EAI Endorsed Transactions on Collaborative Computing

Research Article ICST.ORG

SocialCloudShare: a Facebook Application for a Relationship-based Information Sharing in the Cloud

Davide Alberto Albertini[1], Barbara Carminati[1], Elena Ferrari[1]

[1]DISTA, Università degli Studi dell'Insubria, Via Mazzini 5, Varese, Italy.
{davide.albertini, barbara.carminati, elena.ferrari}@uninsubria.it

Abstract

In last few years, Online Social Networks (OSNs) have become one of the most used platforms for sharing data (e.g., pictures, short texts) on the Internet. Nowadays Facebook and Twitter are the most popular OSN providers, though they implement different social models. However, independently from the social model they implement, OSN platforms have become a widespread repository of personal information. All these data (e.g., profile information, shared elements, users' likes) are stored in a centralized repository that can be exploited for data mining and marketing analysis. With this data collection process, lots of sensitive information are gathered by OSN providers that, in time, have become more and more targeted by malicious attackers.
To overcome this problem, in this paper we present an architectural framework that, by means of a Social Application registered in Facebook, allows users to move their data (e.g., relationships, resources) outside the OSN realm and to store them in the public Cloud. Given that the public Cloud is not a secure and private environment, our proposal provides users security and privacy guarantees over their data by encrypting the resources and by anonymizing their social graphs. The presented framework enforces Relationship-Based Access Control (ReBAC) rules over the anonymized social graph, providing OSN users the possibility to selectively share information and resources as they are used to do in Facebook.

Received on 19 July 2014, accepted on 23 July 2014, published on 20 October 2014

Keywords: Online Social Networks; Collaborative graph anonymization; Controlled information sharing; Privacy-preserving path finding.

doi: 10.4108/ cc.1.2.e6

1. Introduction

In last years Online Social Networks (OSNs) have become one of the most common platforms for sharing data (e.g., pictures, short texts) on the Internet. Nowadays Facebook and Twitter are the most common OSN providers, though they implement different social models (e.g., supporting symmetric or asymmetric relationships). However, independently from the model they implement, OSN platforms have become a widespread repository of personal information. All these data (e.g., profile information, shared elements, users' likes) are stored in a centralized repository, not only to offer users a more customized experience on the OSN, but also to exploit them for data mining and marketing analysis. With this data collection process, lots of sensitive information are gathered by OSN providers that, in time, have become more and more targeted by malicious attackers.

Even though OSN providers give users' the ability to control how their information is shared over the platforms, this does not prevent them from collecting and profiling data. Literature presents several proposals aiming to prevent these marketing analysis. In general, these solutions imply to hide resources to OSN providers, e.g., by encrypting or by moving them to an external platform (see Section 7 for a more detailed discussion). All these proposals, thus, give users the ability to hide their resources from OSN providers, but do not avoid that OSN providers may infer users's personal information by analyzing, for example, the social graph.

This problem is further exacerbated by the fact that some well known OSN provider have not been always honest with respect to users privacy (see, for instance, [11] for a survey on these privacy concerns). Moreover, it occurred that OSN weaknesses brought to release as public some users' private data (e.g., the Google cyber attack in 2009 [13] or Google glitches [25]).

To overcome this problem, in this paper we present an architectural framework that, by means of a Social Application registered in Facebook, allows users to move their data (e.g., resources, relationships) outside

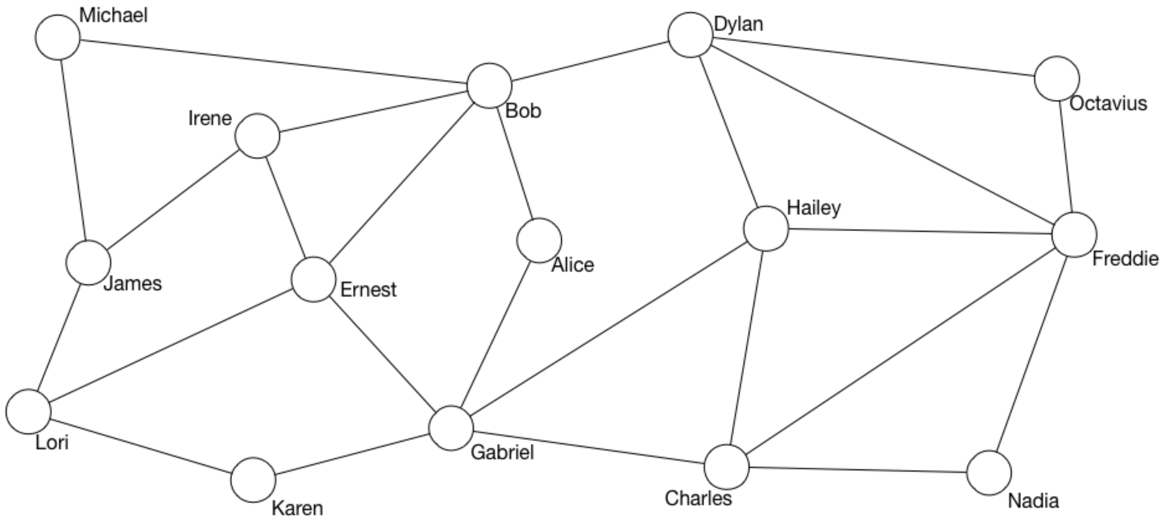


Figure 1. An example of social graph

the OSN realm and to store them in the public Cloud. Given that the public Cloud is not a secure and private environment, our proposal provides users security and privacy guarantees over their data by encrypting the resources and anonymizing their social graphs.

The presented framework enforces Relationship-Based Access Control (ReBAC) (see [5, 8]) rules over the anonymized social graph, granting OSN users the possibility to selectively share resources as they are used to do in Facebook. More precisely, the owner of a certain resource *rsc* can define *relationship-based access control conditions* that have to be verified in order to release *rsc* to the requestors. A relationship-based access control condition *acc* specifies type and depth of the relationship that must exist between the resource owner and the requestor to release *rsc* to the latter. More formally, an access control condition has the form $acc = (RelType, MaxDepth)$, where *RelType* is taken from a finite set of relationship type (e.g., *friend*, *relative*, *sibling*, *colleague*) and *MaxDepth* specifies the maximum number of hops that the shortest path connecting the owner and the requestor may be composed of. In this paper, we allow users to define access control conditions on their resources according to the ReBAC paradygm.

The proposed framework is based on the architecture described in [1], where anonymization and encryption techniques are accurately described. The work in [1], however, was tailored for a Decentralized Social Network (DSN) and, then, it suffers of all the limitations coming from a decentralized management of users data. In this paper, we present a *proof-of-concept* of such model, by implementing it inside Facebook.

The remainder of the paper is organized as follows. Section 2 presents an overall description of the proposal, while Section 3 illustrates the details of the architecture, along with a disussion describing how ReBAC is enforced over an anonymized social graph. Section 4 describes communication protocols, whereas Section 5 provides technical details of the current framework implementation. Section 6 deals with experimental evaluations. Finally, Section 7 gives an overview of the state of art, whereas Section 8 concludes the paper.

