and 2-hop policies, DFS algorithm in general is likely to be more suitable for policies with intermediate hopcount values (e.g., 3, 4, 5, etc) than its BFS counterpart.



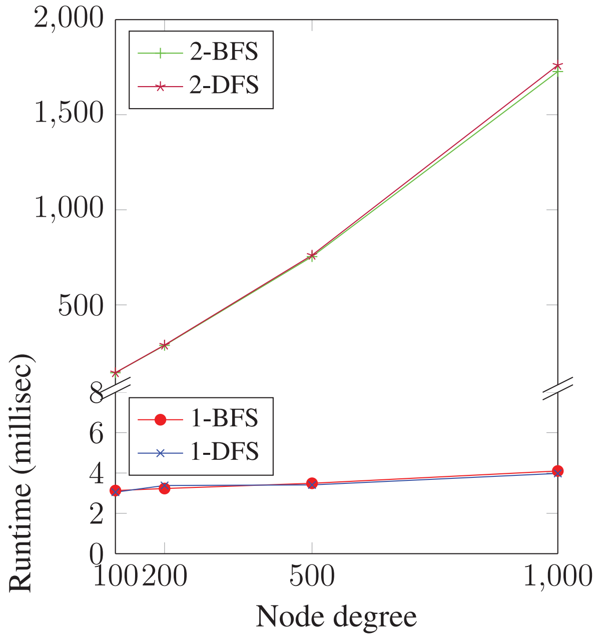
(a) True-case scenarios: hopcount 1



(b) True-case scenarios: hopcount 2



(c) True-case scenarios: hopcount 3



(d) False-case scenarios

Figure 4.2: Experiment 2: BFS vs DFS on node degree

Chapter 5: ATTRIBUTE-AWARE RELATIONSHIP-BASED ACCESS CONTROL

This chapter presents an attribute-aware ReBAC model that integrates attribute-based policies into the UURAC model, allowing finer-grained controls that are not available in ReBAC. We also provide an enhanced path-checking algorithm to determine the existence of the required attributes and relationships in order to grant access.

5.1 The $UURAC_A$ Model

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

5.1.1 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 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 [38, 49, 55]. 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

[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.

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 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.

5.1.2 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 5.1, 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 5.1: Attribute Quantifiers

$\forall$ [+m, -n]	All entities from the mth to the nth 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 mth to the nth, $m \leq n \leq h$
$\forall$ [-m, -n]	All entities from the mth last to the nth last, $h \geq m \geq n$
$\exists$ [+m, -n]	One entity from the mth to the nth last, $m + n \leq h$
$\exists$ [+m, +n]	One entity from the mth to the nth, $m \leq n \leq h$
$\exists$ [-m, -n]	One entity from the mth last to the nth 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 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.

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

5.1.3 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 Chapter 3. Table 5.2 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 5.3, we identify several different types of policies. 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 5.4 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.

Table 5.2: $UURAC_A$ 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 5.3: $UURAC_A$ 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	$\langle act, (r.typename, r.typevalue), graphrule \rangle$

Table 5.4: $UURAC_A$ Grammar for Graph Rules

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

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

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

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$_A$.

5.2 Path-Checking Algorithm

This section addresses the access evaluation of UURAC$_A$. UURAC [17] 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 section, we present a modified path-checking algorithm to incorporate an attribute-based policy evaluation on the fly during path finding process.

Algorithm 5.1 $AccessEvaluation(u_a, act, target)$

```
1: (Policy Collecting Phase)
2: if $target = u_t$ then
3:     $AUP \leftarrow u_a$'s policy for $act$, $TUP \leftarrow u_t$'s policy for $act^{-1}$, $SP \leftarrow$ system's policy for $act$
4: else
5:     $u_c \leftarrow owner(r_t)$, $AUP \leftarrow u_a$'s policy for $act$, $TRP \leftarrow u_c$'s policy for $act^{-1}$ on $r_t$, $SP \leftarrow$ system's
       policy for $act, r.type$
6: (Policy Evaluation Phase)
7: for all policy in $AUP$, $TUP$/$TRP$ and $SP$ do
8:     Extract graph rules ($start$, $path\ rule$) from policy
9:     for all graph rule extracted do
10:        Determine the starting node, specified by $start$, where the path evaluation starts
11:        Determine the evaluating node which is the other user involved in access
12:        Extract path rules $path\ rule$ from graph rule
13:        Extract each path spec $path$, $hopcount$ and/or attribute rule $attpolicy$ from path rules
14:        Simultaneously path-check each path spec and evaluate the corresponding attribute rule using Al-
           gorithm 5.2
15:        Evaluate a combined result based on conjunctive or disjunctive connectives between path specs
16: Compose the final result from the result of each policy
```

5.2.1 Access Evaluation Procedure

In UURAC$_A$, access requests can be evaluated as described in Algorithm 5.1. 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 later in Chapter 6, which is based on a disjunctive, conjunctive, or prioritized strategy.

5.2.2 Attribute-aware Path Checking Algorithm

The path-checking algorithm, as shown in Algorithm 5.2, 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. For ease of explanation, we only consider global attribute-based policies in this section.

Let us consider the example policy P1 in Section 5.1.3: $\langle profile\_access$, $(u_a, ((ff, 2)$: $\exists[+1, -1]$, $occupation(u)$ = "$student$", $count \geq 5))\rangle$. 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 5.2 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 5.2 launches the DFS traversal function $DFST()$, shown in Algorithm 5.3, 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 continues 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 5.2 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 5.2 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 in the previous chapter. 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.

Algorithm 5.2 $DFSPathChecker(G, path, hopcount, s, t, globalattpol)$

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

Algorithm 5.3 $DFST(u)$

```
1: if $d + 1 > hopcount$ then
2:     return FALSE
3: else
4:     for all $(v, \sigma)$ where $(u, v, \sigma)$ $in$ $G$ do
5:         switch
6:         case 1 $v \in currentPath$
7:         break
8:         case 2 $v \notin currentPath$ and $v = t$ and DFA with transition $\sigma$ is at accepting state
9:         if $v$ and $(u, v, \sigma)$ is within the range specified by quantifier then
10:            $attrList \leftarrow attrList.(ATTR(v), ATTR(u, v, \sigma))$
11:            if $f(ATTR(v), ATTR(u, v, \sigma)) = TRUE$ then
12:                $count \leftarrow count + 1$
13:                if $count \geq i$ then
14:                    $d \leftarrow d + 1$; $currentPath \leftarrow currentPath.(u, v, \sigma)$
15:                    $currentState \leftarrow$ DFA takes transition $\sigma$
16:                    $stateHistory \leftarrow stateHistory.(currentState)$
17:                    return TRUE
18:            else
19:                $attrList \leftarrow attrList \backslash (ATTR(v), ATTR(u, v, \sigma))$
20:          else
21:            $d \leftarrow d + 1$; $currentPath \leftarrow currentPath.(u, v, \sigma)$
22:            $currentState \leftarrow$ DFA takes transition $\sigma$
23:            $stateHistory \leftarrow stateHistory.(currentState)$
24:            return TRUE
25:          break
26:          case 3 $v \notin currentPath$ and $v = t$ and transition $\sigma$ is valid for DFA but DFA with transition $\sigma$ is not at accepting state
27:          break
28:          case 4 $v \notin currentPath$ and $v = t$ and transition $\sigma$ is invalid for DFA
29:          break
30:          case 5 $v \notin currentPath$ and $v \neq t$ and transition $\sigma$ is invalid for DFA
31:          break
32:          case 6 $v \notin currentPath$ and $v \neq t$ and transition $\sigma$ is valid for DFA
33:          $d \leftarrow d + 1$; $currentPath \leftarrow currentPath.(u, v, \sigma)$
34:          $currentState \leftarrow$ DFA takes transition $\sigma$
35:          $stateHistory \leftarrow stateHistory.(currentState)$
36:          if (DFST(v)) then
37:            return TRUE
38:          else
39:            $d \leftarrow d - 1$; $currentPath \leftarrow currentPath \backslash (u, v, \sigma)$
40:            $attrList \leftarrow attrList \backslash (ATTR(v), ATTR(u, v, \sigma)$
41:            $previousState \leftarrow$ last element in $stateHistory$
42:            DFA backs off the last taken transition $\sigma$ to $previousState$
43:            $stateHistory \leftarrow stateHistory \backslash (previousState)$
44:      return FALSE
```

