ABSTRACT

PRIVATE KEY ALLOCATION BASED ACCESS CONTROL SCHEME FOR SOCIAL NETWORKS

by Preethi Srinivas

The problem of sharing resources in a secure manner in a social network has been addressed by various publications. Some of these schemes depend on a centralized server as a trusted third party, all of them require both the users (the user providing access and the user accessing the resource) to be online in order for the resource to be shared. This requirement is a drawback in real world as it is not feasible for all the users to come online each time their resource is accessed. We have designed and implemented a social network access control mechanism where the different users can have asynchronous access to their friends’ information (based on access rule satisfaction) and hide information from the server. The access control model in our system allows users to specify access policies based on distance in the social network. This access control model has been enforced via a key management system.
PRIVATE KEY ALLOCATION BASED ACCESS CONTROL SCHEME FOR SOCIAL NETWORKS

A Thesis

Submitted to the Faculty of Miami University

in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science and Software Engineering

by

Preethi Srinivas

Miami University

Oxford, Ohio

2010

Advisor______________________________

Dr. Keith B. Frikken

Reader______________________________

Dr. William John Brinkman II

Reader______________________________

Dr. Lukasz Opyrchal
# TABLE OF CONTENTS

1. Introduction .......................................................................................................................... 1

2. Background and Related Work ............................................................................................ 2
   2.1 Social Network Access Control ..................................................................................... 2
   2.2 Key Management for Access Control ......................................................................... 10
   2.3 Other Topics in Social Network Privacy ........................................................................ 12

3. General Overview ................................................................................................................ 14
   3.1 Problem Definition ........................................................................................................ 14
   3.2 High Level Description ................................................................................................. 15
      3.2.1 Converting Social Network Graph to Access Graph ........................................... 17
      3.2.2 Decentralizing the Scheme ................................................................................ 21
      3.2.3 Support for Removal of Relationship ................................................................. 25
      3.2.4 Other Extensions ................................................................................................ 29

4. Design and Implementation .................................................................................................. 30
   4.1 Overall System Architecture ........................................................................................ 30
   4.2 Database Design ........................................................................................................... 31
   4.3 Preliminary System Design ........................................................................................... 33
      4.3.1 User Creation ....................................................................................................... 33
      4.3.2 Login Process ...................................................................................................... 35
      4.3.3 Searching for users in the Social Network ......................................................... 36
      4.3.4 Offering Relationship ......................................................................................... 36
      4.3.5 Accepting Relationship Offer ............................................................................ 37
      4.3.6 Removing Relationship ....................................................................................... 39
      4.3.7 Posting Content .................................................................................................. 39
      4.3.8 Accessing Content .............................................................................................. 40
   4.4 Logic Flow ..................................................................................................................... 40
      4.4.1 Demo .................................................................................................................... 40
      4.4.2 Database Access ................................................................................................ 46
      4.4.3 Cryptography Used ............................................................................................ 46
### 4.4.4 secureServer Methods

#### 5. Experiments and Test Results.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Evaluation</td>
<td>48</td>
</tr>
<tr>
<td>5.2 Test Results</td>
<td>49</td>
</tr>
<tr>
<td>5.2.1 Offering a relationship</td>
<td>49</td>
</tr>
<tr>
<td>5.2.2 Accepting a relationship</td>
<td>50</td>
</tr>
<tr>
<td>5.2.3 Posting Content</td>
<td>50</td>
</tr>
<tr>
<td>5.2.4 Accessing Content</td>
<td>52</td>
</tr>
<tr>
<td>5.2.5 Removing a relationship</td>
<td>54</td>
</tr>
</tbody>
</table>

#### 6. Conclusion and Future work

#### 7. References

#### 8. Appendix

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Technology Used for Development</td>
<td>60</td>
</tr>
<tr>
<td>8.1.1 Client Front End: User Software</td>
<td>60</td>
</tr>
<tr>
<td>8.1.2 .NET Web Services: ServerSoftware</td>
<td>61</td>
</tr>
<tr>
<td>8.1.3 Backend Database</td>
<td>61</td>
</tr>
</tbody>
</table>

#### 9. Appendix B

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1.1 SHA256 Managed Class and Methods Details</td>
<td>62</td>
</tr>
<tr>
<td>9.1.2 RSACryptoServiceProvider Class and Methods Details</td>
<td>62</td>
</tr>
<tr>
<td>9.1.3 RijndaelManaged Class and Methods Details</td>
<td>63</td>
</tr>
</tbody>
</table>

#### 10. Appendix C

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 User Interface Screenshots</td>
<td>64</td>
</tr>
</tbody>
</table>

#### 11. Appendix D

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 smartKey Class and Methods Details</td>
<td>68</td>
</tr>
<tr>
<td>11.1.1 Graph Class</td>
<td>68</td>
</tr>
<tr>
<td>11.1.2 Node Class</td>
<td>68</td>
</tr>
<tr>
<td>11.1.3 NodeList Class</td>
<td>69</td>
</tr>
<tr>
<td>11.1.4 Edge Class</td>
<td>69</td>
</tr>
<tr>
<td>11.1.5 smyKey Class</td>
<td>69</td>
</tr>
<tr>
<td>11.1.6 rsaCrypt Class</td>
<td>70</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Access Control Mechanism (cited from [1]) 3
Figure 2.2 Portion of Social Network 4
Figure 2.3 Example Network of Users 5
Figure 2.4 After user B becomes friends with user A 6
Figure 2.5 Example RC Distribution 6
Figure 2.6 Example illustrating Onion Signature (cited from [3]) 8
Figure 2.7 Social Networking describing how intermediate user can affect resource access 9
Figure 2.8 Social Networking describing how intermediate users can affect resource access with privacy homomorphism 10
Figure 2.9 Resource Access with Privacy Homomorphism 10
Figure 3.1 Key Derivation 16
Figure 3.2 Example Social Network Graph 19
Figure 3.3 Example Access Graph 19
Figure 3.4 User subgraph for user A 21
Figure 3.5 Access Graph between users A and B 22
Figure 3.6 Access Graph between users A, B and C 26
Figure 3.7 Subgraph for user A after adding level of indirection 27
Figure 3.8 Access Graph between users A and B after adding level of indirection 28
Figure 3.9 Resource Access with Privacy Homomorphism 10
Figure 4.1 Overall System Architecture 30
Figure 4.2 Database Schema 31
Figure 4.3 User Account Creation 34
Figure 4.4 Subgraph for user Alice assuming relationship offer is sent to user Bob 37
Figure 4.5 Access Graph representing relationship between users Alice and Bob 38
Figure 4.6 Login Interface 41
Figure 4.7 User Interface for Signing up a new user 42
Figure 4.8 User Profile Page 42
Figure 4.9 Offering Relationship 43
Figure 4.10 User Update 43
Figure 4.11 Accepting Relationship 44
Figure 4.12 Accessing Content 45
Figure 4.13 Posting Content 45
Figure 4.14 Removing Relationship 46
Figure 5.1 Execution Times while Offering a Relationship ...................................... 50
Figure 5.2 Execution Times while Accepting a Relationship ..................................... 50
Figure 5.3 Execution Times while Posting Content at varying depths and varying network sizes 51
Figure 5.4 Execution Times across different network and content sizes .......................... 52
Figure 5.5 Execution Times while Accessing Content at varying network sizes and depths ...... 53
Figure 5.6 Total Execution Times while Accessing Content in a HighLevel Network ........... 54
Figure 5.7 Execution Times while Removing a Relationship ....................................... 55
Figure 8.1 Compiling C# code to Native code .......................................................... 60
Figure 10.1 Sample User Profile Page ........................................................................ 64
Figure 10.2 Sample Friends List .................................................................................. 64
Figure 10.3 Sample screen for Posting content ......................................................... 65
Figure 10.4 Sample screen for Offering Relationship ................................................ 65
Figure 10.5 Sample user updates ................................................................................ 66
Figure 10.6 Sample screen for Accepting relationship ................................................. 66
Figure 10.7 Sample screen for Accessing content ..................................................... 67
Figure 10.8 Sample screen for Removing Relationship ............................................... 67
Figure 11.1 Relationship between Graph, Node, NodeList and Edge Classes .................. 68
Figure 11.2 symKey Class ......................................................................................... 70
Figure 11.3 rsaCrypt Class ........................................................................................ 71
Figure 11.4 Window1 Class ....................................................................................... 71
Figure 11.5 SignUp Class ............................................................................................ 72
Figure 11.6 ChangePwd Class ................................................................................... 73
Figure 11.7 User Interface for Changing Password ................................................. 73
Figure 11.8 Relationship between Profile and ProfilePage Classes .............................. 74
Figure 11.9 Relationship between ProfilePage class and other classes ........................ 75
Acknowledgement

I would like to thank Dr. Keith B. Frikken for his support and guidance. I also show gratitude to my committee members Dr. William John Brinkman II and Dr. Lukasz Opyrchal for their involvement. Lastly, I would like to thank my family and friends for their support and encouragement.
1. Introduction

Social Networking is very common in today’s world. A Web-based social network is an online community of users where different users can establish relationships between one another and allow resources to be shared. A number of social networking sites such as Facebook, Orkut, FriendFinder, Twitter etc. are in use today. This means that a lot of information is available online. The availability of this information raises many privacy concerns. Such privacy concerns can be mitigated by allowing users to control access to their resources. A number of currently available access control mechanisms provide limited facilities for a user to control access to his resources. While some depend on a centralized server as a trusted third party [1], almost all the others [2, 3, 4, 5] require the resource owner and requestor to come online simultaneously in order to share resources. We have designed and implemented a decentralized system that is independent of a server as a trusted third party. The access control model we have implemented is based on distance between a pair of users who are trying to share resources. This mechanism has been implemented using a key management system similar to the one proposed in [6]. Such a key management system assigns keys that can be used to encrypt/decrypt the resource being shared.

A number of experiments were conducted in order to evaluate the performance of our implementation. These experiments helped determine the scalability of our system. Section 2 presents background and related works. Section 3 outlines the problem and our proposed solution, which includes a base scheme and an extended scheme. Section 4 describes the detailed design process for our implementation smartKey and secureSoftware. Section 5 describes the various experiments that were conducted and portrays the results that were obtained. Section 6 concludes the thesis and discusses future work.
2. Background and Related Work

This section discusses prior work related to social network security. The first subsection discusses prior work in social network access control; the next section describes prior work in enforcing access control via key management. The last subsection discusses other social network privacy research.

2.1 Social Network Access Control

A social network can be modeled as a graph with the nodes of the graph representing the different social actors, and the edges representing the different social relationships.

An important field of work with regard to privacy of data in social networks is the area of access control for social networks. An access control model that allows users to specify access rules for their content is given in [1]. According to this access control mechanism, each user has three parameters: i) relationship type- each user can be related to other users in a variety of ways (friendOf, colleagueOf, friendOffriend etc.), ii) depth- length of shortest path between a pair of users in the social network and iii) trust level- degree of trustworthiness between a pair of users. In summary, the access policy used in [1] is denoted by (rt,d,tl).