2. Overall Description

In order to highlight limitations of current proposals, we introduce a motivating example that reflects a real case of use of OSN functionalities.

Example. Let consider the simple social network represented in Figure 1, where where nodes represent users and edges represent *"friend"* relationships. Let assume that an OSN user, say *Ernest*, is willing to publish on his Facebook wallboard some pictures regarding a Christmas company party. As such, Ernest wishes to share those pictures only with the people working in his company, that is, with Gabriel, James, Karen, and Lori.

In a Facebook-style scenario, Ernest would not be able to keep track of the real-life relationships that he has with his colleagues. Then, in order to distinguish his colleagues from other contacts, Ernest would have to create a *group* or an *event* including all the people who take part in the party and then share the pictures with them. A simple relationship-based sharing, like the one offered by Facebook (e.g., friends or friends-of-friends), in fact, would not reach all the users of the network that attended the party. Indeed, with an "only friends" (OF) privacy setting, James and Karen, who are not directly connected with Ernest, would not be able to see the pictures. On the other hand, with a "friend of friends" (FoF) privacy setting, a larger set of users would be able to see them, including other people such as, e.g., Ernest's friends and relatives and their contacts.

Moreover, users who can see Ernest's pictures are granted the ability to share the pictures on their own wall, diclosing to the OSN community that there exists a connection between them and Ernest. Let assume, then, that pictures are uploaded with an OF privacy setting and Lori shares them on her wall. As such, even James and Karen would be able to see those photos, discovering there exists a connection between Ernest and Lori. This side effect may not be appreciated by Ernest, who may desire to keep this relationship private.

Finally, Ernest may have concerns publishing his pictures, in that he knows that all the published pictures are stored in an OSN repository that could be attacked by malicious users without any possibility for Ernest to prevent this event. The Social Network provider, actually, may try to infer data about Ernest for marketing purposes too and, still, Ernest would have no chances to prevent this profiling.

As highlighted by the example above, we identify three main issues underlying every OSN user experience: a limited set of privacy settings available in today OSNs for sharing resources, the possibility for both users and OSN provider to infer existence of relationships, and the lack of tools for users to control how their data are stored in the social network provider realm. To cope with these issues, we propose to export users' data (i.e., relationships and resources) from the OSN to an external platform, such as, the public Cloud, by, at the same time, enforcing a relationship-based access control more flexible than the one offered by OSN providers. Moreover, since the Cloud itself could act as a malicious party or, simply, it could be targeted by malicious attackers, data stored in the public Cloud have to be protected.

To achieve these requirements, in this paper, we present an implementation of the solutions proposed in [1] having the most popular social network, i.e., Facebook, as target. The framework presented in [1] allows users to share encrypted resources stored on the public Cloud, releasing decryption keys only to users that satisfy the corresponding ReBAC rule. The key management presented in [1] assures that resources can be encrypted/decrypted only at client-side, without disclosing any other information to the framework components. More details on the encryption scheme will be provided in Section 6.

Relationships data have to be processed in a different way with respect to users' resources. Indeed, in order to implement ReBAC, the framework needs to search for path existence in the social graph. Thus, in order to preserve users' privacy, this path discovering is performed on anonoymized structures, called *Anonymized Contact Lists (ACL)s*. More precisely, given a user u and the list of contacts, denoted as $CL^d(u)$, that are at a maximum distance of d-hops from u, the corresponding *Anonymized Contact List*, $ACL^d(u)$, is defined as the coefficients of the polynomial $P_u^d(x)$ whose roots are all and only the identifiers of users in $CL^d(u)$.[1]

By exploiting the ACLs, it is possible to verify the existance of a path of a given distance between two users. As example, if the identifier of a user v is a root for the polynomial whose coefficients are in $ACL^1(u)$ (i.e., $P_u^1(x = id_v) = 0$), it means that between u and v there exists direct relationship. However, since users have only a local view of the social graph (i.e., only their direct contacts), they are only able to compute their ACL^1. Indeed, they cannot retrieve enough information in order to compute any ACL^ns, where $n > 1$, on his/her own. Thus, in order to enforce a ReBAC model, a more complete view of the social graph is necessary, rather than the one offered by ACL^1s.

To overcome this problem, in [1] we propose a method to combine ACL^1s so as to compute such a global view of the social graph. This method is based on the consideration that, given a certain user u, his/her list of contacts $CL^2(u)$ contains all and only those users t such that there exists a user v contained in $CL^1(u)$, such that t is in $CL^1(t)$. Then, with an abuse of notation, we can denote $CL^2(u) = \bigcup_{v \in CL^1(u)} CL^1(v)$. Thus, by means of ACL^1s of all the direct contacts of u, it is possible to compute $ACL^2(u)$.

In particular, to compute the union, we exploit the polynomials property that, given two polynomials $p(x)$ and $q(x)$, the roots of the polynomial which results from their multiplication, that is, $r(x) = p(x) \cdot q(x)$, are all and only the roots in the union set between the roots of $p(x)$ and the roots of $q(x)$. Thus, to privately compute $ACL^2(u)$, it is possible to compute the multiplication of all the polynomials $P_v^1(x)$, where v is a direct contact of u, that is, $P_u^1(x = id_v) = 0$, in order to obtain $P_u^2(x)$. By means of this procedure, at last, it is possible not only to compute ACL^2s, but also to obtain ACL^ns, where $n \geq 2$. As such, in order to compute ACL^ns, where $n \geq 2$, no user interaction is required.

With reference to the social graph depicted in Figure 1, the proposed collaborative graph reconstruction procedure is executed as follows. Let assume that Karen makes use of $SocialCloudShare$ before any other user in her community. At registration time, Karen fetches

[1]This relies on the assumption that, for each user u of the social network, the OSN provider assigns him/her an unique identifier id_u, at registration time, which is true for any popular OSN.

her direct contact list CL^1(Karen) and anonymizes it locally, in order to compose ACL^1(Karen). Let assume, for instance, that CL^1(Karen) = [Lori,Gabriel]; then ACL^1(Karen) is given by a polynomial $P^1_{Karen}(x)$, whose only possible roots are the the values of identifiers of users in CL^1_{Karen}. Then, this ACL^1(Karen) is sent to the *SocialCloudShare* framework (see Figure 2).