Chapter 6: THE URRAC MODEL

In this chapter, we develop an access control model for OSNs that incorporates not only U2U relationships but also U2R and R2R relationships. The model also captures controls on users' administrative activities, and provides simple specifications of conflict resolution policies to resolve possible conflicts among authorization policies.

6.1 Beyond U2U Relationship-based Access Control

In this section, we discuss limitations of U2U relationship-based access control and build a taxonomy of user's access types in OSNs based on U2U, U2R and R2R relationships.

6.1.1 Limitation of U2U Relationship-based Access Control

In OSNs, users are encouraged to create profiles, add content onto their pages (e.g., photos, videos, blogs, status updates and tweets), and share these resource objects with other peers. OSNs offer their users various types of user interaction services, including chatting, private messaging, poking and social games. As OSN systems mature, various types of resources need to be protected, such as user sessions, relationships among users and resources, access control policies and events of users. As shown in Figure 6.1(a), users can launch access requests against both resources (e.g., view a photo or create an access control policy) and users (e.g., invite another user to a game or poke another user).

Social graph represents a global mapping of all individual users and how they are connected in an OSN, where user is a node and a relationship between users is an edge. Access control in most existing OSNs are based on the topology of the social graph, so-called relationship-based access control. Typically, granting access permission to an accessing user is subject to the existence of a particular relationship or a particular sequence of relationships between the accessing user and the target user/resource owner, and access control policies are specified in terms of such U2U relationships. When a user requests access to a resource, current OSNs rely on an implicit

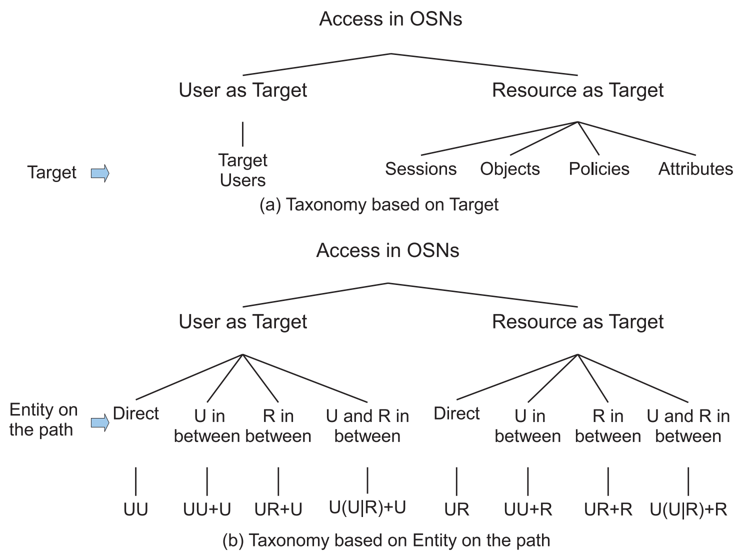


Figure 6.1: Access in OSNs

relationship, namely ownership, between the resource and its owner, hence the authorization of such U2R access is still based on the underlying U2U relationships.

However, due to the various functionality offered by today's OSNs, there exist several different types of relationships between users and resources in addition to ownership. Consider an example where Bob posts a photo that contains Alice and Carol's images in it and tags them. OSNs usually allow only the owner Bob to have control on who can view the photo, regardless of whether or not Alice and Carol may wish to release their images. To enable Alice and Carol control capability on the photo, their relationships with the photo, which is not ownership, should be considered for authorization purposes. After the photo has been shared by Bob's friends several times, more and more users from different neighborhoods in the network come to view the photo and comment on it. When Dave reads through all the comments in Bob's photo and becomes curious about another user Eve who has commented recently, he decides to poke her to say hello. In this case, Dave and Eve are connected through the photo, not through another user (such as the owner of the photo Bob). Also, users may share or like the blog posts or videos posted by others, and gain the ability to determine how the shared/liked copy of the original content or the fact of sharing and liking activities can be seen by others. Consider another scenario where Betty finds a weblink originally posted by Ed interesting and then shares it with her friends. From her activity, she acquires the ability to decide how the weblink can be available to others. As users get increasingly involved in these activities in OSNs, current U2U relationship-based access control mechanism is not able to offer the appropriate control and requires extensions to bring U2R and R2R relationships into consideration.

In recent years, Facebook has gradually expanded the idea of social graph to so-called Open Graph as it launches new services such as photos and places, and includes these in the graph over time. Recently even further extensions to incorporate arbitrary activities and objects are being pushed so as to codify user behaviors effectively. These recent trends in commercial OSNs strengthen our belief that it is useful to include resources, such as objects and activities, in the social graph. By means of such an extended social graph, users and all of the resources related to users are interconnected through U2U, U2R, and even R2R relationships, allowing stronger expressive power of relationship-based access control policies.

6.1.2 Taxonomy of Access Scenarios

As shown in Figure 6.1(b), in OSN, a user can access other users (user as a target) or resources (resource as a target). By means of U2U, U2R and R2R relationships, an accessing user and a target user can have a direct relationship or indirect relationships with user(s) in between, resource(s) in between or user(s) and resource(s) in between. Likewise, an accessing user and a target resource can also be characterized in terms of the entities on the relating path.

In the first two cases of accessing a target user, there is no resource involved. An accessing user should either have a particular direct U2U relationship (shown as UU) or a particular sequence of U2U relationships (shown as UU+U)[1] with the target user. Examples of such access to a target user are that Alice's direct friends can poke her, and Bob's friends of friends can request friendship invitation to him. If resources are introduced onto the path between the accessing user and the target user, it brings in U2R and R2R relationships and leads to other possible combinations of

[1]Note, "+" denotes one or more occurrences of the applicable entity (U, R or U|R) on the path.

relationships. For instance, “users who are tagged in the same photo can visit each other’s profile even though they are not friends” is a typical UR+U example that can be found in OSNs, in which case the photo actually links two unconnected users together. We can make even more complicated policies by connecting users through both users and resources (shown as U(U|R)+U). In the previous example, friends of one of the tagged users may be able to access another tagged user through their mutual friend and the photo.