According to the access policy (rt,d,tl), the requestor (user requesting the resource) has to have a relationship type rt with a depth \( \leq d \) and a trust level \( \geq tl \) in order to access the resource r of the resource owner. In order to enforce this access control mechanism, [1] considers three different types of users, which are: the resource owner, the requestor, and a trusted third party (the central node, CN). Whenever access to a resource is requested, the resource owner gives the access policy (i.e., a set of access rules) to the requestor. In order to access a resource, the requestor then needs to provide the resource owner with the proof that the requestor satisfies the access policy. The resource owner must ensure that this proof provided by the requestor is correct and trustworthy. To facilitate such proofs, [1] uses certificates. Certificates are created and signed at the time of relationship creation and state that there exists a relationship between the users. For instance, when a user, Alice, establishes a relationship with another user Bob, they both create and sign a certificate stating that there exists a relationship between them with a certain trust level. In general, the correctness of a relationship between users Alice and Bob can be verified by retrieving all certificate chains between the two users. In the access control
mechanism defined in [1], certificates are stored in a certificate repository managed by the CN. Putting all these pieces together, the steps for the access control mechanism in [1] are as follows:

1. Requestor sends ID of resource to be accessed to the resource owner
2. Resource owner sends access policy for this resource to the requestor
3. Requestor sends request to CN to obtain all access policies
4. The CN uses its certificate repository to obtain all certificate chains between resource owner and requestor. If the given relationship type, trust level and depth are obtainable, it signs a certificate stating the trust level and relationship types and sends it to the requestor. If access is not granted, no certificate is sent
5. If access policy is satisfied, the requestor sends the certificate from CN to the resource owner

The access control model specified in [1] also allows combining relationships based on transitivity. This proves useful when pair of users is related through intermediate user(s) in the social network. Trust levels are calculated by taking the average of trust levels associated with all the edges in the paths from the requestor to the resource owner.
As an example, consider the above social network. Suppose user E is the owner of an object and she wishes to make it available to users who are of relationship type friendOf or colleagueOf with a trust level > 0.5. If user B can prove that he satisfies the requirements expressed by at least one of the access rules specified by E (applying to the requested object), he can access the requested object. From the above figure, it is seen that user B has a trust of 0.6 and has a relationship of type colleagueOf with user E. Hence, user B can access the resource held by user E. On the contrary, user D cannot access the resource held by user E. (This is because D has a trust level of 0.4, which is less than 0.5). Alternatively, consider the relationship between users B, C, D and E. Let user E have an access policy (friendOf, 2, 0.5). The trust level of user B is calculated as 0.25*(0.8+0.4+1+0.6) = 0.7 (paths considered: B->D->E and B->C->E). User B can still access user E’s resource since it has a relationship of type friendOf, has a depth of 2 and has a trust level of 0.7.

While [1] is a first step towards a private social network, it has a strong dependence on a trusted third party (TTP). This is undesirable for two reasons: i) compromising the TTP will compromise the entire social network, and ii) finding a TTP that the entire social network fully trusts are not always possible.

To mitigate this problem [2] extended the protocol in [1]. According to the protocol proposed in [2], the relationship certificates (that are used by the requestor to prove the existence of relationship with the resource owner) are prevented from being accessed by the central node. This removes the dependence on the central node as a trusted third party. [2] proposes a
mechanism, distribution rules, in order to prevent unauthorized access to relationship certificates. Distribution rules consist of a set of distribution conditions. These distribution conditions specify the conditions under which access to a relationship certificate should be granted. Distribution conditions are created whenever pair of users tries to establish relationship with one another. Both the users trying to establish a relationship negotiate the characteristics of the node that are authorized to read the relationship certificate they have created. This is followed by generation of a certificate key that is used to encrypt the relationship certificate. Hence, a distribution condition and its corresponding certificate key form a distribution rule. The encrypted relationship certificate is stored in a public certificate directory that is managed by the central node. Hence, the central node is not given access to the relationship certificate and merely acts as storage unit. All users in the social network are equipped with a certificate key directory in order to store the set of distribution rules concerning each certificate key. As an example, consider that users A, D are friends and users B, C are friends as shown below.

Let the user B want to be friends with user A

1. A and B generate the distribution condition DC = (B, friendOf, 2), i.e. disclose the relationship certificate to all of B’s friends and friend of friends.
2. A and B negotiate and generate the relationship certificate RC = (B, A, friendOf, 0.5) which indicates that A, B are friends and B can access A’s resource provided trust level >= 0.5.
3. The certificate key, CK = (k_{RC}, id_{RC}) corresponding to RC is also generated.
4. User B now sends CK and modified versions of DC to users A and C, while user A sends CK and modified version of DC to user D as shown below:

An access request from the requestor usually results in generation of access policy which reveals information regarding relationship between the requestor and the resource owner. The access control scheme in [2] uses a cryptographic-based approach, where every relationship type is assigned a symmetric key that is used to hide the relationship information. Whenever a pair of users establishes relationship between one another, a commonly agreed symmetric key and a key identifier are also determined to be associated with the relationship type. Every user in the social network related to another user with a particular relationship type will hold the same symmetric key associated with that relationship type. Hence, only those users who are related based on that relationship type can detect the relationship type.
The access control model in [2] enforces a different method of transitivity while calculating trust levels. Trust levels are calculated by taking the sum of product of trust levels of all the paths and dividing the result by the total number of paths from the requestor to the resource owner. Considering the same social network in Figure 2.2, the trust level of user B with respect to user E is calculated as $0.5\times(0.8\times0.4 + 1\times0.6) = 0.46$ (paths considered are B->D->E and B->C->E). Hence, if user E has an access policy (rid, {B,friendOf,2,0.5}), user B cannot access user E’s resource.

The scheme in [2] reveals relationship type to all those users who are intermediate to the requestor and the resource owner and are related to the requestor and resource owner with same relationship type. For instance, if users A and C are related (as friendOf, say) through user B and A is trying to access the resource held by C. B would know that A and C are related as friends when the request is passed through user B since A and B are friends. A, B and C will hold the symmetric key associated with relationship type ‘friendOf’ and hence can read the same.

In order to overcome the drawback of revealing relationship types to intermediate users, the scheme proposed in [3] uses the concept of collaboration. The goal of the collaboration is to identify a path in the social network satisfying the requirements stated by the access policy. The collaboration process is initiated by the resource owner when a request for resource is received. A collaboration request is sent to all the neighbors of resource owner related through the same relationship type as required by the access rule for the resource. All the users participating in collaboration satisfy the distribution rules associated with all the relationships in the path from the resource owner to the requestor. For this purpose, a data structure named Onion Signature is used in [3]. This is an ordered list containing the distribution rule and signature. This process helps prevent all those users not satisfying all the distribution rules from taking part in the collaboration process. For instance, refer to the social network in Figure 2.6. Consider the path (friendOf,<A,M,T>). This path indicates that there is an indirect friendOf relationship between users A and T. The corresponding Onion Signature is <((A, friendOf, 3), Sign_T(Sign_M(Sign_A(A, friendOf, 3)))), ((M, friendOf, 2), Sign_T(Sign_M(M, friendOf, 2)))>.
Let us assume that the user R is requesting access to resource rid held by user A.

1. A retrieves from its Policy Base all the access rules associated with the resource, rid. The access rule is $AR_1 = (\text{rid}, \{A, \text{friendOf}, 3, 0.8\})$.
2. A determines that the only neighbors of user A with relationship type friendOf are users C and M.
3. A collaboration message is initially sent to user C. If user C satisfies all the distribution rules, it adds itself to the path (partial path) and updates the Onion Signature sent to the user R. Since the user R is the requestor node, user C can verify whether user A satisfies all the distribution rules in the updated message. If the check succeeds, A is sent a message by user C.
4. User A, upon receiving the message can compute the trust level of user R based on the trust levels contained in the received message. Trust level is computed as $(1*0.8) = 0.8$. Since total path length is 2 and computed trust level is 0.8, user R is allowed to access the resource rid.
5. On the contrary, if user A had initially sent a collaboration message to user M, the path would have been $A \rightarrow M \rightarrow T \rightarrow R$ or $A \rightarrow M \rightarrow T \rightarrow C \rightarrow R$. Steps 3 and 4 are followed and if user A satisfies all the distribution rules in final updated message, then trust level is computed as product of trust levels of all the edges in the path.

All of the above schemes used private key cryptography in order to enforce security in resource sharing. One of the main drawbacks of using private key cryptography is the generation
and storage of a large number of keys. Since the symmetric key has to be held by all the users, it requires a complex key management and distribution system. Public key cryptography on the other hand, doesn’t require all the users to store a symmetric key with all other users. The access control scheme proposed in [4] reduces the overhead caused by private key relationships. It is similar to the one proposed in [2] with the exception that it uses public-key cryptography. According to the access policy specified in [4], access is enforced based on relationship path between the requestor and the resource owner that yields maximum trust.

The access enforcement protocol in [4] involves the requestor sending the relationship certificate to the resource owner in order to prove relationship type, depth and trust level once the resource owner gives a set of access rules. Access is provided based on depth of relationship. For instance, consider the network in Figure 2.7. Let user A is the requestor, user C is the intermediate user and user B is the resource owner.

1. User A will ask all the users (in this case user C) with whom it is related whether they have direct relationship (of type specified by the access policy) with user B.
2. Since user C is related to B (same relationship type, friendOf), it will send user B a signed, encrypted certificate of his relationship and informs A that a certificate has been sent to user B.
3. User B can now evaluate the relationship type, trust level in order to grant access to A.

![Figure 2.7 Social Networking describing how intermediate user can affect resource access](image)

A possible drawback with this approach is the fact that intermediate users have to disclose to the resource owner information about their trust level for their neighbors. Such a disclosure doesn’t bring any direct benefit to these users other than maintaining good terms with the upstream neighbor. This drawback is overcome by the access control scheme proposed in [5]. The access control scheme proposed in [5] makes use of privacy homomorphism in order to encrypt trust values contributed by the nodes in the social network. Privacy Homomorphisms (PHs) are encryption transformations mapping a set of operations on cipher text. PHs used in [5] are multiplicative, probabilistic and public-key. Since multiplicative homomorphisms are only
available for integers, [5] proposes to encode rational trust values as integer fractions. Since knowledge of depth can be used by resource owner to infer trust value of relationships between users who collaborate in resource access, the access policy doesn’t include maximum depth. Hence, the access policy in [5] includes only relationship type and trust level. The existence of a trusted third party is assumed in [5]. Such a TTP acts in case of conflict between the users of the social network and is not used during normal execution. Consider the following social network where user A is trying to access the resource held by user B:

1. User A contacts user C to determine if user C is directly related to user B.
2. User C contacts its direct friend user D to determine if it is related to user B.
3. User D has direct relationship to users B and C. Hence it multiplies his own trust related to C and the current trust value that comes from user C. User D sends a message containing \((PK_B(\text{rt}), PK_B(t_{CA} \cdot t_{DC}))\) to user B.
4. User B evaluates whether a relationship of type \(rt\) and trust level \(t_{CA} \cdot t_{DC} \cdot t_{BD}\) is enough to grant access.