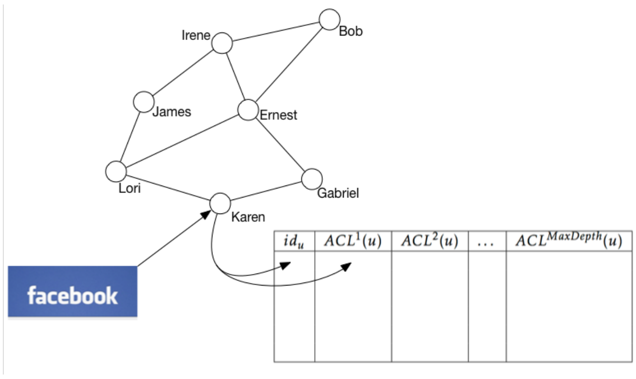


Figure 2. Example of the propagation procedure – phase 1

Assume now that, in a second time, Ernest makes use of *SocialCloudShare*. Similarly to Karen, Ernest fetches CL^1(Ernest) and anonymizes it, obtaining $P^1_{Ernest}(x)$. By having ACL^1(Karen) and ACL^1(Ernest) (see Figure 3), the framework[2] can combine them together so as to reconstruct the graph. More precisely, it has to discover if Karen and Ernest are friends. This can be done by evaluating $P^1_{Karen}(x = id_{Ernest})$, which will return a number $\neq 0$, since Ernest is not a Karen's contact. Then, no information propagation is necessary, in that no new relationship has been discovered.

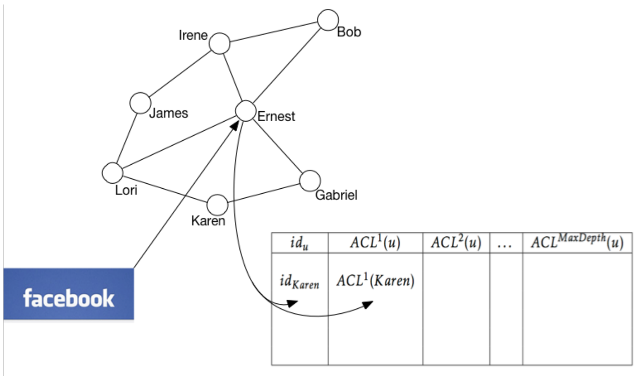


Figure 3. Example of the propagation procedure – phase 2

Finally, assume that Lori makes use of *SocialCloudShare*. Figure 1 depicts that Lori is both a direct contact of Karen and Ernest; as such, when the framework comes to evaluate $P^1_{Karen}(x = id_{Lori})$ the result will be 0. Then, the framework is able to determine

$$P^2_{Lori}(x) = P^2_{Lori}(x) \cdot P^1_{Karen}(x),$$
$$P^2_{Karen}(x) = P^2_{Karen}(x) \cdot P^1_{Lori}(x).$$

Then, the propagation procedure evaluates $P^1_{Ernest}(x = id_{Lori})$ and, again, the given result is 0; as such the information represented by ACL^1(Ernest) and ACL^1(Karen) has to be cross-propagated, resulting in

$$P^2_{Lori}(x) = P^2_{Lori}(x) \cdot P^1_{Ernest}(x),$$
$$P^2_{Ernest}(x) = P^2_{Ernest}(x) \cdot P^1_{Lori}(x).$$

Figure 4 illustrates the impact of these evaluations on the framework current state.

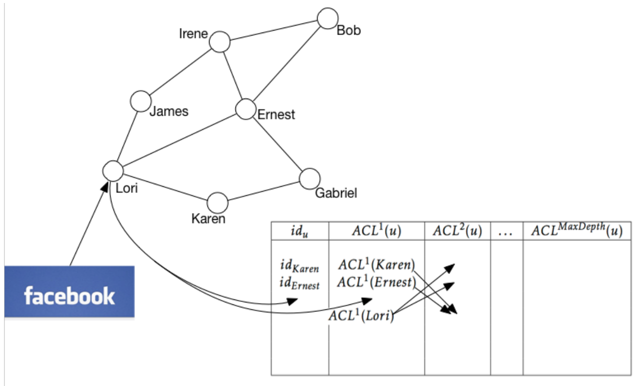


Figure 4. Example of the propagation procedure – phase 3

The procedure, then, continues in propagating users information in deeper levels until no futher computation is possible, that is, when the ACL^1(Lori) has been propagated to each of Lori's contacts and their ACLs have been propagated too (see Figure 5). As such, by exploiting the presented collaborative graph reconstruction, the framework is able to compute ACL^n, where $n \geq 2$, obtaining a social graph representation much wider than the one represented by only ACL^1.

id_u	$ACL^1(u)$	$ACL^2(u)$	$ACL^3(u)$	...
id_{Karen}	$ACL^1(Karen)$	$ACL^1(Lori)$	$ACL^1(Ernest)$	
id_{Ernest}	$ACL^1(Ernest)$	$ACL^1(Lori)$	$ACL^1(Karen)$	
id_{Lori}	$ACL^1(Lori)$	$ACL^1(Karen) \cdot ACL^1(Ernest)$		

Figure 5. Example of the propagation procedure – end phase

A more detailed description of the adopted techniques, along with a security analysis of the framework, can be found in [1].

[2]In Section 3 it will be explained which entity of the framework is in charge of this activity.

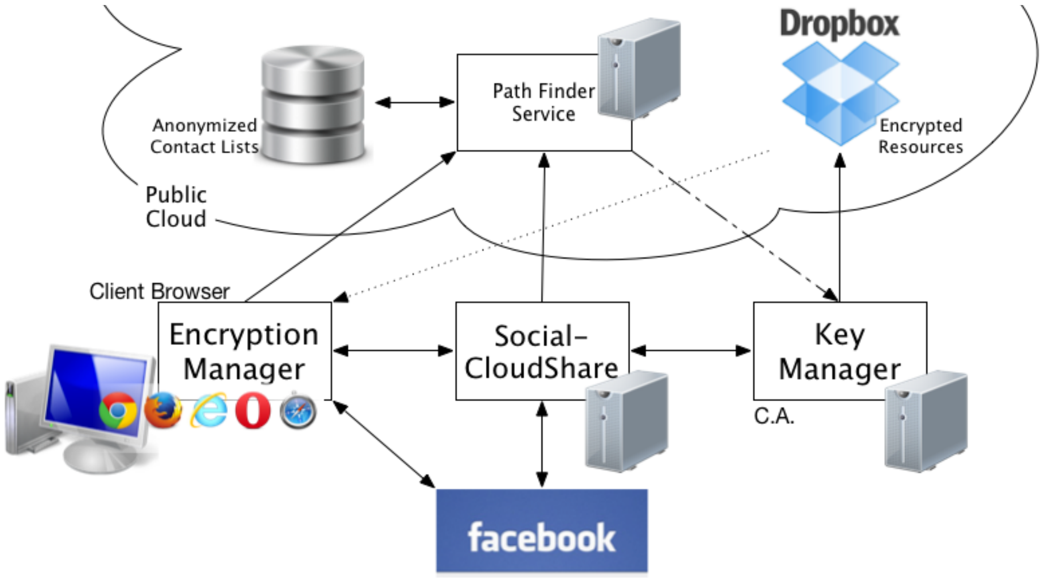


Figure 6. Framework architecture

3. Framework Architecture

According to the proposed architecture (see Figure 6), users' resources to be shared are locally encrypted by owners and stored into a Cloud storage (i.e., Dropbox). In support of this, we assume that the user is provided with the *Encryption Manager (EM)*, a browser plugin that is mainly in charge of owner's resources encryption and of generation of the *Anonymized Contact List* of user's direct contacts, that is, ACL^1. As described in Section 2, these structures are computed by anonymizing the information of CL^1, which is gathered directly from the OSN (i.e., Facebook). The channel between Facebook and the browser plugin is handled by the JavaScript Facebook SDK,[3] that allows the user to fetch structured data about the social graph (e.g., a contact list) directly from the OSN, without relying on any thirdy-party application.