Similarly, a user may access a resource that directly relates to her (shown as UR), or may find a resource through one or more users in the network (shown as UU+R). Most of the current commercial OSNs and the prior work [11,15,23–25] deal with these two cases with an implicit assumption of the existence of “own” relationship. When a user requests an access against resource, the system checks if there is a qualified relationship between the accessing user and the resource owner, and then determines the authorization. We believe that it is useful to distinguish different types of U2R relationships, such as “tag”, “share” and “like”, in the policy specifications rather than relying only on “own”. Incorporating R2R relationships and connecting resources on the path enables UR+R and U(U|R)+R cases, so that users may be able to access some resources that are connected to users’ related resources. For example, if user Alice is tagged in one of Bob’s photo, then Alice may get the privilege to view other photos in the same album without being Bob’s contact of any type.

The above discussion indicates that allowing U2R and R2R relationships in access control gives users more complete and flexible expressive power than the currently prevailing U2U-only approaches.

6.2 The URRAC Model Components and Characteristics

In this section, we identify the components of the proposed relationship-based access control model for OSNs, and discuss crucial characteristics of the model.

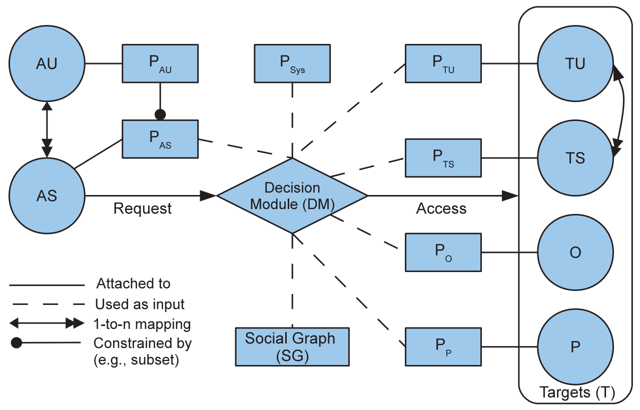


Figure 6.2: The URRAC Model Components

6.2.1 Components

Figure 6.2 shows a conceptual diagram for the relationship-based access control model. It comprises six categories of basic components: users, sessions, resources, policies, social graph and decision module, as discussed below.

Users. A user represents a human being registered in an OSN system to whom authorization may be granted. Users maintain relationships with each other, own a number of resources, and perform various kinds of actions against resources and users in the system. Users can be identified as accessing users (AU) and target users (TU), based on the roles they play with respect to access. Accessing users are users who perform certain types of access against targets, carrying authorization policies (Accessing User Policies or P_{AU}). Target users are users against whom access is performed. Target users also carry authorization policies (Target User Policies or P_{TU}).

Sessions. A session is an active instance of a user who has logged into the OSN. Accessing users perform access through sessions (AS), while target users may or may not have a session instance (TS) at the time of access. In other words, some accesses can be placed only when a target user is online (e.g., chatting) while other accesses do not require this and can be placed on a target user who is not logged in at the time of the accesses (e.g., poking). The user-session distinction allows sessions to have different policies and attributes from those of the corresponding user by partially inheriting them possibly along with some other policies and attributes. This is depicted in Figure 6.2 as “constrained by”. A user can have multiple sessions with differing access control policies, while a session is only associated with a single user. In general, all sessions are considered objects and can be created, suspended, or resumed by another session or by the user. Sessions are also called subjects in the access control literature. The term session was introduced in the role-based access control literature and has become widely used in that context.

Resources. Resources are non-user targets to be protected in access. They include target user’s sessions (TS), objects (O) users shared in the system as well as access control policies (P). Because resources are not human beings, access control policies for resources are defined by users who possess the corresponding administrative privileges.

Policies. Access control policies are a set of rules that govern the ability of sessions (subjects) to access targets. Like in many computer systems, OSNs allow the system security administrators to define a central policy that is guaranteed to be enforced for all users and resources in the system, called *system-specified policy* (P_{Sys}). Additionally, users have the ability to express own preferences with respect to themselves or their related users and resources. *Accessing user policy* (P_{AU}), *target user policy* (P_{TU}), *accessing session policy* (P_{AS}), *object policy* (P_O), *policy for policy* (P_P) are defined by users and applied to accessing users, target users, accessing sessions, objects and policies, respectively. System-specified policies consist of two types of policies, namely *authorization policies* and *conflict resolution policies*. Authorization policies allow the system and users to specify who is authorized to exercise which action on the user or resource, while conflict resolution policies specify how conflicts among authorization policies from multiple parties are to be solved. For the purpose of this work we assume that conflict resolution policies are specified entirely by the system administrators as part of P_{Sys}.

Social Graph (SG). Social graph denote connections among users and resources in the system. Although “relationships” on the graph usually refer to relationships among individual users in many OSN systems in practice and theory, they also can include U2R relationships and R2R relationships as we have argued above. U2U relationships are typically represented by the social graph. We extend the social graph to incorporate U2R relationships and R2R relationships as well, thus forming a network of users, sessions, objects as well as their access control policies.

Decision Module (DM). The access decision module in Figure 6.2 consolidates all the necessary policies from P_{AS}, P_{TU}, P_{TS}, P_O, P_P and P_{Sys} as well as the relationships on the social graph, and makes a decision at the time of request. Access decision module can handle potential policy conflicts by consulting conflict resolution policies in P_{Sys}.

6.2.2 Characteristics

We identify three essential characteristics that need to be addressed by OSN access control models, as follows.

Policy Individualization. As identified in [45,46], unlike in traditional access control systems, OSNs allow individual users to express their own preferences over access to the content rather than having a single system-wide access control policy defined by the system security administrator. Moreover, users other than the resource owner are also able to configure policies for user and resource related to them. For example, parents of a child want to prescribe a boundary within which their child might perform access, and Alice wants to block her colleagues from seeing the party pictures which contain her image. The system needs to collect all of the related individual policies along with the system-specified policies for making access control decisions.

Policy Administration. In OSN, policy administration becomes very important since allowing individual users to specify policies requires the OSN to ensure that only the right users are authorized to specify policies. Our model enables users to specify policies for other users and resources as long as they meet the relationship requirement stated in the policies for the target policy. For example, the system-specified policy may allow users who have “own” or “tag” relationships with the resource to set the policy for that resource. Then the owner or tagged user can control the resource’s policy and later modify it to enable or disable who can access the resource.

User-session Distinction. We know that a session is a process in execution on behalf of a user. A user can have multiple sessions with different sets of privileges by creating different degrees of access control policies with the original user’s. The user-session distinction facilitates better security and privacy control by minimizing a session's privilege to an adequate level. It becomes especially useful in OSN environments as more and more smart devices and location-based applications are introduced into OSN world. Users logged in from different devices may have distinct access control policies and thus distinct privileges. Users with location-based services enabled may be offered extra functionality than ordinary users are. Much of the current literature in OSN access control does not distinguish a session from a user. We believe differentiating user and session is crucial for effective access control in OSNs.

6.3 The URRAC Model

In the following, we formally define an access control model for OSNs, expressing authorization policies and conflict resolution policies in terms of relationships existing among users and resources in the system.