Refer to [5] for more details regarding access control.

2.2 Key Management for Access Control

Our work enforces access control through key management. According to our approach a user, Alice, will have another user’s, Bob’s, key if and only if Alice is trusted by Bob in the social network. The problem of key management for access hierarchies has been well studied. An instance of access hierarchy is role based access control model (RBAC). Such a model is used in many organizations. According to this model, roles are defined for different users based
on which they can access resources. This user population is modeled as a set of classes that are partially ordered. A user with access to a certain class also has access to all of its descendant classes. Such a model is used in [6] and is depicted with the help of a directed graph $G$, where vertices denote classes and edges indicate their ordering (this represents the hierarchy). Each class in the hierarchy is associated with a set of objects. The key management scheme defined in [6] is used to assign keys to access classes and distribute a subset of keys to a user, which in turn permits access to all descendant classes. According to this scheme, if there is a path from vertex $A$ to vertex $B$ in the access graph, then $A$ can derive $B$’s key. However, absence of a path would mean $B$’s key should be indistinguishable from a randomly generated value.

The key management scheme described in [6] includes a key allocation scheme and a key derivation scheme. Every vertex in the access hierarchy is assigned a random private key and has a unique public label associated with it. The key derivation scheme defined in [6] is used to describe how to generate a vertex’s key from another vertex’s private information and the public information. The paper also extends the base scheme to support dynamic changes to the access graph without having to rekey the individuals. The dynamic changes supported are changing key of a node, adding edges, removing edges and adding nodes.

The key management scheme proposed in [6] however depended on a server as a key authority for managing the keys. The key management scheme proposed in [7] extends [6] in order to avoid this dependence on a server as a key authority. It applies its key management scheme to an access control model for protecting the access policies through encryption. It makes use of two layers of encryption, the inner layer being imposed by the owner (for providing initial protection) and the outer layer being imposed by the server (to reflect policy modifications). This avoids re-encrypting the resources every time the authorization policy is changed. It assumes the existence of set of symmetric keys and public tokens in the system. Each user is associated with a single key, which is sent to her by the resource owner when the user joins the system. Key derivation is done using the public tokens (key derivation scheme defined in [6] is followed here). A major advantage of using tokens is that they are public and they allow each user to derive multiple encryption keys, while having to securely manage a single one.

While the scheme proposed in [7] takes advantage of the public tokens in order to derive keys, it exposes the access graph. The scheme proposed in [7] cannot be applied to a social
network graph as it would reveal the entire social network. The scheme proposed in [8] mitigates this problem by describing an approach to protect the privacy of the tokens published in the public catalog. This solution adds an encryption layer to the catalog. The key derivation strategy followed in [8] is the similar to that described in [6].

2.3 Other Topics in Social Network Privacy

Social Networking is ubiquitous and involves a huge amount of resources being shared between the different users. A substantial amount of work has been attempted to address the problem of sharing resources in a secure manner. In this section we will discuss prior work related to privacy in social networks.

With social networks becoming ubiquitous, it is important in order to perform social network analysis. Social network data has to be published in order to perform analysis on it. Such an analysis might involve determining relationships between individuals based on various contexts (for instance to identifying common interests of a group of individuals or identifying individuals who belong to a particular group and so on). Privacy is an important concern when social network data is published. It is easy for an adversary to get to know the private information of a social network user based on published data and some background knowledge. One approach used to avoid such a privacy breach is anonymizing the social network data that is published. This involves replacing identifying attributes (such as name, SSN etc.) with meaningless identifiers. This is called naïve anonymization. This is however not enough according to [9] since it is possible for the adversary to learn whether edges exist or not between some specific pairs of vertices by embedding a random sub graph in the network. Once the social network data is published, if the adversary can locate the inserted sub-graph, then it can follow edges from this sub-graph to locate the target vertices and their location in the main graph. Two methods for anonymizing social network data, namely clustering-based method and graph modification method are surveyed in this paper.

1. The clustering–based method essentially involves grouping all users in the social network based on a characteristic. The various sub-categories in this approach are i) vertex clustering, ii) edge clustering, iii) vertex and edge clustering, iv) vertex attribute clustering method. All these approaches involve reduction of a cluster of vertices and edges into a super-vertex.
2. In graph modification approach, the scale and structure of the social network graph is preserved.

A protocol is defined in [10] for privately assembling a graph that is published among a large number of people. Such a protocol generates a naively-anonymized graph. A naively-anonymized graph consists of ‘synthetic’ identifiers in place of actual identifying attributes (This is done by encrypting the actual identifiers). The main drawback with the approaches in [9, 10] is that a user’s connection can be distinguished and hence, can be used to re-identify an otherwise anonymous individual.

These re-identification attacks are overcome in [11] by partitioning the nodes and then describing the graph at the level of partitions. Such an approach generates a ‘generalized’ graph with ‘supernodes’ (one for each partition) and a set of ‘superedges’ (these report density of edges between the partitions they connect).

In all the approaches mentioned above, a central authority is assumed, which can publish social network data for analysis. The goal of these approaches is very different from the goal of our work. Our model proposes to avoid the use of a central authority that has the knowledge of the entire social network. It provides the ability to each user to share resources with one another without depending on a trusted third party.
3. General Overview

3.1 Problem Definition

From literature review in Section 2, the following issues or shortcomings were noticed with respect to the current work for access control in social networks:

- Most of the solutions involved a centralized server for enforcing access control, hence exposing user information and user content to the server.
- Solutions that avoided the use of a centralized server enabled the user to manage their own access control. However, these solutions required multiple users to come online simultaneously in order to facilitate resource sharing (i.e., they are synchronous).

The goal of this thesis is to develop an access control enforcement mechanism that does not have the above-mentioned drawbacks. We begin by describing a base scheme that achieves the following:

- User content is protected from the server. The social network is however not hidden from the server.
- Users can be added to the system. Relationship removal is not supported.
- Content can be distributed based on distance between the users in the social network.

We now proceed to explain the notation used in the rest of this thesis for our access control model. The social network is represented as a directed graph $G = (V, E)$, where $V$ is the set of vertices representing each user and $E$ is the set of edges representing the edges between the users. Each user in the social network corresponds to a node in the directed graph. An edge $(u,v)$ in the graph $G$ between the users $u$ and $v$ indicates that the user $u$ trusts the user $v$. The shortest distance between a pair of users $u$ and $v$ in the social network is defined as $\text{depth}_G(u,v)$. We define $F_{u,d}$ as all nodes in the set $V$ that are at depth $d$ from node $u$ in the graph $G$. We also define $F'_{u,d}$ as all nodes in the set $V$ that are at a depth $\leq d$ from node for user $u$ in the graph $G$. Hence, the set $V-F'_{u,d}$ consists of all nodes in the graph $G$ that are at a depth $>d$. Let $L$ be the maximum allowable depth up to which a user can share information. A particular user from the social network holding content ‘rid’ will define his access policy as $(\text{rid},d)$, where $d$ represents the depth up to which the content rid can be accessed. Hence, all those users who’s nodes belong to
the set $F'_{u,d}$ can access the content rid. All the users who’s nodes belong to the set $V-F'_{u,d}$ cannot access the content rid.

### 3.2 High Level Description

We enforce the access control model outlined in the previous section via key management. This thesis enforces the above mentioned access control model by utilizing the key management scheme from [6]. In a key management solution, each user has a set of keys which is used to encrypt their content. The key management system ensures that the set of users with a key used to encrypt a specific piece of content are exactly the users that are allowed to access this content. Since the content is encrypted, users that do not satisfy the access policy for a particular content will not be able to access this content. To reduce the number of keys in the system, key management scheme often employs key derivation. More specifically, a collection of public information allows users with some keys to derive other keys.

The key management scheme in [6] was defined to enforce access control in a hierarchical access control model (such as RBAC). Such a hierarchy was represented as a directed graph (known as the access graph) where each node represents an access level. Every node in the graph is assigned a key. The key at a particular node can be used to encrypt the objects held at that node. The access model is the following: a node has access to all the objects held by its descendants. To support this model, the key derivation method of [6] allowed anyone with a specific node’s key to derive all keys of that node’s descendants. According to the key management scheme from [6], every vertex in the access hierarchy is assigned a random private key and a unique public label. The key derivation scheme defined in [6] is illustrated with an example below:
1. Let the vertices have a public label $l_i$ and a private key $k_i$.
2. Let the public edge labels between the vertices be as shown above.
3. Given the public vertex labels, edge labels and the private key $k_1$, the keys $k_2$ and $k_3$ are then derived as
   \[ k_2 = y_{1,2} + H(k_1, l_2) \]
   \[ k_3 = y_{1,3} + H(k_1, l_3) \]
4. It is not possible for the vertex with key $k_3$ to derive the key $k_2$ since it requires inversion of the cryptographic hash function $H()$.
5. The above process can be reversed to generate the public edge label between the vertices. For instance, $y_{1,2}$ can be generated as
   \[ y_{1,2} = k_2 - H(k_1, l_2) \]

The key management scheme proposed in [6] cannot be directly applied to the social network graph. This is because every user in the social network will have access to any key as long as there is a path between the users in the social network graph (i.e., there is no distinction based on depth), which is not the goal of our access control model. Hence, we convert the social network graph into an access graph and then apply the key management scheme defined in [6] on the resulting access graph. In this conversion, each node in the social network graph $G$ will correspond to $(L+2)$ vertices in the resulting access graph $G'$. Each of these vertices is assigned a
public node label and a secret key (secret information). One vertex is referred to as the master vertex and the remaining (L+1) vertices are referred to as the content vertices. A particular content vertex is denoted by $V^d_v$, where $v$ represents the user from $G$ and $d$ corresponds to the depth that is represented by the content vertex. The master vertex is denoted as $V^d_v$, where $v$ represents the user from $G$. Each of these content vertices represents a particular depth at which user can post encrypted content. For instance, a user A can post content at depth 2 by posting content using the key of content vertex $V^2_A$. We design the access graph such that there is a path from any of user v’s vertices to $V^d_u$ in $G'$ if and only if there is a path of length $\leq d$ between the users $u$ and $v$ in the social network graph.

By applying the key derivation mechanism from [6] to the access graph, it is possible for a user $v$ to derive the key associated with the content vertex $V^d_u$ as long as there is a path from master vertex $V_v$ to the content vertex $V^d_u$ in the access graph. By the property of the access graph, such a path exists if and only if there is a path from node of user $u$ to node of user $v$ of length $\leq d$ in the social network graph. Hence, by applying the key management scheme from [6] on our newly created access graph, it is possible to provide our above mentioned access control model.