ReBAC enforcement is carried out by releasing encryption keys only to those requestors that satisfy at least one of the owner's access rules. This enforcement requires the presence in the framework of two more entities. The first entity is a Social Application, named *SocialCloudShare (SCS)*, that provides users the possibility to manage access control rules and to share resources directly from the Facebook web page. The second is an entity, called *Key Manager (KM)*, in charge of the management of encryption keys.

Encryption keys are generated by exploiting two secret parameters: the first parameter, denoted with $secret_{owner}$, is unique per user and it is generated by $SocialCloudShare$; the second parameter, denoted with $secret_{rsc}$, is unique per resource and it is generated by KM. As such, this results in an encryption key unique per resource, that can be obtained only by combining the two corresponding secrets. As it will be discussed in Section 4, protocols regulating resources release have been designed so that neither $SocialCloudShare$ nor KM can decrypt owner's resources, as well as infer any information on owner's relationships. In particular, these are designed such that only EM is able to combine the encryption secrets; as such, $SocialCloudShare$ is not able to discover $secret_{rsc}$ values, while KM is not able to unveil $secret_{owner}$ values. This holds under the assumption that $SocialCloudShare$ and KM do not collude together. In support of this assumption, we assume that $SocialCloudShare$ is implemented on a tailored server and acts only inside the OSN realm, whereas the KM is an external trusted entity, whose role could be played by a Certificate Authority.

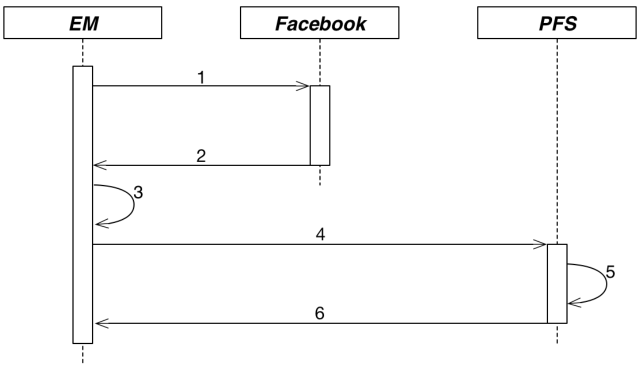
Moreover, to determine if a relationship-based access control rule is satisfied, it is required to find those paths in the social graph that connect the *owner* to the *requestor*. To protect relationships privacy, this path finding is carried out on ACLs stored in the public Cloud. This task is performed by the *Path Finder Service* (PFS) at Cloud side. As described in Section 2, ACLs are combined together to convey a deeper view of the social graph, with respect to the simple user local view that is represented with ACL^1. More precisely, for each user u PFS computes $ACL^d(u)$ representing the list of all the contacts that u can reach with a d-hop path.

[3]https://developers.facebook.com/docs/javascript .

4. Communication Protocols

Let us now introduce how the proposed framework enforces relationship-based information sharing, by illustrating the messages exchanged in each step. In doing that, we assume that the communication between entities is transmitted over secure channels.[4]

In this section, we will denote with K a symmetric encryption key, with K^+ and K^- a public and private key, and with $K^{session}$ a session key valid only for the current comunication session. For any key, we report as subscript the framework component for which the key has been generated (e.g., K^+_{SCS} denotes a public key generated for *SocialCloudShare*). Moreover, we denote with K_{rsc} a resource encryption key, whereas $secret_{owner}$ and $secret_{rsc}$ denote the secret tokens that are used for the generation of the resource encryption keys. Finally, with such defined keys, we denote with $\{message\}_K$ a message that is encrypted exploiting K as encryption key.



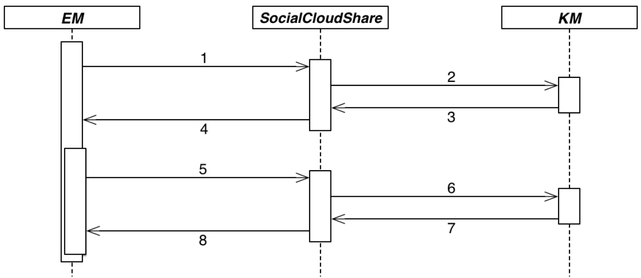
1. FB.api{ 'me/friends', {fields: 'id'} }
2. JSON-formatted CL^1
3. $ACL^1 :=$ anonymize(CL^1)
4. $\{id_{user}, ACL^1\}_{K^+_{PFS}}$
5. storeAndPropagate(id_{user}, ACL^1)
6. $\{$ "ACK" $\}_{K^-_{PFS}}$

Figure 7. Registration phase: messages Exchange

User Registration. Figure 7 depicts the messages exchange when users access *SocialCloudShare* for the first time. Exploiting Facebook JavaScript SDK, the user's contact list is requested (message 1) and gathered directly from the OSN (message 2) with no need to rely on any intermediate service. The anonymization process (message 3 in Figure 7) produces ACL^1 at user side, taking as input the direct contact list CL^1; as such, no relationship data are sent to the provider before being anonymized. The anonymized contact list is then sent to the PFS, which stores it and propagates in all the ACLs (see message 5 in Figure 7).

The messages exchange is ended with a response message produced by the PFS, i.e. message 6, to notify the EM that the protocol has been properly executed by both parties and the sent data have been successfully handled.

Login Phase. Since we assume that EM is not aware of $SocialCloudShare$ and KM public keys, the communication is initalized by requesting K^+_{SCS}, K^+_{KM}, where K^+_{SCS} and K^+_{KM} respectively denote the public keys of *SocialCloudShare* and of the *Key Manager* (see messages 1-4 in Figure 8).