6.3.1 Model Definition

We begin by identifying each component of the model.

U is the current set of users, including accessing users (AU) and target users (TU). Associated with each user is a collection of sessions. S is the current set of sessions, which is composed of accessing sessions (AS) and target sessions (TS). R is the set of resources, including target sessions (TS), objects (O) and access control policies (P). We refer to target users and resources as targets, which are the targets of access.

We write $ACT = \{act_1, act_2, \ldots, act_n\}$ which is the set of OSN supported actions, denoting the access modes a user or a session can execute in the system. Each action is defined in active form with accessing user or session as the actor and target users and/or resources as targets. For each action act_i the passive form act_i^{-1} represents the action from the target's perspective.

The overall set of policies P in the OSN is categorized as follows.

- $P_{AU} \subseteq P$, $P_{TU} \subseteq P$, $P_{AS} \subseteq P$, $P_{TS} \subseteq P$, $P_O \subseteq P$, $P_P \subseteq P$ and $P_{Sys} \subseteq P$ are authorization policies for accessing user, target user, accessing session, target session, objects, policies and system-specified policies, respectively.

- $AP_{Sys} \subseteq P_{Sys}$ and $CRP_{Sys} \subseteq P_{Sys}$ represent system-specified authorization policies and conflict resolution policies, respectively.

The first two user-specified policies P_{AU} and P_{TU} are associated with and specified by the corresponding users, while rest of the user-specified policies are specified by users but associated with the resource or another user. The user who actually specifies a policy for other user or resource is called the controlling user (CU), who has a certain type of relationship with the user/resource to whom the policy applies.[2] Accessing session policies are partially inherited from its corresponding accessing user's policies, but may have additional content specific to the session, such as location. System-specified policies, on the other hand, are expressed by the system and applied to all relevant activities across the system. While a resource is always the target of an access, a user, on the other hand, can participate both as an accessing user or a target user in an access with different access control requirements. Hence, the distinction between the active and passive forms of an action becomes significant. We write act and act^{-1} as the active and passive form of an action act, respectively. Accessing user policies, accessing session policies and system-specified policies are indexed by act, and target user policies, target session policies, object policies and policies for policy are indexed by act^{-1}. Authorization policies specified by multiple users may possibly give conflicting results for a requested access. We introduce system-defined conflict resolution policies (CRP_{Sys}) to make unambiguous decisions for authorization policies specified by multiple parties with conflicting interests. The specification of policies is discussed below in subsection 6.3.2.

As depicted in Figure 6.3, we abstract an OSN as a directed labeled simple graph, where each vertex represents a user or a resource, whereas each edge corresponds to a relationship among users and resources. The social graph of an OSN SG is formally denoted as a triple $\langle V, E, \Sigma \rangle$:

- $V = U \cup R$ is a finite set of vertices on the graph, representing registered users and their resources in the system.

[2] In some cases, accessing a user/resource is subject to the relationships existing between the accessing user and the controlling user [17]. See Example 3 in Section 6.4.

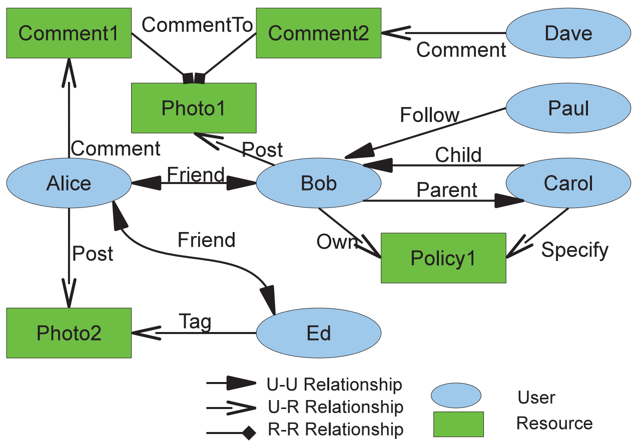


Figure 6.3: A URRAC Sample Social Graph

- $\Sigma = \Sigma_{u\_u} \cup \Sigma_{u\_r} \cup \Sigma_{r\_r} = \{\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_n^{-1}\}$, denotes the set of relationship type specifiers in the system. Each relationship type specifier σ is represented by a string of characters. Given a relationship type $\sigma_i \in \Sigma$, we write the inverse of the relationship $\sigma_i^{-1} \in \Sigma$. Relationships are further divided into three categories: U2U (U_U), U2R (U_R) and R2R (R_R) relationships. We express this formally as follows.

  - U_U relationships: $U \times \Sigma_{u\_u} \times U$,
  - U_R relationships: $U \times \Sigma_{u\_r} \times R$ or $R \times \Sigma_{u\_r} \times U$,
  - R_R relationships: $R \times \Sigma_{r\_r} \times R$,

  where $\Sigma_{u\_u} \subseteq \Sigma$, $\Sigma_{u\_r} \subseteq \Sigma$ and $\Sigma_{r\_r} \subseteq \Sigma$.

- $E = E_{u\_u} \cup E_{u\_r} \cup E_{r\_r}$, where $E_{u\_u} \subseteq U \times \Sigma_{u\_u} \times U$, $E_{u\_r} \subseteq (U \times \Sigma_{u\_r} \times R) \cup (R \times \Sigma_{u\_r} \times U)$, $E_{r\_r} \subseteq R \times \Sigma_{r\_r} \times R$, represents the existing relationships among users and resources in the system.

Note that E is defined as directed, since not all of the relationships in OSNs are mutual. For every $\sigma_i \in \Sigma$, there is $\sigma_i^{-1} \in \Sigma$ representing the inverse of relationship type σ_i. Although not explicitly shown on the social graph, we assume the original relationship and its inverse twin always exist simultaneously. Given a vertex $v_1 \in V$, a user $v_2 \in V$ and a relationship type $\sigma \in \Sigma$,

Table 6.1: URRAC 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. For example, $friend*$ means direct or indirect friends of a user or user herself. $\Sigma*$ is $\bigcup_{i=0}^{\infty} \Sigma^i$, denoting the node itself or nodes with any connection on social graph.
Plus (+)	Denotes concatenating σ one or more times. Similarly for $\Sigma+$.
Question Mark (?)	Represents occurrences of σ zero or one time. $co-worker \cdot friend?$ means only co-worker or co-worker's direct friends can access. Similarly for $\Sigma?$.
Square Bracket ([])	Contains a path rule: a sequence of relationship specifiers with an indicated hopcount limit.
Double Square Bracket ([[]])	Denotes skipping of the path rule contained. The meaning of the skipping feature is discussed in the text.
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.

a relationship (v_1, v_2, σ) says that there exists a relationship of type σ originating from vertex v_1 and terminating at v_2. There always exists an equivalent form (v_2, v_1, σ^{-1}) at the same time.

It remains to formally define the concept of access request.

- (s, act, T) represents an access request, where $s \in S$ indicates the accessing session, $act \in ACT$ denotes the requested action and $T \subseteq (2^{TU \cup R} - \emptyset)$ gives a non-empty set of target users and resources;

The cardinality and types of the targets are determined by the action.