Unfortunately, we cannot just apply the solution from [6] to the access graph. This is because the key management scheme proposed in [6] assumes the existence of a key authority for assigning keys. If we applied [6] directly to our solution, then all the keys are visible to the server, and thus exposes the user content to the server. Hence, we decentralize the scheme from [6] so that the keys are generated by the individual users instead of a centralized server.

The rest of the section is organized as follows: in section 3.2.1 we describe the conversion process and prove the key properties for this conversion. Then in section 3.2.2 we describe how to decentralize the scheme from [6]. Next, in section 3.2.3 we describe how to extend the base scheme to support removals. Finally, in section 3.2.4 we briefly summarize some other extensions that were not implemented as part of this thesis.

### 3.2.1 Converting Social Network Graph to Access Graph

As mentioned in high level description, the social network is represented as a directed graph $G = (V, E)$ before it is converted in to an access graph, $G'$. Every user in the social
network will have a node that is a part of the set V, and all edges between the users in the social network are a part of the set E. For every node in the set V, (L+2) vertices are created in the access graph G’. These vertices represent the master and content vertices as described in previous section. For every node in set V, edges are added between its master vertex and each of its content vertices. Edges are also added between the content vertices. For every edge (u, v) in the set E, an edge is added between the content vertices $V_v^i$ and $V_u^{i+1}$ where $i$ varies from 0 to (L-1). The process of converting a social network graph into an access graph is described in algorithm 1.

<table>
<thead>
<tr>
<th>Algorithm 1 CreateAccessGraph(G, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V' = {}$</td>
</tr>
<tr>
<td>$E' = {}$</td>
</tr>
<tr>
<td>for all $v$ in V do</td>
</tr>
<tr>
<td>$V' = V' \cup {V_v}$</td>
</tr>
<tr>
<td>for $i = 0$ to $L$ do</td>
</tr>
<tr>
<td>$V' = V' \cup {V_v^i}$</td>
</tr>
<tr>
<td>$E' = E' \cup {(V_v, V_v^i)}$</td>
</tr>
<tr>
<td>if $i \neq 0$ AND $i+1 \leq L$ then</td>
</tr>
<tr>
<td>$E' = E' \cup {(V_v^i, V_v^i)}$</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>for all $(u,v)$ in E do</td>
</tr>
<tr>
<td>for $i = 0$ to $(L-1)$ do</td>
</tr>
<tr>
<td>$E' = E' \cup {(V_v^i, V_u^{i+1})}$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>$G' &lt;-(V', E')$</td>
</tr>
<tr>
<td>return $G'$</td>
</tr>
</tbody>
</table>
As an example, consider the social network graph as shown in Figure 3.1. It is converted to an access graph as shown in Figure 3.2.

Figure 3.2 Example Social Network Graph

Notice that there is a change in direction of edges between the users in the social network graph and the access graph. An edge from node of user A to node of user B in social network graph is shown as an edge from content vertex $V_B^i$ to $V_A^{i+1}$. This implies that if user A trusts user B in social network graph, then user B should be able to derive user A’s key corresponding to its content vertices in the access graph. Also, notice that $i$ varies from 0 to $L$, where $L$ is the maximum allowable depth up to which a user wants to share content.

Figure 3.3 Example Access Graph

We now proceed to prove the key property of access graph, which is stated as Theorem 1.

**Theorem 1**: There is an edge from master vertex $V_B$ of user B to the content vertex $V_A^d$ of user A if and only if $\text{depth}_G(A, B) \leq d$. 
Proof:

Let $G'$ be the access graph generated from the social network graph $G$. There are two parts to completing this proof:

I. Assuming $\text{depth}_G(A,B) \leq d$, we must prove that a path exists from master vertex $V_B$ of user $B$ to the content vertex $V_A^d$ of user $A$.

II. Assuming a path exists from master vertex $V_B$ of user $B$ to the content vertex $V_A^d$ of user $A$, we must prove $\text{depth}_G(A,B) \leq d$.

Let us assume that the given a social network graph $G$ has users $A, A_1, A_2, \ldots, B$ such that the distance between the users $A$ and $B$, $\text{depth}_G(A,B) \leq d$.

We must prove that there exists a path in access graph $G'$ from master vertex $V_B$ of user $B$ to the content vertex $V_A^d$ of user $A$.

Since $\text{depth}_G(A,B) \leq d$, we have a path between users $A$ and $B$ of length $p \leq (d-1)$ i.e. there are intermediate users between users $A$ and $B$ such as $A, A_1, A_2, \ldots, A_p, B$. The edges ($V_{A_{i+1}} \rightarrow V_{A_i}$) are added based on algorithm 3 and the edges ($V_B \rightarrow V_B^0$) are added based on algorithm 1. The corresponding access graph $G'$ for this social network graph will then have a path as

$$V_B \rightarrow V_B^0 \rightarrow V_{A_1}^1 \rightarrow \ldots \rightarrow V_{A_p}^{p-1} \rightarrow V_{A_1}^p \rightarrow V_A^{d+1}$$

The total path length of this path is $(p+1) \leq d$. Hence, proved.

In order to prove statement II, let us assume that there exists a path in access graph $G'$ from master vertex $V_B$ of user $B$ to the content vertex $V_A^d$ of user $A$.

We must prove that the distance between the users $A$ and $B$, $\text{depth}_G(A,B) \leq d$ in the social network graph.

The assumed path in access graph $G'$ is

$$V_B \rightarrow V_B^0 \rightarrow V_{A_1}^0 \rightarrow \ldots \rightarrow V_{A_p}^p \rightarrow V_A^d$$
We know that there are no incoming edges for any master vertex. Hence, this path has only one such master vertex. Based on algorithm 3, notice that if there is an edge from $V_x^i \rightarrow V_y^j$, then $j=i+1$. Hence, we can re-write the above path as

$$V_B \rightarrow V_B^0 \rightarrow V_{Ap}^1 \rightarrow \ldots \rightarrow V_{Ap}^{d-1} \rightarrow V_{Ap}^d$$

If the intermediate users are not duplicate, i.e. $A_i \neq A_{i+1}$, then we know that there exists an edge between $A_i$ and $A_{i+1}$. Thus, we know that there exists a path $A, A_1, A_2, \ldots, A_n, B$ in the social network graph $G$. Also, $n < d$. Hence, there exists a path between users $A$ and $B$ with a length $\leq d$ i.e depth$_G(A,B) \leq d$. Hence, proved.

3.2.2 Decentralizing the Scheme

In this section we define a scheme that removes the dependence on a server as the key authority. According to this decentralized solution, each user will create its own access graph, known as the user subgraph. Such a subgraph consists of a master vertex and $(L+1)$ content vertices, where $L$ is the maximum allowable depth up to which the user can share content. The master vertex for a user $v$ is denoted by $V_v$ and the content vertices are denoted by $V_v^i$ where $i$ varies from 0 to $L$. Each of the vertices in the user subgraph is assigned a public node label and a secret key by the user, thus avoiding the need for a server as key authority. Since the keys are assigned by the user, they are known only to the user. For instance, user $A$ will create a user subgraph as shown in Figure 3.3. The user can derive edge labels for the user subgraph using techniques from [6].

![Figure 3.4 User subgraph for user A](image)

An edge from user $A$ to user $B$ in the social network graph $G$ is added in two steps:
- User A initially sends its subgraph information along with the keys associated with its content vertices to user B (as a relationship offer). We assume that the communication between user A and user B is hidden from the server.
- User B computes the edge label and adds edges from its content vertices to the content vertices of user A’s subgraph (response to relationship offer).

Notice that the resulting access graph is similar to the access graph described in previous section. The resulting access graph between the users A and B is as shown in Figure 3.4.

![Access Graph between users A and B](image)

**Figure 3.5 Access Graph between users A and B**

The user subgraph is created using algorithm 2. It is assumed that the keys assigned for all the vertices in the user subgraph are stored in a local storage by every user. A user can send a relationship offer (algorithm 3) using the SEND method. SEND(sender, receiver, msg) sends a message from sender to receiver through a private authenticated channel. The response to a relationship offer is made by using algorithm 4. An edge is added between a pair of vertices using the CreateEdge method. This method essentially reverses the DER algorithm from [6].
Algorithm 2 CreateSubGraph(A, L)

\[ V' = \{\} \]

\[ E' = \{\} \]

\[ V' = V' \cup \{V_A\} \]

for \( i = 0 \) to \( L \) do

\[ V' = V' \cup \{V_A^i\} \]

\[ E' = E' \cup \{\text{CreateEdge}(V_A, V_A^i)\} \]

if \( i \neq 0 \) AND \( i + 1 < L \) then

\[ E' = E' \cup \{\text{CreateEdge}(V_A^i, V_A^{i+1})\} \]

end if

end for

\[ G' \leftarrow (V', E') \]

return \( G' \)

Algorithm 3 OfferRel(A, B)

\[ G_A \leftarrow \text{Obtain user subgraph for user A} \]

SEND (A, B, \( G_A \))
Algorithm 4 AcceptRel(B, A)

G <- subgraph sent by user A

G’ <- Obtain subgraph of user B

for i = 0 to L do
    if i + 1 < L then
        \[ E_i = \text{CreateEdge}(V_B^i, V_A^{i+1}) \]
        \[ E' = E' U \{E_i\} \]
        \[ \text{msg} = \text{msg} \parallel E_i \]
    end if
end for

SEND(B, A, msg)

CreateEdge(L_A, k_A, L_B, k_B)

\[ y_{AB} <- k_A - H(k_B \parallel L_A) \]

return \[ y_{AB} \]
3.2.3 Support for Removal of Relationship

Making changes to the access graph by removing vertices or edges is not trivial because the system is decentralized. There are a number of challenges when these dynamic changes are added to the base scheme. These are:

1. All incoming edges should be updated when content vertex is re-keyed. For instance, consider the access graph between users A and B in Figure 3.3. If user A has to re-key the node $V_A^1$, it is not possible since there is no access to the edge between the nodes $V_A^1$ and $V_B^0$. Re-keying the node $V_A^1$ without updating this edge label will not allow user B to access content of user A at depth 1 in future.

2. When removing a vertex, all vertices reachable from it have to change keys. For instance, consider the access graph between the users A, B and C in Figure 3.5. Let us assume that user B is being removed. This would mean user C should no longer be able to derive the key of the content vertex $V_A^2$ and hence not gain access to A’s content at depth 2. However, if user C had stored a copy of this key earlier, he can still access A’s content at depth 2. Hence, user A will have to re-key the content vertices $V_A^1$ and $V_A^2$. This also includes decrypting content with original key and re-encrypting with the newly generated content keys.