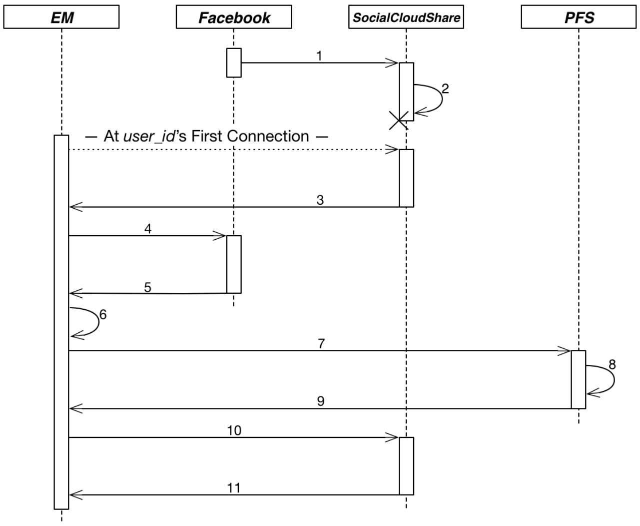
1. $\{id_{user}$, "PUBLIC_KEYS_REQ"$\}$
2. $\{id_{user}$, "PUBLIC_KEYS_REQ"$\}$
3. $\{id_{user}, K^+_{KM}\}$
4. $\{id_{user}, K^+_{KM}, K^+_{SCS}\}$
5. $\{id_{user}, K^{session}_{SCS}, \{id_{user}, K^{session}_{KM}\}_{K^+_{KM}}\}_{K^+_{SCS}}$
6. $\{id_{user}, K^{session}_{KM}\}_{K^+_{KM}}$
7. $\{id_{user}$, "SESSION_KEY_RECEIVED"$\}_{K^-_{KM}}$
8. $\{id_{user}$, "SESSION_KEY_RECEIVED", … …$\{id_{user}$, "SESSION_KEY_RECEIVED"$\}_{K^-_{KM}}\}_{K^-_{SCS}}$

Figure 8. Login phase: messages exchange

Once the user has received these keys, the EM generates a pair of 128 bit random keys, denoted as $K^{session}_{SCS}$ and $K^{session}_{KM}$, that will be exploited as session keys for the user current session. Note that, as depicted by the architecture in Figure 6, EM communicates with KM relying only on $SocialCloudShare$, since there exist no direct communication channel between the EM and the KM. Indeed, we adapted the structure of Needham-Schroeder protocol (see [23]). As such, when the EM has to communicate with the KM, it creates a message for $SocialCloudShare$ and encapsulates inside this the message directed to KM. $SocialCloudShare$, when receives such message, forwards to KM the encapsulated chunk (e.g., messages 5,6 in Figure 8). Assuming that only KM knows his private key K^-_{KM}, $SocialCloudShare$ cannot decrypt the encapsulated

[4] Beyond encryption primitives present in messages schemas, we assume that HTTPS connections can be instantiated before communicating, so that an additional security layer can be granted.

message, but it has just to forward it to the KM.[5] Once both $SocialCloudShare$ and KM correctly receive the session key, they reply to the user with messages 7-8 in Figure 8.



1. {´id’: ´user_id’, ´changed_fields’: ´friends’ }
2. storeUpdate(user_id, “CL_OUT_OF_DATE”)
3. { “CL_OUT_OF_DATE” }
4. FB.api{ ´me/friends’, {fields: ´id’} }
5. JSON-formatted CL^1
6. ACL^{1-new} := anonymize(CL^{1-new})
   $ACL^{1-removed}$:= anonymize($CL^{1-removed}$)
7. $\{id_{user}, ACL^{1-new}, ACL^{1-removed}\}_{K^{+}_{PFS}}$
8. updateAndPropagate($id_{user}, ACL^{1-new}, ACL^{1-removed}$)
9. $\{$ “ACK” $\}_{K^{-}_{PFS}}$
10. $\{$ “CL_UPDATE_DONE” $\}_{K^{session}_{SCS}}$
11. $\{$ “ACK” $\}_{K^{session}_{SCS}}$

Figure 9. Contact list update: messages exchange

Contact List Update. Figure 9 summarizes the messages exchange when the users’ contact lists are modified (i.e., by adding or removing relationships). In the current implementation, we exploit Facebook *Real Time Updates (RTU)*.[6] RTU is a feature of Facebook Graph API[7] which allows Facebook thirdy-party Social Apps to be informed, directly from the OSN provider, when certain pieces of data change (e.g., new profile pictures, new friendship requests). With this functionality, $SocialCloudShare$ does not need to continuously keep synchronized with the social graph, because a callback function is called, by means of an HTTP POST request, every time a user changes his/her own contact list (see message 1 in Figure 9).

Unfortunately, the OSN only notifies $SocialCloudShare$ about the changed fields, without revealing any other information. As such, it is then necessary to fetch from the OSN social graph all the data about new or removed friends. For this reason, we designed $SocialCloudShare$ to keep track of all those users whose contact lists are not synchronized with the $ACLs$ stored at Cloud side. Then, when each of those users makes use of $SocialCloudShare$, he/she receives a message that informs the EM that the contact list has to be synchronized (see messages 2,3 in Figure 9). Exploiting JavaScript functions, the current contact list is fetched from the social graph and new users (or, equivalently, removed users) are detected. CL^{1-new} and $CL^{1-removed}$ denote the two contact lists computed by the EM representing the lists of the new and the removed contacts. By exploiting the *anonymize* function in message 6 of Figure 7, CL^{1-new} and $CL^{1-removed}$ are anonymized.

Then, the user sends to the PFS these two separate $ACLs$ (or just one of them, in case the other one results in an empty list) (see message 7 in Figure 9). The PFS runs again the process of ACL propagation, adding the new information whenever these data are missing, or removing old information in case of relationship removal (i.e., by dividing polynomials instead of multiplying them). The messages flow is concluded with a special flag (see messages 9-11), in order to inform both the PFS and $SocialCloudShare$ that the protocol has been properly executed.

Resource Upload. The messages exchange for the resource upload phase follows the same schema as the one depicted in Figure 8, whereas the messages content is depicted in Figure 10.

1. $\{id_{user}, id_{rsc},$ “RSC_UPLOAD_REQ”$, \dots$
   $\dots \{id_{user}, id_{rsc},$ “RSC_UPLOAD_REQ”$\}_{K^{session}_{KM}}\}_{K^{session}_{SCS}}$
2. $\{id_{user}, id_{rsc},$ “RSC_UPLOAD_REQ”$\}_{K^{session}_{KM}}$
3. $\{id_{user}, id_{rsc}, secret_{rsc}\}_{K^{session}_{KM}}$
4. $\{id_{user}, secret_{user}, \{id_{user}, id_{rsc}, secret_{rsc}\}_{K^{session}_{KM}}\}_{K^{session}_{SCS}}$
5. $\{id_{user}, id_{rsc}, \mathcal{R}_{rsc}, \{id_{user}, id_{rsc}, \{rsc\}_{K_{rsc}}\}_{K^{session}_{KM}}\}_{K^{session}_{SCS}}$
6. $\{id_{user}, id_{rsc}, \{rsc\}_{K_{rsc}}\}_{K^{session}_{KM}}$
7. $\{id_{user}, id_{rsc},$ “RSC_STORED”$\}_{K^{session}_{KM}}$
8. $\{id_{user}, id_{rsc},$ “ACR_STORED”$, \dots$
   $\dots \{id_{user}, id_{rsc}$“RSC_STORED”$\}_{K^{session}_{KM}}\}_{K^{session}_{SCS}}$

Figure 10. Resource upload phase: messages exchange

Before uploading a certain resource rsc in Dropbox, its owner, say user u, has to encrypt it using a symmetric key, that is, K_{rsc}. This key is computed as combination of two secrets, i.e. $K_{rsc} = \mathcal{F}(secret_{owner}, secret_{rsc})$, which

[5]This still relies on the assumption that $SocialCloudShare$ and KM do not collude.
[6]https://developers.facebook.com/docs/graph-api/real-time-updates/v2.0 .
[7]https://developers.facebook.com/docs/graph-api/ .