6.3.2 Policy Specifications

The notations used in the policy specification language are defined in Table 6.1, familiar from typical regular expression notation with the addition of hopcount limits and skipping. As described earlier, there are several types of access control policies: accessing user policy, accessing

Table 6.2: URRAC Authorization Policy Representations

Accessing User Policy	$\langle act, graphrule \rangle$
Accessing Session Policy	$\langle act, graphrule \rangle$
Target User Policy	$\langle act^{-1}, graphrule \rangle$
Target Session Policy	$\langle act^{-1}, graphrule \rangle$
Object Policy	$\langle act^{-1}, graphrule \rangle$
Policy for Policy	$\langle act^{-1}, graphrule \rangle$
System Policy for User	$\langle act, graphrule \rangle$
System Policy for Resource	$\langle act, o.type, graphrule \rangle$ where $o.type$ is optional

session policy, target user policy, target session policy, object policy, policy for policy, and system-specified policy. Here, system-specified policies comprise authorization policies and conflict resolution policies. System-specified authorization policy allows the system to express access control requirements that apply to the entire set of users or resources, while system-specified conflict resolution policy deals with the potential conflicts of interest among the user-specified authorization policies.

Authorization Policy (AP). Authorization policies are modeled in different formats as shown in Table 6.2. *Accessing User Policy* and *Accessing Session Policy* are represented as a pair $\langle act$, $graph\ rule \rangle$ and regulate how an access requester in access can behave. Here, act indicates the requested action while $graph\ rule$ denotes the access rule based on social graph. *Target User Policy*, *Target Session Policy*, *Object Policy* and *Policy for Policy* are about how others can perform access on the target, so they use passive form act^{-1} instead of act because the target is always the entity to be accessed, whereas $graph\ rule$ has the same meaning as in the previous policies. *System-specified policies* do not differentiate the active and passive form of an action, since they are not attached to a particular entity in action. However, it is likely that we need to refine the scope of the objects to which the policies apply, thus we bring object types $o.type$ into policy specifications. Hence, system-specified policies are defined in two formats: $\langle act, graph\ rule \rangle$ for user and $\langle act, o.type, graph\ rule \rangle$ for resource. Note that $o.type$ is optional and used only if target is an object.

Table 6.3 defines the grammar for the graph rules, based on which each graph rule specifies a $startingnode$ and a $pathrule$. Starting node stands for the user or resource where the policy evaluation begins, which can be the accessing user, the controlling user or the target. A path rule is composed of one or more path specs, with each spec stating the required sequence of relationship types and the corresponding hopcount limit for the sequence. Users are allowed to specify a more complicated and fine-grained policy for an action against a target by connecting multiple path specs with conjunctive connective "$\wedge$" and disjunctive connective "$\vee$". Also, negation "$\neg$" over path specs is used to imply the absence of the specified pattern of relationship types and hopcount limit as authorization requirements. Each path spec is denoted as a tuple ($path$, $hopcount$), where $path$ represents from the starting node a sequence of relationship type expressions segmented by "[]" or "[[]]" with local hopcounts, denoting the pattern of relationship types required to grant authorization, whereas $hopcount$ describes the maximum distance between the accessing user and the target on the graph. Within each path segment there is a local $hopcount$ defining the maximum distance requirement for the particular piece of relationship type expression. In some policies, $path$ can be left blank to indicate only the starting node can access. With the use of U2R and R2R relationships, the distance between two users on the graph may be growing significantly. The notion of distance in U2R and R2R relationships is somewhat different from that of U2U relationships. For example, suppose Alice and Bob are friends and Bob owns a photo and Dave is tagged to the photo. Here, the distance between Alice and Bob is 1 and distance of Bob and Dave is 2. While the distance between Alice and Dave is 3, this combined distance is not as meaningful as individual U2U and U2R/R2R distances. Therefore, we may want to omit the distance created by resources by introducing the "skipping" notation, denoted "[[]]", which means that the local hopcount stated inside "[[]]" will not be counted in the global hopcount. For instance, in the path rule "([$f*$,3][[$c*, 2$]], 3)", the local hopcount 2 for $c*$ does not apply to the global hopcount 3, thus allowing $f*$ to have up to 3 hops.

Conflict Resolution Policies (CRP). Due to the nature of policy individualization, multiple policies applicable to authorization of an access request may result in decision conflicts in many scenarios. We assume that policies specified by the system will always be unambiguous so there

Table 6.3: URRAC Grammar for Graph Rules

$GraphRule \rightarrow$ “(”$StartingNode$“,”$PathRule$“)”
$PathRule \rightarrow PathSpecExp\ |PathSpecExp\ Connective\ PathRule$
$Connective \rightarrow \vee\ |\wedge$
$PathSpecExp \rightarrow PathSpec\ |$“$\neg$”$PathSpec$
$PathSpec \rightarrow$ “(”$Path$“,”$HopCount$“)” $|$“(”$EmptySet$“,”$HopCount$“)”
$HopCount \rightarrow Number$
$Path \rightarrow [$“[”$TypeSeq$“]”$|$“[”$TypeSeq$“,”$HopCount$“]” $|$“[[”$TypeSeq$“,”$HopCount$“]]”$]+$
$EmptySet \rightarrow \emptyset$
$TypeSeq \rightarrow TypeExp\ \{$“$\cdot$”$TypeExp\}$
$TypeExp \rightarrow TypeSpecifier\ |TypeSpecifier\ Quantifier$
$StartingNode \rightarrow u_a|u_c|t$
$TypeSpecifier \rightarrow \sigma_1|\sigma_2|\ldots|\sigma_n|\sigma_1^{-1}|\sigma_2^{-1}|\ldots|\sigma_n^{-1}|\Sigma$ where Σ=$\{\sigma_1,\sigma_2,\ldots,\sigma_n,\sigma_1^{-1},\sigma_2^{-1},\ldots,\sigma_n^{-1}\}$
$Quantifier \rightarrow$ “$*$”$|$“?”$|$“$+$”
$Number \rightarrow [0-9]+$

are no conflicts within P_{Sys}. Conflict resolution policies are then responsible for interpreting how the potential policy conflicts within each category of P_{AS}, P_{TU}, P_{TS} P_O and P_P can be resolved in terms of the precedence or connectives over relationship types. Relationship precedence is used to produce a collective result from multiple policies specified by users with different relationships to the policy holder. To resolve conflicts, we consider three simple and intuitive approaches: disjunctive, conjunctive or prioritized. In a disjunctive approach, satisfying any of the involved policies guarantees access. While for some sensitive contents, it is more meaningful to conjunctively query all the involved policies so that authorization is only allowed by satisfying the requirements of every policy. Whereas, if parental control is facilitated, parents' policies always get priority over children's policies. We write $\vee$, $\wedge$ and $>$ to denote disjunction, conjunction and prioritized order between relationship types, whereas the symbol @ represents a special relationship "null" that denotes "self".

Let us consider some examples of conflict resolution policies as follows.

$\langle read^{-1}, (own \wedge tag)\rangle$

Both the owner's and the tagged users' "$read^{-1}$" policies over the photo are honored.

$\langle friend\_request, (parent > @)\rangle$

When child attempts friendship request to someone, parents' policies get precedence over child's own will.