3. Removing an edge requires re-keying the content vertices. For instance, consider the access graph between the nodes A, B and C in Figure 3.5. If user B wants to stop being friends with user C, the edges between their corresponding content nodes can be removed. In such a case, user B will have to inform user A to re-key its content vertices. This is done in order to avoid user C to use the stored content key of user A to access content when it is forbidden to.
To minimize these problems, we added a level of indirection to the access graph in order to make dynamic changes possible. Adding level of indirection involves creating additional vertices named *derivation vertices* in the user subgraph. Edges are added between the derivation vertices and the content vertices in the user subgraph and between the derivation vertices. Each of these derivation vertices are assigned *derivation keys* in a process similar to that of assigning keys to the other vertices in the user subgraph. Content held by a user is still encrypted using the keys corresponding to the content vertices. The derivation keys are used to ‘derive’ the keys associated with the content vertices in order to gain access to content. Hence, when user A is offering a relationship to user B, it sends a user subgraph similar to the one shown in Figure 3.6. On receiving the offer, user B computes and adds edges from its content vertices to the corresponding derivation vertices as shown in Figure 3.7.
Figure 3.7 Subgraph for user A after adding level of indirection
The key property of the access graph, which is, an edge from master vertex of user B’s subgraph to the content vertex of user A’s subgraph exists if and only if there exists a path of length \( \leq d \) between the users A and B in the social network graph is still true. In order to prove that the problem of re-keying (as mentioned above) is solved, let us assume the case where user A wants to re-key the content vertex \( V_A^1 \). This process will involve changing all the edges coming into the content vertex. Notice that this will also involve downloading all the encrypted content at depth 1 from the server and decrypting them with original key before being encrypted with the new key. This process of re-keying the content vertex will not affect user B’s access to user A’s content at depth 1 since the edge information of the paths \( V_B \rightarrow V_B^0 \rightarrow V_{AB}^0 \) is not changed. User B can still derive the new key at the content vertex \( V_A^1 \) as described above.

We now proceed to explain how removing a user from the system affects resource sharing. Consider the access graph in Figure 3.5. Let us assume that user B is removed from the social network. This would mean the user subgraph of user B should be removed from the access
graph in Figure 3.5. This would also mean that user C should no longer be able to access user A’s content at depth 2. If user C has cached a copy of user A’s key at content vertex $V_{A^2}$, and if user A does not re-key the corresponding content vertex, then user C will still be able to access content posted by user A at depth 2. Hence, when user B is removed from the social network, user A will have to re-key its content vertices (as mentioned above) before removing the edges between its derivation vertices and content vertices of user B.

Hence by adding a level of indirection and introducing the concept of deriving keys, we have achieved the following:

- A particular user can remove a relationship with another user. This affects the way friend of friends access content.
- A particular user can re-key his content vertices.
- A particular user can be removed from the social network.

### 3.2.4 Other Extensions

In this section, we provide a summary of other possible extensions that can be made on the base scheme. These extensions were not implemented due to lack of time.

- While protecting the resources, the base scheme fails to protect the entire social network since the public information of the social network graph can be used to re-create the entire graph. The scheme proposed in [10] can be used to hide the entire social network by encrypting the edge to the destination node and the destination node. This scheme however requires multiple decryptions in order to perform derivation and requires the entire social network to be downloaded each time resource is shared.
- Web-based social networks generally allow multiple types of relationships to exist between the different users. For instance, user A can be related to B as a friend and as a colleague. In such a case, there should be a policy that allows friends to access certain content and colleagues to access certain content. People with both the relationship types should be allowed to access both the contents.
- Strengths can also be associated with relationships. From Background/Related work section (section 2), it can be seen that the access control mechanisms defined relationship between a pair of users to be associated with trust levels [1,2,3,4,5]. These trust levels
were transferred in a multiplicative manner. Each trust level can thus be associated with a content vertex.

4. Design and Implementation

This section will describe the design details of smartKey and secureSoftware. First, the general overview of the entire system is discussed. This is followed by database design and preliminary system design. The section ends with a brief discussion of the classes and methods used by smartKey and secureSoftware.

4.1 Overall System Architecture

In this section we outline the overall architecture of the system that was described in previous section. This implementation is a client/server based solution which allows users to create an account, allows other users to be related to them and share resources based on the shortest distance between the users. The main goal of the implementation is to determine the efficiency of various tasks such as creating a user account, creating a relationship and sharing the resources.

The entire system consists of

- A centralized server (server software)
- Client-side front end (user software)
- A Database Management System (DBMS) for storing user information

![Overall System Architecture](image)

Figure 4.1 Overall System Architecture

Figure 4.1 above is an overview of how the smartKey and secureServer systems work. The user primarily has access to the user software (smartKey). The user can create an account using the user software which sends the user information to the server (secureServer) for storing
it in the database. The user information is sent over a secure channel by means of SSL. An SSL certificate will help enable a private and authenticated communication channel.

4.2 Database Design

The database schema is as follows:

![Database Schema Diagram](image)

**Figure 4.2 Database Schema**

Each of the tables in the database is used for a specific purpose. The table `userInfo` is used to store user information such as:

- **uid** – an automatically generated random, unique value for every user. Acts as the primary key for the table.
- **uname** – user name (chosen by the user).
- **fname** – first name of the user.
- **lname** – last name of the user.
- **email** – email address of user.
- **salt** – random value used to concatenate with user chosen password.
• pwd – user chosen password. This is concatenated with a random salt value to generate a new value that is hashed and stored as user password. This is done in order to avoid a malicious user from gaining access to user’s password.
• publicKey – public key generated for the user. This is used while exchanging messages between users. This is done in order to hide information that is exchanged from the server.

The table `nodeInfo` is used to store information related to a particular node in the subgraph:

• vid – unique public node label. Acts as the primary key for the table.
• uid – user ID of the user. Acts as foreign key to table `userInfo`.
• type – type of the node. There are 3 types of nodes: master, content, derived.
• depth – depth at which the node is related to its parent node.

The table `edgeInfo` is mainly used for storing edge information between a pair of nodes in the subgraph:

• edgeID – unique identifier. Acts as the primary key for the table.
• uid – user ID of the user. Acts as foreign key to table `userInfo`.
• vid – node label of target node. Acts as foreign key to table `nodeInfo`.
• srcid – node label of source node (from which edge is originating).
• desid – node label of destination node (into which edge is entering). Value is same as that of vid.
• info – edge label that was created while an edge was added between the source and destination nodes.

The table `contentInfo` is used to store content information of every user as:

• uid – user ID of user. Acts as foreign key for the table `userInfo`.
• vid – node label of the node with which the content is associated. Acts as foreign key for the table `nodeInfo`.
• depth – depth at which the content was posted (0,1,2,…).
• [content] – encrypted content.
• meta – meta information of the content (date of creation, date last modified, etc.)
• contentID – unique identifier. Acts as the primary key for the table.

The table relInfo is used to store information related to relationship between a pair of users:

• relID – unique identifier. Acts as the primary key of the table.
• uid – user ID of user. Acts as foreign key for the table userInfo.
• depth – distance between the users in the social network (1,2,…).
• relType – type of relationship (friend, colleague, etc.).
• relatedTo – username of user to whom this user is related.

The table updateInfo is used to store information related to updates or the messages that are passed between users in the social network as:

• updateID – unique identifier. Acts as the primary key of table.
• uid – user ID of user. Acts as foreign key for the table userInfo.
• updateFrom – username of user from whom a message (or update) was sent.
• updateTo – username of user to whom message was sent.
• updateType – type of message sent (request, response, post). Request would mean a relationship offer while response would mean an offer is either accepted or rejected.
• flag – Boolean value used to check if the update has already been seen.
• updateInfo – the actual update or the message. This is encrypted with the public key of the user to whom the update is being sent.

4.3 Preliminary System Design

In this section, we outline the preliminary system design. Each of the processes involved in the entire system are outlined as a summary of operations.

4.3.1 User Creation

The user has user software installed in his machine. This user software helps create the user subgraph, whose edges can be sent to the server. The keys for every vertex in the subgraph are also created and stored in a separate file by this user software. This separate file can be stored on an external USB drive by the user and plugged in every time he logs into his account. The keys are not shared with the server and are not stored in the database. This keeps the keys safe and known only to the user. While creating a new user, the user software requires the user
information such as first name, last name, email and so on. The user can choose a username and password. These are stored in the *userInfo* table. The username has to be unique. A unique user ID is generated every time a user account is created. This is the table’s primary key and is used to uniquely identify the user.

The user is presented with a user creation page like the one shown below:

![User Account Creation Form](image_url)

The user software creates the user subgraph with the master and content vertices. It assigns random, unique symmetric keys to all the created vertices. A subgraph (mentioned in Section 3.2.2) consisting of a master vertex and \((L+1)\) content vertices are created (from Section 3.2.1, \(L\) refers to the maximum allowable depth up to which the user wants to share resource). The keys (master and content keys) are stored in a separate file that can be stored by the user in an external thumb drive and used the next time he tries to login. While the master key is used to encrypt/decrypt the user profile picture, the content keys are used to encrypt/decrypt the content stored at each content vertex. The user information along with the subgraph vertices, edges are sent over a secure channel (SSL) to the server and stored in the backend database. The subgraph vertices and edges are stored in the *nodeInfo* and *edgeInfo* tables.

The summary of the actions involved in user creation are:

- The user chooses a username and password and enters user information
• User’s software creates a subgraph for the user and stores the depth level keys in a separate file (this file will be stored locally or in an external hard disk)

• The user is allowed to choose a profile picture. This profile picture is encrypted using the master key before being sent to the server software for being stored (the table `contentInfo` gets updated).

• SSL will help establish a private communication channel that is valid for a certain time period (this will act as a session).

• User information and edges of the subgraph are sent to the server.

• The server software adds the user information to the database (the tables `userInfo`, `contentInfo`, `nodeInfo` and `edgeInfo` get updated). The password is concatenated with a random salt value. The server software stores the hash of this value and the salt instead of storing the original password. It will also generate a profile page with user information and sends it to the user software, where it is displayed to the user.

4.3.2 Login Process

A user who already has an account can login using his username and password. The user software generates a user login page where the user can enter his username and password. This process checks if the user already exists in the `userInfo` table.

A summary of steps involved in login process are as follows:

• The user software requests for username and password. This is sent to the server software. The server software computes the hash of this password information concatenated with salt (obtained from the database) and compares this value with the stored password in the database. The username is also checked with the value in the database.

• The server software retrieves the user information from the backend database. The profile page is sent to the user software in a secure manner. If the user doesn’t exist, a login error message is sent to the user software in which case the user should re-enter the user name and password, or create a new account if his account doesn’t already exist.
A number of preliminary operations are also performed. These are:

- Determining the friends of user who has logged in. This helps determine the users’ content and the depth at which they that can be accessed.

- Obtaining and decrypting the update messages (using private key) sent to the user who has logged in. This helps determine the relationship offers sent to the user.

### 4.3.3 Searching for users in the Social Network

This is used when a user has to look for other users in social network before offering relationship. This process looks up the `userInfo` table based on the username input and returns the matching user ID. The steps involved in searching are as follows:

- User sends the username of the person he/she is searching for to the user software.
- The user software sends this entered user name to the server software.
- The server software can connect to the database and obtain the user information of the user who is being searched. The Search module returns concatenation of the user ID and public key.