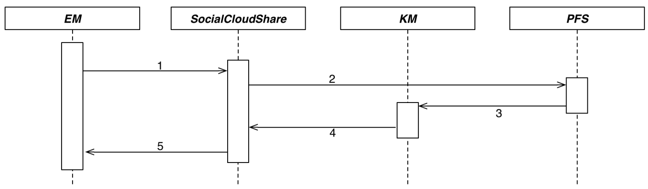
are separately generated by $SocialCloudShare$ and the KM.[8] In the given implementation, we choose to encrypt resources exploiting AES-256 algorithm [22], operating in Cipher Block Chaining (CBC) mode [10], where the plaintext is padded according to PKCS#7 [17]; for this reason we designed the two secrets with length of 256 bit. As it will be discussed later, these secrets are released to a requestor by $SocialCloudShare$ and KM if and only if he/she satisfies a t l east one access rule condition associated with rsc.

Thus, before any upload, resource owner has to interact with both $SocialCloudShare$ and the KM so as to retrieve the corresponding $secret_{owner}$ and $secret_{rsc}$. Assuming the user shares a symmetric session key only with KM, negotiated during the login phase, $SocialCloudShare$ cannot decrypt the encapsulated message and thus cannot discover $secret_{rsc}$. Once the secrets have been generated by $SocialCloudShare$ and the KM, they are received by u encrypted with pre-shared session key (see message 4 in Figure 10); as such, the user is able to compute K_{rsc}. Hence, u composes a message including the encrypted resource (to be transmitted to KM) and the set of access control rules $\mathcal{R}_{rsc}$ that $SocialCloudShare$ has to store. In our implementation $\mathcal{R}_{rsc}$ is a 1 byte value; the 5 more significant bits translate the relationship type (with a maximum of possible relationship types equal to 32) and the 3 less significant bits translate the maximum depth value of the access control condition. Even though our implementation currently supports only *"friend"* relationships, this implementative choice leaves the framework ready to further improvements.

As depicted in Figure 10, the EM sends all messages to $SocialCloudShare$, which then forwards nested encrypted messages to the KM. After the execution of the protocol illustrated in Figure 10, the Cloud data storage service contains the encrypted resource, whereas $SocialCloudShare$ and the KM contain only resource metadata. In particular, the KM stores id_{rsc} and $secret_{rsc}$, whereas $SocialCloudShare$ saves id_{rsc} along with the resource access control rules $\mathcal{R}_{rsc}$, where id_{rsc} denotes a unique identifier for the resource.

Resource Download. In order to enforce a relationship-based resource sharing, the framework has to release encryption keys only to requestors satisfying at least an access control rule associated with the requested resources. To determine if an access rule is satisfied, the PFS service is inquired. To protect the communication between $SocialCloudShare$, the KM, and the PFS we assume there exists a symmetric encryption key, denoted as K_{PFS}, shared between those three entities. By using this key, the communication encrypted with K_{PFS} cannot be decrypted by anyone unless the components of the framework.

If a requestor req wishes to download and decrypt rsc, it has to send a message to $SocialCloudShare$ with the related ids (message 1 in Figure 11). $SocialCloudShare$ retrieves the corresponding access rules $\mathcal{R}_{rsc}$ and the id of rsc's owner (i.e., id_{own}). Then, assuming for simplicity $\mathcal{R}_{rsc}$ contains only one access control condition $acc = (t, d)$, it inquires the PFS to search for a path connecting the requestor to the owner, with all edges labeled with t and length less than d (i.e., message 2 in Figure 11). It is important to note that if the PFS sends the yes/no answer back directly to $SocialCloudShare$, this might bring to some information leakage. Indeed, for some particular access rules, knowing whether the rule is satisfied gives exact information on existing paths. As such, the answer produced by the PFS is sent to the KM (see message 3 in Figure 11).



1. $\{id_{req}, id_{own}, id_{rsc}, \text{"RSC\_DWNLD\_REQ"}, \ldots$
   $\ldots \{id_{own}, id_{rsc}, \text{"RSC\_DWNLD\_REQ"}\}_{K_{KM}^{session}}\}_{K_{SCS}^{session}}$
2. $\{h(id_{req} \| acc_{rsc}.type), h(id_{own} \| acc_{rsc}.type), acc_{rsc}.depth \ldots$
   $\ldots \{id_{own}, id_{rsc}, \text{"RSC\_DWNLD\_REQ"}\}_{K_{KM}^{session}}\}_{K_{PFS}}$
3. $\{result, \{id_{own}, id_{rsc}, \text{"RSC\_DWNLD\_REQ"}\}_{K_{KM}^{session}}\}_{K_{PFS}}$
4. $\{token_{secret}, \mathsf{URL}_{rsc}\}_{K_{KM}^{session}}$
5. $\{id_{req}, secret_{own}, \{token_{secret}, \mathsf{URL}_{rsc}\}_{K_{KM}^{session}}\}_{K_{SCS}^{session}}$

Figure 11. Resource download phase: messages exchange

The URL sent from the KM (see message 4 in Figure 11) is a temporarily valid URL provided by the Cloud storage service upon KM requests. The rsc to be downloaded is reachable at this URL only for a small and fixed interval of time, afterwards rsc is moved back to the private realm of the storage service, without any public access.[9] Message 4 (see Figure 11) contains, along with the above mentioned URL, the value of $token_{secret}$, which is $token_{secret} = secret_{rsc}$ in case the PFS sent a positive answer, or a random value otherwise. $SocialCloudShare$ inserts $secret_{own}$ into the received message, encrypts it with pre-shared session key and forwards it to the user (i.e., message

[8]Several $\mathcal{F}$ functions can be adopted. In our implementation, we make use of XOR.

[9]Moving resources on temporary URL is a common approach used by several Cloud storage services (e.g., Dropbox, in this implementation) to limit access of requested resources.

5 in Figure 11). Then, the user decrypts $secret_{own}$ and $token_{secret}$ values and generates $\mathcal{F}(secret_{own}, token_{secret})$, which returns the correct encryption key K_{rsc} only if $token_{secret} = secret_{rsc}$, that is, only if the KM receives a positive answer from the PFS, confirming the existence of a path satisfying the rule.

5. Implementation

In this section, we provide some details concerning the implementation of *SocialCloudShare*.

5.1. Encryption Manager – Browser Plugin

The *Encryption Manager (EM)* is the component in charge of client-side resource encryption and of anonymized contact lists generation. We choose to implement the aboved-mentioned functionalities with a set of JavaScript functions, in order to achieve a better usability than a customized software and to give users the possibility to make use of it with no restriction given by his/her operative system.