$\langle share^{-1}, (own \vee tag \vee share)\rangle$

A weblink is sharable if either the original owner, or any of the tagged users or shared users allows.

While evaluating an access request, if the decision module discovers two or more opposing policies from the same policy set, it looks up the corresponding conflict resolution policy for the action to determine how to reconcile the conflict. Note that conflict resolution policies only apply to the conflicting policies from the same policy category, the decision module still takes the conjunction of $P_{AS}, P_{TU}, P_{TS}, P_O, P_P$ and AP_{Sys} to make a final decision.

6.3.3 Access Evaluation Procedure

Algorithm 6.1 specifies how the access evaluation procedure works. After a session of a user s requests an act against target(s) T, say (s, act, T), the access decision module first collectively assembles s's session policy about act, a collection of act^{-1} policies from each target in T and the system-wide policies over act and object type, if target is an object. Once all the necessary policies are collected, the decision module extracts each path spec from the graph rules, determines the starting node and the evaluating node, and runs path checking for each path spec using the algorithm similar to one introduced in Chapter 4. The evaluation result of each policy is derived from combining the result of each path spec in the policy. Due to possible conflicts between the results of multiple policies, the decision module looks up the system-defined conflict resolution policies to resolve conflicts and compose the final result, and then determines the access.

Algorithm 6.1 $AccessEvaluation(s, act, T)$

```
1: (Policy Collecting Phase)
2: $s.P_{AS}(act) \leftarrow s$'s policy for $act$
3: if $(T \cap TU) \neq \emptyset$ then
4:     $T.P_{TU}(act^{-1}) \leftarrow \bigcup_{i=1}^{|T \cap TU|} tu_i.P_{TU}(act^{-1})$
5: if $(T \cap TS) \neq \emptyset$ then
6:     $T.P_{TS}(act^{-1}) \leftarrow \bigcup_{j=1}^{|T \cap TS|} ts_j.P_{TS}(act^{-1})$
7: if $(T \cap O) \neq \emptyset$ then
8:     $T.P_O(act^{-1}) \leftarrow \bigcup_{k=1}^{|T \cap O|} o_k.P_O(act^{-1})$
9: if $(T \cap P) \neq \emptyset$ then
10:    $T.P_P(act^{-1}) \leftarrow \bigcup_{l=1}^{|T \cap P|} p_l.P_P(act^{-1})$
11: if $(T \cap O) \neq \emptyset$ then
12:    $P_{Sys}(act) \leftarrow \bigcup_{k=1}^{|T \cap O|} P_{Sys}(act^{-1}, o_k.type)$
13: else
14:    $P_{Sys}(act) \leftarrow$ system's policy for $act$
15: (Policy Evaluation Phase)
16: for all policy in $s.P_{AS}(act), T.P_{TU}(act^{-1}), T.P_{TS}(act^{-1}), T.P_O(act^{-1}), T.P_P(act^{-1})$ and $P_{Sys}(act)$
    do
17:    Extract graph rules ($start$, $path\ rule$) from policy
18:    Get the controlling user $u_c$, if the policy is not specified by $s$ or any $t \in T$
19:    for all graph rule extracted do
20:        Determine the starting node, specified by $start$, where the path evaluation starts
21:        if graph rule is extracted from $s.P_{AS}(act)$ and $P_{Sys}(act)$ then
22:            if $start = s$ then
23:                $u_c$ and every $t \in T$ becomes the evaluating node
24:            else
25:                every $t \in T$ becomes the evaluating node
26:        else
27:            $s$ becomes the evaluating node
28:        Extract path rules $path\ rule$ from graph rule
29:        Extract each path spec $path$, $hopcount$ from path rules
30:        Path-check each path spec for each pair of starting and evaluating node
31:        Evaluate a combined result based on conjunctive or disjunctive connectives between path specs
32: Compose the final result from the result of each policy using $CRP_{Sys}$
```

6.3.4 Hopcount Skipping

According to the classic idea of "six degrees of separation" and the results of "small world experiment" [44, 53], any pair of persons are distanced by about six people on average. A recent study by Backstrom et al [2] further indicates that on the current social graph of Facebook, the average distance has shrunk to 4.74. Therefore, the network of U2U relationships is characterized by short path lengths, and the hopcount limit in a practical policy is not likely to be a large number. In contrast, U2R and R2R relationships may exhibit a different characteristic. For example, comment may be followed up by a sequence of comments, which may take a long journey for the author of the first comment to reach the author of the last comment. For this and similar cases, we introduce the "skipping" of hopcount limit of resource-related relationships, which differentiates the global hopcount limit on U2U relationships only from the possible long distance of the resource-related relationships that are found in two entities involved in request.

6.4 Use Cases

Given the social graph depicted in Figure 6.3, below we show how access control of these examples can be realized within the model.

Example 1: Run into a new acquaintance in a photo. *Alice and Dave are strangers. Dave realizes that Alice and him both commented on Bob's photo, so he decides to poke her to say hello:*

$$(Dave, poke, Alice)$$

We need the following policies to determine authorization:

- $Dave$'s $P_{AS}(poke)$:
  $\langle poke, (u_a, ([Comment][[CommentTo{\cdot}CommentTo^{-1}, 2]][Comment^{-1}], 2))\rangle$
- $Alice$'s $P_{TU}(poke^{-1})$:
  $\langle poke^{-1}, (t, ([Comment][[CommentTo{\cdot}\, CommentTo^{-1}, 2]][Comment^{-1}], 2))\rangle$
- $P_{Sys}(poke)$: $\langle poke, (u_a, ([\Sigma_{u\_r}][[\Sigma_{r\_r}*, 2]][\Sigma_{u\_r}], 2))\rangle$

The comments from Alice and Dave are connected through Bob's photo with two R2R relationships. Dave's policy says that he is free to poke his fellow commenter, while Alice allows her fellow commenter to poke her. The system facilitates many kinds of participating users (e.g., comment, like, share, etc.) to poke each other.

Example 2: View a photo where a friend is tagged. *Bob and Ed are friends of Alice, but not friends of each other. Alice posted a photo and tagged Ed on it. Later, Bob sees the activity from his news feed and decides to view the photo:*

$$(Bob, read, Photo2)$$

In this example, Bob is trying to access a resource through his friend Alice. Whether his request can be granted or not depends on the corresponding policies from himself, the target resource and the system.

- Bob's $P_{AS}(read)$: $\langle read, (u_a, ([\Sigma_{u\_u}*, 2][[\Sigma_{u\_r}, 1]], 2))\rangle$
- $Photo2$'s $P_O(read^{-1})$ by $Alice$: $\langle read^{-1}, (t, ([post^{-1}, 1][friend*, 3], 4))\rangle$
- $Photo2$'s $P_O(read^{-1})$ by Ed: $\langle read^{-1}, (u_c, ([friend], 1))\rangle$
- $AP_{Sys}(read)$: $\langle read, (u_a, ([\Sigma_{u\_u}*, 5][[\Sigma_{u\_r}, 1]], 5))\rangle$
- $CRP_{Sys}(read)$: $\langle read^{-1}, (own > tag)\rangle$

Bob, as the access requester, allows himself to read any resource that has a direct relationship with his contacts within two hops. Note that "[[]] indicates that the local hopcount "1" is not counted in the global hopcount limit "2". Alice is the original owner of the photo and Ed's image is on it, so based on our default policy, both of them are able to express their own preferences on how the photo should be exposed to others. Alice decides to share the photo with all her direct and indirect friends within three hops, while Ed prefers to keep his privacy and only wants his direct friends to see it. The system, on the other hand, specifies a more liberal rule to promote sharing that allows a user to access resource that relate to his contacts within five hops. We notice that Alice and Ed's authorization policies are apparently in conflict, which needs to be resolved. $CRP_{Sys}(read)$ says that owner's policy takes precedence over tagged user's, so the decision module will ignore Ed's policy and only consider Alice's policy. A system may configure $CRP_{Sys}(read)$ with conjunction or disjunction of the owner's and tagged users' policies for different decisions.