### 4.3.4 Offering Relationship

In order for a user Alice to establish a relationship with another user Bob, Alice has to send the derivation vertices, their corresponding keys and edges along with a message “want to be related as friends?” This information is used to update the `updateInfo` table. When the user Bob comes online at a later point of time, this request is displayed as an update in his profile page. He can see the request and can choose to accept/revoke the relationship offer. The steps involved are as follows:

- User software sends request to the server software to search for Bob, to whom offer is to be sent. If the user Bob exists, the user software requests for the subgraph of Bob. The server software returns the subgraph of Bob.
- User software creates derivation vertices, edges from derivation vertices to the content vertices (as shown in Figure 4.4). Derivation keys are also generated. User Software then uses all this information to create an update message (this includes the username of the
user Bob). This update message is encrypted with the public key of the user Bob. This is done in order to avoid revealing the derivation keys to the server. The encrypted message is sent to the server software.

![Diagram](image)

**Figure 4.4 Subgraph for user Alice assuming relationship offer is sent to user Bob**

- The server software posts the above information to the database (*updateInfo* table is updated)

### 4.3.5 Accepting Relationship Offer

When the user Bob logs in, the *updateInfo* table is checked for any relationship offer in the form of updates (as mentioned in Section 4.3.2). The server software retrieves these updates from *updateInfo* table and sends it to the user software. Once the user Bob is ready to accept the relationship offer, he can do so by adding an edge from his subgraph to Alice’s subgraph. The steps involved are as follows:

- The user software sends a request for the subgraph of user Bob to the server.
The server software returns the subgraph of user Bob to the user software.

The decrypted update message gives the username of user from whom offer was sent (Alice, in this case), the derivation vertices, derivation keys and edges between the derivation vertices and corresponding content vertices of Alice’s subgraph.

If the user Bob is willing to accept the relationship offer, the user software creates edges between Bob’s content vertices and derivation vertices (as shown in Figure 4.5). The user software creates a new update message including the received derivation nodes, edges, derivation keys and the newly created edges. This message is encrypted with the public key of the user Alice.

Encrypted information is then sent to the server software, which updates the tables `userInfo`, `nodeInfo`, `edgeInfo` and `relInfo`.

![Figure 4.5 Access Graph representing relationship between users Alice and Bob](image-url)
4.3.6 Removing Relationship

This involves removing the edge between the derivation and content vertices. Let us assume that Bob wants to stop being friends with user Alice. The steps involved are similar to that in accepting a relationship offer. They are as follows:

- The user Bob sends user name of the user Alice to the user software. The user software sends a request to sever software to confirm the existence of the user Alice. The server software uses the Search module in order to determine the user ID (UID) of Alice.

- The user software also requests for the access graph connecting users Alice and Bob from the server software.

- Once the access graph is obtained, the edges between the content vertices of Bob and the derivation vertices are identified and sent to the server software.

- The server software removes the edges that are sent by the user software including the derivation vertices.

- The user software then requests for content at each of the content vertices (pertaining to user Bob in the access graph) from the server software. These are decrypted with existing content keys.

- New keys are generated at each of these content vertices. These newly generated keys are used to encrypt the content depending on depth and finally sent to the server software.

4.3.7 Posting Content

The user is allowed to post content such as pictures. The steps involved are:

- User is allowed to choose the content (a picture in this case) and the depth at which it has to be posted in the user software. The user software encrypts the picture to be posted using the corresponding depth level key based on the depth chosen by the user. It then send this encrypted information, the meta information of the picture and depth to the server software.

- Server software stores this information contentInfo table from where other users (who are entitled to access the information based on depth) can access it.
4.3.8 Accessing Content

A user is allowed to access content of another user to whom he is related based on distance between the users. While a user at depth 1 (direct friend) can access content at depths 1, 2 and 3; a user at depth 2 (friend of friend) can access content at depths 2, 3 and a user at depth 3 (friend of friend of friend) can only access content at depth 3. The various steps involved are:

- User selects the username of user whose content he wants to access from the user software.
- *smartKey* is equipped to automatically determine the depth up to which content access is granted as mentioned above.
- The user updates his derivation keys with this selected user by sending a request for information on edges between the derivation vertices and content vertices to the server software. This is done in the event that the selected user has removed a relationship before this access event is processed.
- The selected username and selected depth are also sent to the server software.
- Server software returns the encrypted content (if present) to the user software.
- User software can use the derived key to decrypt the content.
- Decrypted content is displayed for the user to view if content is found at the selected depth. Otherwise, no content is displayed.

4.4 Logic Flow

This section provides a high level view of how *smartKey* and *secureSoftware* work together to store/retrieve data from the backend database. First, a demo of the entire implementation will be discussed. This is followed by a brief overview of the concepts related to database access and cryptography.

4.4.1 Demo

This section demonstrates the working of the entire system. The *smartKey* user interface is designed to be used with ease. It has a number of tabs, with each tab corresponding to a particular operation mentioned in Section 4.3. Let us assume a new user Amanda is trying to create an account. On starting the user software, *smartKey*, Amanda is presented with a login page as shown in Figure 4.6. Since Amanda is new, she can “Sign Up” for a new account. The
new user creation interface is as shown in Figure 4.7. A login page is generated for Amanda on successful account creation (Figure 4.8). Notice that she has no updates or friends at this point of time. Amanda can choose to offer relationship to an already existing user. This is done by clicking on the Offer tab (Figure 4.9) and typing in the name of the user to whom relationship offer is to be sent. Let us assume that another user Joel logs in and sends a relationship offer to Amanda. When Amanda logs in at a later point of time, she will see an update as shown in Figure 4.10.
Figure 4.7 User Interface for Signing up a new user

Figure 4.8 User Profile Page
Amanda can choose to accept/revoke this relationship offer sent by user Joel by going to the Accept tab (Figure 4.11). Checking the checkbox before click on Respond button will accept the
relationship offer, while leaving the checkbox unchecked will reject the relationship offer. Let us assume that Amanda accepted the relationship offer. Amanda can now access Joel’s content at depths 1, 2 and 3 since she is a direct friend. Notice the friends tab gets populated with usernames of users with whom Amanda is friends. Amanda can access Joel’s content by going to the Access tab as shown in Figure 4.12. Amanda can also post content at depths 1, 2, and 3 as shown in Figure 4.13.
If Amanda wishes to remove relationship with any user, she can do so by going to the Remove tab and typing in the name of the user with whom she is related as shown in Figure 4.14.
Refer to Appendix D for a more detailed explanation of the methods and classes of \textit{smartKey}.

4.4.2 Database Access

SQL Server 2008 is used as the DBMS. The database schema defined in Section 4.2 was designed in Visual Studio 2008 (the Visual Studio 2008 Integrated Development Environment comes with an easy-to-use GUI for creating a database schema). Simple SQL queries help access data stored in the tables. Refer to Appendix A for more information on how the database was connected to \textit{secureServer}.

4.4.3 Cryptography Used

The System.Security.Cryptography namespace that is a part of the .NET Framework Class Library helped include cryptography in both \textit{smartKey} and \textit{secureServer}. While symmetric key cryptography was used to encrypt/decrypt content, public key cryptography was used to exchange information between the users communicating with one another. Refer to Appendix B on more information about this namespace.

4.4.4 \textit{secureServer} Methods

\textit{secureServerSoftware} acts as the server software that performs the operations at the server side. It is essentially a Web Service that can be consumed by a client application like \textit{smartKey}. Such a Web Service uses XML to represent data and Simple Object Access Protocol
(SOAP) to transport the data. This service was published on IIS (Internet Information Service) running on a local machine [22]. Since the SOAP messages that are exchanged between the client (smartKey) and server (secureServerSoftware) are sent in the clear, it is possible for a malicious user to intercept the message and read the same. In order to avoid this situation, any communication between the client and server has to be encrypted. It is due to this reason SSL was configured on IIS. For this purpose, a self-signed certificate was created (using OpenSSL tool) and installed in the web browser [23]. The secureServerSoftware was changed accordingly in order to run on HTTPS rather than on HTTP [24].

secureServerSoftware connects to a backend database (on the same local machine) which is designed as shown in Figure 4.2. All the methods in this Web Service make use of SQL queries or stored procedures in order to store/retrieve data from the database tables. The login application in smartKey essentially calls the Web method addUser to add a new user account. The user software can request to determine the existence of a user by calling the searchUID method, which returns the user ID of the searched user. While communicating messages between users, the messages are encrypted with the receiving user’s public key. This public key is obtained using the getPublicKey Web method. The Web method loginUser is used to determine if the username/password combination matches the one stored in the database. On success, the loginSuccess Web method retrieves user information and sends it to smartKey which generates a user profile page.

The method postContent is used to store user content in contentInfo table while the method getContent is used to get user content based on depth. The methods getUpdateBySource and getUpdateByDestination are used to get update messages from updateInfo table based on whether the information request is from a user receiving the updates or the user sending the updates. User subgraph information is returned by the method OfferRel while the list of friends of a particular user is determined by the method getFriends. The method insertDerivedNodesAndEdges is used to update the tables nodeInfo, edgeInfo while accepting a relationship offer.
5. Experiments and Test Results

5.1 Evaluation

This section describes the evaluation of smartKey. A number of experiments were performed in order to evaluate the performance of smartKey. The experiments were created for measuring the performance of the following operations:

1. Offering a relationship
2. Accepting a relationship
3. Posting Content
4. Accessing Content
5. Removing a relationship

Specifically for each of these operations we measured the maximum time, minimum time, mean time for a random pair of users. A total of 1000 operations were performed in order to determine the client time, server time and total time. The experiments were run using a machine with following properties:

- Operating System – Windows XP Professional SP3
- Processor – Intel Core 2 Duo, E6750 @ 2.66 GHz
- RAM – 1.96 GB
- Development Environment – Visual Studio 2008
- Language – C#
- Database – SQL Server 2008

As mentioned in Appendix D, in order to avoid latencies due to cold start, a total of 1010 runs are made. Thus the final calculation avoids the first 10 runs and calculates maximum or minimum or average on the remaining 1000 values.

We perform three different types of experiments:

- By varying network size – in this case, 3 different types of datasets were considered. These vary in size and are named lowlevel, midlevel and highlevel datasets.
o LowLevel dataset represents a real-world dataset with 33 users and was obtained from *prefuse* [13].

o MidLevel dataset represents a real-world dataset with 129 users and was obtained from *prefuse* [13].

o HighLevel dataset represents a synthetic dataset with 1000 randomly generated users. These random users were generated using a random name generator in the user software.

- By varying size of data posted/accessed – three different sizes are considered
  - Order of 1 KB (any size < 5KB)
  - Order of 10 KB (any size <50KB)
  - Order of 100 KB (any size <200KB)

- By varying depth or distance between users. Depths considered are 1, 2 and 3.