By exploiting jQuery library[10] and AJAX-like[11] techniques, *EM* is able to process user actions (e.g., mouse clicks, page requests, upload/download requests) inside *SocialCloudShare*. The most important functionalities offered by *EM* are the encryption primitives for resources/messages encryption/decryption. For what concerns the resource encryption/decryption phase, the *EM* can be seen as a cipher black box. Thus, plaintext resource is taken as input and coded into a ciphertext resource and vice versa. As such, no entity except the *EM* takes part in these processes. At this purpose, we decided to exploit an existing library, named Crypto-JS,[12] available under BSD-3 License on Google Code, offering several encryption primitives ready to be used. In particular, for resources encryption, we exploit AES-256 algorithm applied according to Cipher Block Chaining (CBC) mode, where the plaintext is padded according to PKCS#7.

In order to exploit CBC mode, an Initialization Vector *iv* is necessary during the encryption and decryption phases. For this reason, the *EM* generates each time a random value as initalization vector (by exploiting *CryptoJS.lib.WordArray.random(128/8)*), which is added prior to the ciphertext, such that the *iv* itself can be securely stored along with the encrypted resource.

Figures 12 and 13 depict the functions used in the *EM* implementation.

[10]http://jquery.com/ .

[11]http://www.w3schools.com/ajax/default.ASP .

[12]https://code.google.com/p/crypto-js/ .

```
CryptoJS.AES.encrypt(
  'Resource-Stream',
  'Resource-Secret-Key',
  { iv: 'iv',
    mode: CryptoJS.mode.CBC,
    padding: CryptoJS.pad.Pkcs7
  }
);
```

Figure 12. Javascript AES encipher

```
CryptoJS.AES.decrypt(
  'Encrypted-Resource-Stream',
  'Resource-Secret-Key',
  { iv: 'iv',
    mode: CryptoJS.mode.CBC,
    padding: CryptoJS.pad.Pkcs7
  }
);
```

Figure 13. Javascript AES decipher

Another important feature handled by *EM* is the generation of ACL^1. To compute such ACL^1, the JavaScript library contains functions implementing the polynomial multiplication, i.e., computing the discrete convolution bewteen number sequences. As first step, the direct contacts list is fetched from Facebook social graph, by means of Facebook JavaScript SDK. As depicted in Figure 14, the JavaScript SDK needs to be initalized with a valid *Social-App-Id*, which is the identifier assigned by Facebook when registering a Social App inside its realm.

```
<script type='text/javascript'>
  $(document).ready(function() {
    $.ajaxSetup({ cache: true });
    $.getScript('//connect.facebook.net/en_UK/all.js',
      function(){ FB.init({
        appId: 'Social-App-Id',
      });
    });
  });
</script>
```

Figure 14. Facebook JS-SDK load phase

The *EM* can request to Facebook, by means of the Javascript FB Object, the logged user's friend list (e.g., see messages 1,2 in Figure 7) so that it can receive the current user's direct contacts identifiers. By having these identifiers, the *EM* can generate the user's ACL^1. Once this ACL^1 is fully computed, it is sent to the *Path Finder Service*, which is the component in charge of handling the anonymized social graph.

5.2. Path Finder Service

As outlined above, the *Path Finder Service (PFS)* is the component of *SocialCloudShare* that handles the anonymized social graph. All the *ACL*s are stored into the *ACL Repository* table, where the record is in the form $[id_u, ACL^1(u), ACL^2(u), \dots, ACL^{MaxDepth}(u)]$, that is, it contains the user identifier and all his/her *ACL*s of different path length (see example of *ACL Repository* in Figures 2, 3, 4, 5).

The *PFS* is implemented as a web service, by means of a Java servlets that handles HTTP requests. The request received from *EM* instances are encrypted with the *PFS* public key, i.e., K^+_{PFS}. On the other hand, requests received from *SocialCloudShare* entity are encrypted with a pre-shared session key, denoted as K_{PFS}, that grants a lower overhead than an asymmetric-key encryption.

Such component, like *SocialCloudShare* and the *KM* entities presented in the following sections, has been developed inside the Spring framework[13] and exploiting STS[14], an eclipse-based IDE.[15]

Algorithm 5.1 describes the procedure executed each time a new ACL^1 is received from a *SocialCloudShare* user. This algorithm makes use of the *ACL Repository*, denoted with *R*, and of a boolean matrix, *updates*, that keeps track of the *ACL*s that have been modified during the propagation procedure.

Each time a new ACL^1 is received, along with the user *id*, the *PFS* stores inside the *ACL Repository* those new information (see Line 3 in Algorithm 5.1) and sets as true the corresponding cell of the *updates* matrix (see Line 4). Once the data have been stored, the procedure analyzes, from the shallowest level to the deepest, the *ACL Repository* record (see Lines 5,7). We denote with *e.id* the user identifier stored in the repository entry *e*, and with $e.ACL^d$, the ACL^d stored in the same reporitory entry.

For each record *e*, the procedure performs a second iteration over all different record e' (see Line 8). If the *updates* matrix contains `true` in the cell corresponding to e', the procedure performs a polynomial evaluation, where the polynomial is the *ACL* taken from e' and the user identifier is taken from *e* (see Line 9). In case the polynomial evalutation results 0, and each polynomial evaluation for smallest path length (see Lines 10, 11) result in a value different from 0, the information carried by $e.ACL^1$ and $e'.ACL^1$ is cross-propagated to level $d+1$, where *d* is the variable iterated over the path depth values (see Lines 12, 13). Along with this cross-propagation, the procedure updates the values of the *updates* matrix, that is, it keeps track of the above modified entries. Finally, a boolean variable *stop*, initally set with `true` (see Line 6), is set with `false` (see Line 16).

The above described procedure terminates when, given a path depth *d*, ACL^ds are no more modified throughout the whole iteration over the repository records, that is, the boolean value of the variable *stop* is `true` when the loop cycle at Line 7 ends, and the procedure is forced to terminate (see Line 18).

Algorithm 5.1: ACL propagation procedure

Input: id_u, $ACL^1(u)$, *ACL Repository R*
1 **begin**
2   **boolean**[][] updates;
3   R.push({ $id_u, ACL^1(u), 1, 1, \dots$ });
4   updates[1][u] = `true`;
5   **foreach** $d \in \{1, 2, \dots, MaxDepth\}$ **do**
6   **boolean** stop = `true`;
7   **foreach** *entry* $e \in R$ **do**
8   **foreach** *entry* $e' > e$ **do**
9   **if** $(updates[d][e'.id])$ **AND** $(e'.ACL^d(x = e.id) == 0)$ **then**
10   **foreach** $d' < d$ **do**
11   **if** $e'.ACL^{d'}(x = e.id) \neq 0$ **then**
12               $e.ACL^{d+1} = e.ACL^{d+1} \cdot e'.ACL^1$;
13               $e'.ACL^{d+1} = e'.ACL^{d+1} \cdot e.ACL^1$;
14   updates[d+1][e.id]=`true`;
15   updates[d+1][e'id]=`true`;
16   stop = `false`;
17   **if** *stop* **then**
18   exit;
19 **end**

5.3. SocialCloudShare

Differently from the *PFS* and the *KM*, *SocialCloudShare* has been developed with both a back-end system and a graphical interface, which is displayed when users access *SocialCloudShare* inside Facebook.