Example 3: Friend recommendation. *Alice is a friend of Bob, Paul follows Bob, while Alice and Paul are strangers. Bob would like to recommend Alice and Paul to be friends:*

$$(Bob, suggest\_friend, Alice, Paul)$$

Policies applied to this example are shown as follows:

- Bob's $P_{AS}(suggest\_friend)$: $\langle suggest\_friend, (u_a, ([\Sigma_{u\_u}*], 2))\rangle$
- $Alice$'s $P_{TU}(suggest\_friend^{-1})$: $\langle suggest\_friend^{-1}, (t, ([friend], 1))\rangle$
- $Paul$'s $P_{TU}(suggest\_friend^{-1})$: $\langle suggest\_friend^{-1}, (t, ([friend*], 2))\rangle$
- $P_{Sys}(suggest\_friend)$: $\langle suggest\_friend, (u_a, ([\Sigma*], 2)) \wedge (t, ([\Sigma*], 2))\rangle$

The access request contains two targets Alice and Paul, so we need target user policies from both of them. Bob can suggest friends for his contacts within two hops. Alice welcomes friend recommendation from her direct friends, while Paul allows his friends of friends to do that. The system-specified policy is more liberal, allowing users with any relationship of two hops to be able to suggest friends (e.g., two people who commented on the same photo).

Example 4: Parental control of policies. *The system features parental control such as allowing parents to configure their children's policies. The policies are used to control the incoming or outgoing activities of children, but are subject to the parents' will. For instance, Bob's mother Carol requests to set some policy, say $Policy1$ for Bob:*

$$(Carol, specify\_policy, Policy1)$$

The following policies are used to make access decision:

- $Carol$'s $P_{AS}(specify\_policy)$: $\langle specify\_policy, (u_a, ([own], 1) \vee ([child \cdot own], 2))\rangle$
- $Policy1$'s $P_P(specify\_policy^{-1})$ by Bob: $\langle specify\_policy^{-1}, (t, ([own^{-1}], 1))\rangle$

- $P_{Sys}(specify\_policy)$: $\langle specify\_policy, (u_a, ([own], 1) \vee ([child \cdot own], 2))\rangle$
- $CRP_{Sys}(specify\_policy)$: $\langle specify\_policy, (parent \wedge @)\rangle$

Carol's policy offers her the ability to define her and her child's policies. Bob only allows himself to manage his own policies. The system enables parental control with the child's consent, so that parents can control their children's policies.

6.5 Related Works on Policy Conflict Resolution

There is substantial literature on conflict resolution of access control policies, especially in distributed systems, database systems and collaborative environments. Simultaneous presence of conflicting policies can be resolved by various strategies, such as permissions-take-precedence [36,37], denials-take-precedence [7, 36, 37], specificity precedence [6, 20], recency precedence, strong authorization overriding weak authorization [5, 6, 48], or explicit specification of policy priority [3, 19, 50], etc. Most conflicts discussed in this literature are conflicts between positive and negative authorizations (permissions vs. prohibitions) typically arising due to generality or specificity of the applicable policy in a hierarchy. However, in OSNs possible policy conflicts will likely arise due to policies specified by distinct users carry contrasting authorization.

In OSN systems, as long as each user can specify individual policies, policy conflicts become inevitable. [51] applied game theory to a solution for collective policy management in OSNs, where data resources may belong to multiple users. [34] proposed a formal model to address multiparty access control in OSNs with a policy conflict resolution mechanism based on voting scheme to deal with collaborative policies. In this approach, the release of a resource depends on the sensitivity scores assigned by each controlling user and the chosen decision making strategy, such as setting a sensitivity threshold, owner-overrides and full-consensus-permit. Although policy conflict resolution is not the main focus of this dissertation, it is necessary to explicitly express an unambiguous strategy, whether a conjunction, a disjunction or a prioritized order of relationships between the policy specifiers and the user or resource the policies apply to.

Chapter 7: CONCLUSION

The following sections summarize contributions of this dissertation and discuss some future research directions that can be further studied.

7.1 Summary

In this work, we proposed a U2U relationship-based access control (UURAC) model for OSNs and a regular expression based policy specification language, which gives greater generality and flexibility in policy specification than prior models did. We also provided two path-checking algorithms based on DFS and BFS traversals, and established the correctness and complexity. Due to the sparseness nature of social graph, given the constraints on relationship types and hopcount limit, the complexity of the algorithms can be dramatically reduced. We demonstrated the feasibility of our approach by discussing a proof-of-concept implementation of both algorithms, followed by the evaluation results.

We extended the UURAC model to incorporate 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.

We also further included U2R and R2R relationships in policies and developed URRAC model that provides finer-grained access control for users' usage and administrative access. Specifically, we introduced the skipping of some relationship path expression in policy specification in order to offer more expressive policies. The decision modules of the system determine authorizations by retrieving different policies from the access session, the target and the system, and then making a collective decision. Conflict resolution policies are applied to address policy conflicts.

7.2 Future Work

There are several opportunities for extending the work presented in this dissertation.

To improve the versatility of ReBAC, it is possible to capture some unconventional relationships found in OSN systems, including temporary relationships and one-to-many relationships. The attribute-aware ReBAC model also needs to be adjusted accordingly to express the attributes of such new relationships.

We scoped out the access control problem regarding third party applications in OSNs in this work, which could be an important topic to be addressed in the future. Social network platforms and third party applications have made the popularity of OSN systems soar, but also pose serious risk in that rare control has been provided over disclosure of user data to the applications and these applications often gain much more privileges than necessary. We have made some initial progress on this topic so far [18], and will continue to work on applying relationship-based access control to solve the problem.

In Chapter 6, we considered system-specified conflict resolution policy to resolve conflicts between authorization policies. Since the system is the only one responsible for making policy, such conflict resolution will be unambiguous and will not conflict with itself. A further potential area of research is to design user-specified conflict resolution policy. This would allow more flexible and finer-grained control, as the policy is specified by users and applies to a smaller context.

BIBLIOGRAPHY

[1] Evangelos Aktoudianakis, Jason Crampton, Steve Schneider, Helen Treharne, and Adrian Waller. Policy templates for relationship-based access control. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*, pages 221–228. IEEE, 2013.

[2] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Proceedings of the 3rd Annual ACM Web Science Conference*, pages 33–42. ACM, 2012.