### 5.2 Test Results

#### 5.2.1 Offering a relationship

This experiment was performed to determine the effect of network size on the performance of the offer operation (between a random pair of users). In this experiment, the Offer operation is applied to a pair of random users. The total average execution time (time taken for the entire operation includes server time and client time) for the three network sizes is shown as in Figure 5.1.

![Total Execution Time - Offer Relationship](image-url)
While offering a relationship, the user offering the relationship can request the subgraph of the user to whom the relationship request is to be sent. As the network size increases, it is logical to expect the system to take more time to search through the database. However, when an index is applied, the network size has little effect on the execution time of the Offer experiment.

### 5.2.2 Accepting a relationship

This experiment was performed to determine the effect of network size on the performance of Accept operation. In order to carry out this experiment, a pair of random users are generated and a relationship offer is sent from one user to another (the process of sending a relationship offer is also timed). The accept operation is then performed. It can be seen from Figure 5.2, that this experiment indicates that performance is independent of the network size. This is because; this operation mainly involves getting the user subgraph information and the update message.

![Total Execution Time - Accept Relationship](image)

**Figure 5.2 Execution Times while Accepting a Relationship**

#### 5.2.3 Posting Content

This experiment was performed in order to determine the effect of network size, depth and content size on the performance of Post operation. In this experiment, a random user posts content at depths varying from 1, 2, 3 and at varying network sizes. The results of these nine
experiments are as shown in Figure 5.3 (HL refers to HighLevel network, ML refers to MidLevel network and LL refers to LowLevel network).

Notice that as the network size increases, the execution times remains similar. This is because this operation essentially sends information to be stored in the database and is independent of the size of the network. A similar trend is seen on varying depths.

We now consider the effect of content size on the performance. In this experiment a random user posts content in a HighLevel network and varies the size from 1KB to 10 KB to 100 KB. The user then varies the network size and performs similar operation. The results are as shown in Figure 5.4.
From the above chart it can be seen that irrespective of the network size, the total time taken to post content increases with increasing content size.

5.2.4 Accessing Content
This experiment was performed to evaluate the performance of accessing content with varying network sizes and content sizes. This operation involves a random user trying to access content of another user at depths 1, 2 and 3 depending on distance between the users. For this operation, a random user, user1 is chosen, and another user, user2 is chosen from the list of friends of user1. User1 is now allowed to access the content posted by user 2 when the Access operation is applied. Similarly, users are chosen at depths 2 and 3 in a random manner and the access operation is applied. The total average execution time for the three network sizes when accessing content at depths 1, 2 and 3 are as shown in Figure 5.5.
Notice that when depth is kept a constant and network size is varied, the execution time increases with increase in network size. This is because, while determining the number of hops between a pair of users, a breadth first search is performed. This is an expensive process and hence the overhead. Consider varying depths at constant network size. It can be seen that as the distance between the users increases, the time taken to access content also increases. As mentioned earlier, breadth first search determining the number of hops between a pair of users causes the total execution time to be high.

Consider a similar scenario where size of the content is varied while network size and depth are kept constant.
The above chart is based on the total average execution times while accessing content. From the above chart it can be seen that time taken to access content increases as size of content and depth increases irrespective of network size.

### 5.2.5 Removing a relationship

This experiment was performed to determine the effect of network size on the performance of Remove operation. The total average execution time is as shown in Figure 5.7. It can be seen that as the network size increases, the total average execution time also increases. This is mainly due to the breadth first search being performed while determining the number of hops between a pair of users. Notice that the derivation vertices, edges between the derivation vertices and content vertices are being deleted. Also, the user content is downloaded, decrypted, re-encrypted with new depth level keys and uploaded again.
Figure 5.7 Execution Times while Removing a Relationship

The above experiments and test results enable us to determine if our system can be applied on a social network. An increasing trend was initially seen in the time taken to offer a relationship when a linear search was performed on the database. This was overcome by indexing selected columns from the tables in the database. The Accept and Post operations take the same amount of time with varying network sizes and are hence scalable with increasing network size. The time taken to Access content is seen to increase with increasing network size. This is mainly due to the breadth first search being performed while determining the number of hops between a pair of users. This overhead can be overcome by making use of better search algorithms such as A* search or depth limited search. An increasing trend in accessing content is also seen with increasing content size because the content is downloaded. A similar trend is seen in the remove operation for similar reasons.

The above experiments were created to be run on network sizes varying to a maximum of 1000 users in the dataset. This can be extended to support many more users at the order of millions. Given the hardware and software environment as mentioned in this section beginning, it is possible for anyone to re-create the same system and hence run the above mentioned experiments to confirm the illustrated results. The current system is run on one machine which has the client front end connecting to the server software (Web Service) that is hosted on a local IIS. The server software connects to the backend database on the same machine. This can be
extended to make the IIS host the server software on Internet. Thus we can have multiple machines connecting to the server software. Also, the backend database can be maintained on a different stand-alone machine. Making such extensions will make the system more similar to current social networks that support millions of users.

6. Conclusion and Future work

As social networking is becoming more common, the need for privacy of user data is also increasing. This is a first step for providing user content protection in a social network using key management system. *smartKey* and *secureServer* are a part of such a system which can control what a user shares with his friend or friend of friend and so on. Future work includes implementing the extension, that hide the entire social network, that support multiple relationship types and that support relationship strengths.

*secureServer* could be optimized for faster performance. If *serverSoftware* was to use Windows Communication Foundation (WCF) as a means of talking to *smartKey*, the entire communication would be much faster. This is mainly because ASP .NET Web Service uses SOAP to communicate XML messages between the client and server. XML Serialization between the client and server for every communication adds overhead while this can be avoided in a WCF Service by exchanging simple binary data. A future work can be to learn WCF to detail and upgrade *secureServer* from ASP.NET Web Service to a WCF application. While this can help make the system perform faster, it cannot change the way the system works.

With Facebook becoming a famous social networking platform, it will be interesting to determine its performance against this system. Other future work is to develop a Facebook application that incorporates the features of this system and compare them.
7. References


8. Appendix

8.1 Technology Used for Development

8.1.1 Client Front End: User Software

The user software was created in the Visual Studio 2008 Integrated Development Environment (IDE) using C# as the programming language. The main reason for this choice is that .NET framework enables rapid development of rich Windows-based applications with support for user-interface controls. WPF (Windows Presentation Foundation), which is a component of Microsoft .NET Framework 3.5 provides support for better UI with vector graphics and hence was chosen over a normal WinForm application in Visual Studio IDE. Vector-based graphics allows the user-interface elements (like buttons, textboxes etc.) to scale to any size without distortion. WPF employs a language XAML (Extensible Application Markup Language) in order to link the user interface elements together. This is a declarative XML-based language which was pretty simple to learn. The .NET framework consists of the Common Language Runtime (CLR) as the core component. This basically defines the execution environment for any code written in Visual Studio IDE. The basic working is as follows

![Figure 8.1 Compiling C# code to Native code](image)

The C# compiler generates the Common Intermediate Language (CIL) from C# source code when the source code is compiled. This is also known as the Microsoft Intermediate Language (MSIL), is the lowest-level human-readable language. The generated CIL is finally taken up by the CLR and converted into code that is native to the operating system.

Since the XAML elements are directly mapped to CLR object instances, it was easier to design a good looking user interface and define functionality for the different user interface elements. Refer to [15] for more details regarding WPF.
8.1.2 .NET Web Services: ServerSoftware

The ASP.NET Web Services is used as a means of communicating between the user software (client) and the backend database. This basically acts as the server software. It has methods known as the ‘web methods’ which can be referenced from a local client application. For this purpose, a Web Reference has to be made from the user software. A Web Reference provides a proxy class which locally represents the functionality of the Web Service. Thus, it is possible to access the methods defined in the Web Service class by creating an instance of the Web Reference. Whenever a web method is called on the proxy object, the request is encoded as a SOAP request message. The proxy object processes the response message once processing is done at the service [20].

8.1.3 Backend Database

The Visual Studio 2008 IDE provides tools to connect to a database from within a project. It also provides the GUI for creating new tables and manipulating them. The backend database design has been illustrated in section 3. Such a database was created in order to store and manipulate user information. The article in [21] was used to create stored procedures specific to smartKey application. These stored procedures basically had the queries to be executed on specific tables of the database.

9. Appendix B

9.1 System.Security.Cryptography Namespace

The concept of namespaces was introduced in .NET Framework Class Library in order to organize code and avoid name collisions. It is a well known fact that a class is a section of code that defines an object and its behavior. However, problem arises when there are two classes with same names, for instance (Eg. There can be 2 classes named System, one specific to Graphics and another specific to File handling). In such a situation, it becomes tough for the compiler to determine which class a particular type (method or member) belongs to. The article in [17] provides an introduction to namespaces. According to this article, a namespace can contain other namespaces or types.
smartKey made use of the System.Security.Cryptography namespace in order to perform cryptographic operations. The article in [16] provides a list of classes, structures and interfaces that are a part of this interface. The 3 main classes that were used in smartKey are

- SHA256 Managed, which basically computes hash for input data
- RSACryptoServiceProvider, which provides asymmetric encryption/decryption with RSA algorithm
- RijndaelManaged, which provides symmetric-key encryption standard AES

The following subsections provides details of each of these classes and their methods that were used.

9.1.1 SHA256 Managed Class and Methods Details

This class produces hash data that is of size 256 bits. The method `ComputeHash` computes hash of data which has the property that 2 sets of data match if and only if their corresponding hashes match. The input to this method should be a byte array.

9.1.2 RSACryptoServiceProvider Class and Methods Details

This class was mainly used in order to generate public/private key pair and use the generated keys for public key encryption/decryption. Before generating keys, calls are made to methods in `CspParameters` class. This class contains parameters that are passed to the cryptographic service provider (CSP) for performing cryptographic operations. The article in [18] provides a listing of these parameters.

After creating a new instance of the RSACryptoServiceProvider class, the corresponding object can be used to access the public and private key information. This information is stored as an XML string by using the method `ToXMLString`. This method takes a Boolean value as a parameter to differentiate between the public and private key information (false – public key information is exported as XML string; true – private key is exported as XML string).

The encryption operation is performed by making call to the method `Encrypt`. This method takes in two parameters namely the plain text that has to be encrypted and a Boolean value indicating the requirement for optimal asymmetric encryption padding (OAEP). According to the article in [19], OAEP a masked data string is appended to the original plaintext as an
additional form of securing data. A malicious user is forced to have the plaintext message in order to construct a valid OAEP message. Similar to the encryption operation, the decryption operation is performed by making a call to the method \textit{Decrypt}.

\textbf{9.1.3 RijndaelManaged Class and Methods Details}

This class is a managed version of the Rijndael class, provides methods for Rijndael symmetric encryption algorithm. This class supports a key size of 256 bits. A symmetric key can be generated using the \textit{GenerateKey} method while an initialization vector (IV) can be generated by calling the method \textit{GenerateIV} method on an instance of this class. An encryptor has to be generated in order to perform encryption operation using the generated key and IV. It is done by calling the method \textit{CreateEncryptor} method, which takes the key and IV as parameters. The plain text is represented as a memory stream in order to perform encryption. Encryption is performed by making a call to the Encryptor in a cryptographic stream (a stream that links data stream or memory stream to cryptographic operations). This is because the CLR uses a stream-oriented design for cryptography. The decryption operation is similar to the encryption operation with call to \textit{CreateDecryptor}. 
10. Appendix C

10.1 User Interface Screenshots

Figure 10.1 Sample User Profile Page

Figure 10.2 Sample Friends List
Figure 10.3 Sample screen for Posting content

Figure 10.4 Sample screen for Offering Relationship
Figure 10.5 Sample user updates

Figure 10.6 Sample screen for Accepting relationship
Figure 10.7 Sample screen for Accessing content

Figure 10.8 Sample screen for Removing Relationship
11. Appendix D

11.1 smartKey Class and Methods Details

11.1.1 Graph Class

The smartKey implementation depends on creating a user subgraph for almost all its operations. Such a subgraph consists of nodes and edges between nodes. The graph class represents the user subgraph structure. It consists of methods to add a node and edge between a pair of nodes. Adding a node (AddNode) is as simple as adding a new node to an existing node list, while adding a directed edge (AddDirectedEdge) between a pair of nodes involves creating an edge label (based on source and target node) and then appending it to the graph. The graph class is related to the node, nodeList and edge classes as shown in Figure 10.1.

![Figure 11.1 Relationship between Graph, Node, NodeList and Edge Classes]

11.1.2 Node Class

The node class is used to create a node in the user subgraph. Each node has a data, type and a key associated with it.

- Data is used to represent the public node label. This is useful while adding edges between a pair of nodes
- Type refers to the type of the node. There are 3 types namely master, content and derived
• Key is a random array of bytes that is assigned to every node in the subgraph using the key allocation scheme defined in [6].

These data members are required in order to create a node. When a node is created, it is appended to a nodeList in order to determine its neighbors. Each subgraph consists of a set of nodes that are connected by directed edges. These set of neighboring nodes can be determined by using the method \textit{getNeighbors}.

\textbf{11.1.3 NodeList Class}

This class represents a set of nodes that are part of a user subgraph. It basically represents a collection of objects where the object is of type Node. This means, we can add, remove or manipulate the nodes in a nodeList.

\textbf{11.1.4 Edge Class}

An edge is of type String that can be added between a pair of nodes namely the source node and the target node. An edge is created (\textit{AddEdge}) based on the key derivation mechanism defined in [6]. For instance, for nodes source and target with their corresponding keys, the edge is calculated as

\[
\text{edge} = \text{targetKey} - H(\text{target} \ || \ \text{sourceKey})
\]

The \textit{SHA256} class (and its methods), which is a part of the System.Security.Cryptography namespace of .NET Framework class library is used in hash computation. Refer to Appendix B for more information regarding the \textit{SHA256} class and its methods.

\textbf{11.1.5 smyKey Class}

This class is mainly used to perform symmetric key cryptographic operations. The class \textit{SignUp.xaml.cs} depends on this class in order to assign random keys to every node in the user subgraph. The keys assigned (\textit{keyGen}) to every node are used to encrypt/decrypt content at the corresponding node in the user subgraph. This class has method to encrypt (\textit{encryptEdge}) content at a node using the key assigned to that node. The \textit{RijndaelManaged} class, which is a part of the System.Security.Cryptography namespace of .NET Framework Class library, is used to generate
keys and perform encryption/decryption operations. Refer to Section 8.1.3 for more information regarding this class and its methods.

The class profilePage.xaml.cs depends on this class and its methods in order to derive the key associated with a particular node in the user subgraph. This derived key can be used to encrypt/decrypt content at that node. Key derivation is done using the key derivation mechanism defined in [6]. This class also uses the encryptEdge method to post/access content at a node in the user subgraph.

Figure 11.2 symKey Class

11.1.6 rsaCrypt Class

This class represents public key cryptographic operations that can be performed while a message is being exchanged between a pair of users in the system. This is mainly done in order to hide information from the server and other malicious users. This class uses the RSACryptoServiceProvider class, which is a part of the System.Security.Cryptography namespace of the .NET Framework class library. The method assignParams is mainly used for initializing the parameters used for RSA encryption/decryption. Each user is assigned a private key and a corresponding public key (createKeys) while creating an account. All the information sent to the server is encrypted with the user’s private key. This helps hide the information from the server.

While exchanging information between a pair of users, the concept of public key cryptography is used. Consider the case where pair of users A and B is exchanging information. When user A wants to send information to user B, he can encrypt (encryptData) the information using B’s public key (which is obtained from the database). On receiving the encrypted message, user B can decrypt (decryptData) it using its own private key. This process hides the information.
from the server and other malicious users. Refer to Section 8.1.2 for more information regarding `RSACryptoServiceProvider` class, its methods and parameters.

![rsaCrypt Class](image)

Figure 11.3 rsaCrypt Class

### 11.1.7 Window1 Class

This class consists of methods to sign up a new user (`CreateUserBtn_Click`) or login (`LoginBtn_Click`) an existing user. Facility to change password (`forgotPwdLink_MouseLeftButtonDown`) is also provided in this class. An existing user can login using his username and chosen password. A successful login implies the username-password combination matches the one stored in the database. Notice here that user entered password is concatenated with the salt value (obtained from the database) and encrypted before checking against the stored password. A profile page is generated on successful login.

![Window1 Class](image)

Figure 11.4 Window1 Class

The user interface designed is easy to understand and use. The `CreateDynamicProgressBar` method is used to provide an animation for the progress bar while the user software waits for the server to determine if the username-password combination is
correct and return user information. The returned user information is used to generate a profile page for the user by making calls to the methods in Profile class.

11.1.8 SignUp Class

This class is used in order to create a new user account. User information such as first name, last name, email ID, username and password are sent to the server while creating a user account. A unique userID is generated (userIDGen) for every user. In addition to the user information, the user subgraph information (createSubGraph) is also sent to the server. User subgraph is created by calling methods defined in the classes graph, node, nodeList and edge. A subgraph consists of one master node and (L+1) content nodes (L is the maximum depth up to which the user is ready to share content) as defined in section 3. A total of (L+2) nodes and (2L+1) edges are created for every subgraph. Once the user information is sent to the server and stored, the user is allowed to pick a profile picture (uploadProfilePic). In the event that user refuses to choose a profile picture, a default profile picture is sent to the server. Following this, a profile page is generated for the user by making a call to the profile class.

![SignUp Class](image)

Figure 11.5 SignUp Class

11.1.9 ChangePwd Class

This class has methods defined that can be used in the event that a user forgets his/her password. The method (changePwdBtn_Click) defined in this class enables the user to choose a
new password. The new chosen password is sent to the server where it is concatenated with the existing salt value before being encrypted and stored in the place of original password.

![Figure 11.6 ChangePwd Class](image)

**Figure 11.6 ChangePwd Class**

![Figure 11.7 User Interface for Changing Password](image)

**Figure 11.7 User Interface for Changing Password**

### 11.1.10 Profile Class

This class is used to generate a profile page (genProfilePage) every time a user creates a new account or logs in using an existing account. A profile page is created using the user information returned by the server to the user software. The profile picture is decrypted using the master key stored in the local key file.
11.1.11 ProfilePage Class

This class has the methods that implement most of the operations mentioned in Section 4.3. Figure 10.9 shows the relationship between this class and the rest of the classes. Before a profile page is actually loaded, a number of initialization operations are performed by the user software. These are as follows:

- **getFriends** gets the list of friends for the user who’s profile page has been generated. It also populates the friendsList listbox in the user interface.
- **getBaseKeys** gets and creates a byte array containing the master and content keys from the local key file. These are called the ‘base keys’.
- **genSubGraph** generates a user subgraph and assigns keys to each of the nodes in the subgraph from the base keys.
- **getUpdates** populates the updatesList listbox in the user interface. An update could be of type request (denoting a relationship offer), response (denoting response to a relationship offer, can be either accepted or rejected) or post (denoting a friend has posted content for this user to access).
• prepareAccept generate combo boxes randomly (createCheckBox) based on the number of relationship offers and populates the accept tab in the user interface

![Figure 11.9 Relationship between ProfilePage class and other classes](image)

The user interface is designed to include tabs for every operation described in Section 4.3. The names on the tabs clearly indicate which operation they are a part of. Refer to Section 9.1 for more screenshots of the user interface. The OfferBtn_Click method allows a user to offer relationship to another user who is existing in the system. This method performs operations mentioned in Section 4.3.4. The update message (request/response/post) that is sent from one user to another is generally encrypted using the public key of the user to whom the message is being sent. However, the RSACryptoServiceProvider class doesn’t provide method to encrypt huge amounts of data. It is due to this reason, a symmetric key (session key) is generated using RinjdaelManaged class and is used to encrypt the actual message. This encrypted message and the symmetric key are finally enveloped using the public key of the receiving user before sending. The receiving user can decrypt the envelope using his private key to get the symmetric key and encrypted message. This encrypted message can be finally decrypted using the symmetric key. As mentioned above, whenever a user logs in, the prepareAccept methods creates dynamic combo boxes for the user to either accept or reject the request. These combo
boxes are populated in the Accept tab. The RespondBtn_Click button allows the users to accept the offer if the combo box is checked, otherwise the offer is rejected. This method performs operations mentioned in Section 4.3.5.

A user can post content by going to the Post tab in the user interface. While posting content (PostBtn_Click), the user is allowed to choose the depth at which it is posted and the content itself according to the operation specified in Section 4.3.7. A user can access his/her friend’s content in the Access tab. This tab has been designed to dynamically populate the friends and the corresponding depths up to which they allow content access. For instance, for if Bob is a direct friend of Alice, then Alice can access content at depths 1, 2,..etc. Such an interface forces malicious users from trying to access content of other users’ at depths they are not allowed to. The operations specified in Section 4.3.8 are used to define the method accessBtn_Click.

As mentioned in Section 4.3.6, a user can stop a relationship with another user. This is enabled by removing a relationship (removeBtn_Click) between a pair of users. A user can specify the name of the user with whom he wants to be related in the Remove tab.

11.1.12 Splash.xaml.cs

When a WPF application is launched for the first time after reboot, it needs to access the disk to load many of Common Language Runtime (CLR) and WPF code pages that otherwise may be present in the OS memory manager's standby list. This slows down the WPF application. One of the ways to improve the performance of a WPF application is to make use of a Splash Screen. Such a splash screen helps load the WPF user interface components before they are actually used in the application. This also enables the application to run smoothly without giving the impression that the application is stalling with the back-end processing. It is due to this reason, the Splash.xaml.cs class was created.