[3] Salem Benferhat, Rania El Baida, and Frédéric Cuppens. A stratification-based approach for handling conflicts in access control. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 189–195. ACM, 2003.

[4] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):71–127, 2003.

[5] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 94–107. IEEE, 1996.

[6] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems (TOIS)*, 17(2):101–140, 1999.

[7] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. Authorizations in relational database management systems. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 130–139. ACM, 1993.

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

[9] Glenn Bruns, Philip WL Fong, Ida Siahaan, and Michael Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *Proceedings of the second CODASPY*, pages 117–124. ACM, 2012.

[10] B. Carminati, E. Ferrari, and J. Girardi. Performance analysis of relationship-based access control in OSNs. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pages 449–456, 2012.

[11] Barbara Carminati, Elena Ferrari, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. A semantic web based framework for social network access control. In *Proceedings of the 14th SACMAT*, pages 177–186. ACM, 2009.

[12] Barbara Carminati, Elena Ferrari, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. Semantic web-based social network access control. *Computers & security*, 30(2):108–115, 2011.

[13] Barbara Carminati, Elena Ferrari, and Andrea 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.

[14] Barbara Carminati, Elena Ferrari, and Andrea Perego. A decentralized security framework for web-based social networks. *International Journal of Information Security and Privacy*, 2(4):22–53, 2008.

[15] Barbara Carminati, Elena Ferrari, and Andrea Perego. Enforcing access control in web-based social networks. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):6, 2009.

[16] Yuan Cheng, Jaehong Park, and Ravi Sandhu. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *PASSAT 2012*, pages 646–655. IEEE, 2012.

[17] Yuan Cheng, Jaehong Park, and Ravi 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.

[18] Yuan Cheng, Jaehong Park, and Ravi Sandhu. Preserving user privacy from third-party applications in online social networks. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 723–728. International World Wide Web Conferences Steering Committee, 2013.

[19] Frédéric Cuppens, Nora Cuppens-Boulahia, and Meriam Ben Ghorbel. High level conflict management strategies in advanced access control models. *Electronic Notes in Theoretical Computer Science*, 186:3–26, 2007.

[20] Dorothy E Denning, Teresa F Lunt, Roger R Schell, William R Shockley, and Mark Heckman. The SeaView security model. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 218–233. IEEE, 1988.

[21] Maeve Duggan and Aaron Smith. Social media update 2013. *Pew Internet Report*, 2014.

[22] Adrienne Felt and David Evans. Privacy protection for social networking apis. *2008 Web 2.0 Security and Privacy (W2SP)*, 2008.

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

[24] Philip WL Fong, Mohd Anwar, and Zhen Zhao. A privacy preservation model for facebook-style social network systems. In *Computer Security–ESORICS 2009*, pages 303–320. Springer, 2009.

[25] Philip WL Fong and Ida Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th SACMAT*, pages 51–60. ACM, 2011.

[26] Hongyu Gao, Jun Hu, Tuo Huang, Jingnan Wang, and Yan Chen. Security issues in online social networks. *Internet Computing, IEEE*, 15(4):56–63, 2011.

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

[28] Jennifer Golbeck and James Hendler. Inferring binary trust relationships in web-based social networks. *ACM Transactions on Internet Technology (TOIT)*, 6(4):497–529, 2006.

[29] Jennifer Ann Golbeck. *Computing and Applying Trust in Web-based Social Networks*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2005.

[30] Ralph Gross and Alessandro Acquisti. Information revelation and privacy in online social networks. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pages 71–80. ACM, 2005.

[31] Saikat Guha, Kevin Tang, and Paul Francis. NOYB: Privacy in online social networks. In *Proceedings of the first workshop on Online social networks*, pages 49–54. ACM, 2008.

[32] Keith N. Hampton, Lauren Sessions Goulet, Harrison Rainie, and Kristen Purcell. Social networking sites and our lives : how people's trust, personal relationships, and civic and political involvement are connected to their use of social networking sites and other technologies. *Pew Internet Report*, 2011.

[33] Michael Hart, Rob Johnson, and Amanda Stent. More content-less control: Access control in the Web 2.0. *IEEE Web 2.0*, 2007.

[34] Hongxin Hu and Gail-Joon Ahn. Multiparty authorization framework for data sharing in online social networks. In *Data and Applications Security and Privacy XXV*, pages 29–43. Springer, 2011.

[35] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and VS Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.

[36] Sushil Jajodia, Pierangela Samarati, and VS Subrahmanian. A logical language for expressing authorizations. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 31–42. IEEE, 1997.

[37] Sushil Jajodia, Pierangela Samarati, VS Subrahmanian, and Eliza Bertino. A unified framework for enforcing multiple access control policies. *ACM Sigmod Record*, 26(2):474–485, 1997.

[38] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI*, pages 41–55. Springer, 2012.

[39] Sebastian Ryszard Kruk, Sławomir Grzonkowski, Adam Gzella, Tomasz Woroniecki, and Hee-Chul Choi. D-FOAF: Distributed identity management with access rights delegation. In *The Semantic Web–ASWC 2006*, pages 140–154. Springer, 2006.

[40] Matthew M Lucas and Nikita Borisov. Flybynight: mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, pages 1–8. ACM, 2008.

[41] Wanying Luo, Qi Xie, and Urs Hengartner. Facecloak: An architecture for user privacy on social networking sites. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 3, pages 26–33. IEEE, 2009.

[42] Mary Madden. Privacy management on social media sites. *Pew Internet Report*, 2012.

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

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

[45] Jaehong Park, Ravi Sandhu, and Yuan 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.

[46] Jaehong Park, Ravi Sandhu, and Yuan Cheng. A user-activity-centric framework for access control in online social networks. *Internet Computing, IEEE*, 15(5):62–65, 2011.

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

[48] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems (TODS)*, 16(1):88–131, 1991.

[49] Haibo Shen and Fan Hong. An attribute-based access control model for web services. In *PDCAT 2006*, pages 74–79. IEEE, 2006.

[50] HongHai Shen and Prasun Dewan. Access control for collaborative environments. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 51–58. ACM, 1992.

[51] Anna Cinzia Squicciarini, Mohamed Shehab, and Federica Paci. Collective privacy management in social networks. In *Proceedings of the 18th international conference on World wide web*, pages 521–530. ACM, 2009.

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

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

[54] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[55] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 561–569. IEEE Computer Society, 2005.

[56] Chi Zhang, Jinyuan Sun, Xiaoyan Zhu, and Yuguang Fang. Privacy and security for online social networks: challenges and opportunities. *Network, IEEE*, 24(4):13–18, 2010.

VITA

Yuan Cheng was born in May, 1986 in Jinhua, Zhejiang, China and grew up in Ningbo, Zhejiang, China. Following graduation from Ningbo Foreign Language School and Ningbo Xiaoshi High School, Yuan received his Bachelor of Engineering degree with a major in Information Security from Huazhong University of Science and Technology, Wuhan, Hubei, China in 2008. He subsequently entered the doctoral program in the Department of Computer Science at the University of Texas at San Antonio in Fall 2008. He joined the Institute for Cyber Security at UTSA and started working with Dr. Ravi Sandhu since 2009. His research interests include security and privacy in cyber space. In particular, his focus is on developing access control solution for online social network systems.