SocialCloudShare back-end is implemented as a web application, designed according to the Model-View-Controller architectural pattern, such that a precise HTTP request on a given URL calls a certain method of the underlying servlet. The most relevant methods offered by *SocialCloudShare* are called by handling HTTP requests incoming on the following URLs:

[13]http://projects.spring.io/spring-framework/ .
[14]http://spring.io/tools .
[15]Spring is an application framework with built-in modules that facilitate Java application development, in which code dependecies are directly handled by Apache Maven (http://maven.apache.org/) and Gradle (http://www.gradle.org/) at build-time, generating a `.jar` archive that can run under, for example, an Apache Tomcat (http://tomcat.apache.org/) web server.

SCS/ : A request to the base URL of the web application generates and returns *SocialCloudShare* homepage. The underlying controller, when necessary, fetches and stores some of the users' data (e.g., full name, profile picture). These data are collected interacting directly with the OSN provider, exploiting Facebook Graph API in order to receive users' profile information.

SCS/key/broadcast : This URL is requested automatically when the *EM* detects that the public keys of *SocialCloudShare* and the *KM* are not stored at client-side. It represents the arrival point of message 1 in Figure 8. The controller forwards the received parameters to the *KM* on its URL *KM/key/broadcast*.

SCS/key/negotiate : This URL is requested automatically when the *EM* detects that the session keys for communicating with *SocialCloudShare* and *KM* are not stored at client-side. It represents the arrival point of message 5 in Figure 8. The controller forwards the message that is encapsulated in the received one, that is the message from *EM* to the *KM*, on the URL *KM/key/negotiate*.

SCS/fb_updates : This URL is reachable both with HTTP GET and HTTP POST requests, but it is supposed to be requested only from Facebook provider. The application listens to information from the OSN, waiting for *Real Time Updates (RTU)*. Once an HTTP GET request has been received, the application communicates with Facebook in order to control and regulate the subscription for RTU. HTTP POST requests, on the other hand, are assumed to include information about the user activity in the OSN (e.g., a new profile picture, a new friendship in the social graph). The controller underlying these requests keeps track of those users that have a contact list that is not synchronyzed with the *ACL*s stored in the *PFS* (e.g., see message 1 in Figure 9).

SCS/upload : The controller that handles HTTP requests to this URL is the one responsible of starting the resource upload procedure (see message 1 in Figure 10). The received message is decrypted with the corresponding session key, and the encapsulated message (that cannot be decrypted by *SocialCloudShare*) is forwarded to the *KM* on its URL *KM/upload*. Once the message is forwarded, the controller holds and waits for a response from the *KM*.

SCS/upload/finalize : A request done to this URL finalizes an upload procedure already started. As such, the underlying controller waits messages such as message 5 in Figure 10. Once received, the message is decrypted and the encapsulated part is forwarded to the *KM*. The remainder of the message, thus, includes the access control rule $\mathcal{R}_{rsc}$ of the uploaded resource. As such, $\mathcal{R}_{rsc}$ is stored by *SocialCloudShare* along with the resource owner identifier and the resource identifier.

SCS/download : The controller that handles the incoming requests to this URL is the one in charge of listening to download requests (e.g., message 1 in Figure 11), representing the initialization of a download process. As such, this message gathers information about the *requestor*, the *owner*, and the *resource* involved in the download process. These data are sent to the *PFS* that, after checking the existence of a path on *ACL*s, sends the corresponding result to the *KM* on its URL *KM/download*.

5.4. Key Manager

Similar to *SocialCloudShare*, the *KM* has been developed as a web application, exploiting the STS IDE. On the other hand, the *KM* is sligthly different from the previously presented *SocialCloudShare*. The *KM* is designed to listen to communication exclusively coming from *SocialCloudShare*, the *PFS*, and Dropbox; as such the application performs an IP filtering prior to accept incoming data. In case some data are received by a peer that is not regognized as belonging to one of those three parties, its requests are rejected and the communication channel closed. Then, the *KM* is designed without any front-end interface; any incoming message that is identified as valid brings the *KM* to perform some data processing and the output is directly sent as response to the message sender. The main methods offered by the *KM* are called by handling HTTP requests incoming on the following URLs:

KM/key/broadcast : This URL can only be requested by *SocialCloudShare* (e.g., via IP filtering) and it is requested only when *EM* detects that the public keys of *SocialCloudShare* and the *KM* are not stored at client-side. It represents the arrival point of message 2 in Figure 8.

KM/key/negotiate : This URL is requested automatically when the *EM* detects that the session keys for communicating with *SocialCloudShare* and *KM* are not present at client-side. It represents the arrival point of message 5 in Figure 8.

KM/upload : With reference to Figure 10, the controller handling these requests listens to messages such as message 2. The underlying methods are the ones responsible of generating and storing the resource secret $secret_{rsc}$, where rsc represents the identifier of the resource that is going to be uploaded .

KM/upload/finalize : The underlying controller includes the methods for resources upload to the Cloud Storage Service (e.g., Dropbox). Once a resource is stored into Dropbox, the *KM* locally stores certain resource metadata, such as the name, the filetype, and the last modification date. Those data are then displayed to users, through the *SocialCloudShare* GUI.

KM/download : This URL is listening only to requests coming from *PFS*. Indeed, the underlying controller is listening to messages resulting from a path search on the anonymized graph (message 3 in Figure 11). Messages received at this URL not only contain the result of the path search performed by *PFS*, but they include pieces of data that were included in the dowload request sent from the requestor. With these information, the *KM* is able to ask Dropbox to generate $\mathtt{URL}_{rsc}$, that is a temporary valid URL for the encrypted resource download. $\mathtt{URL}_{rsc}$ is included in the response along with $token_{secret}$, that may be a randomly generated value, in case the path finding returns a negative answer, or the value of $secret_{rsc}$ otherwise, where rsc is the downloaded resource.

6. Experimental Evaluations

To evaluate the framework performance, we carried out several tests. In doing that, we kept into account that the *PFS* efficiency has been studied in [1]. In particular, [1] presents the time needed by the *PFS* to perform polynomial evaluation and multiplication, that is, to verify a relationship-based access control rule and propagate an *ACL* throughout the *ACL Repository*. As such, in this section, we focus more on the overhead introduced by resource management, such as messages encryption size, messages encryption average time, and resource encryption average time. The workstation used for these experiments is an Intel Core 2 Quad Q6600 @ 2.40 GHz × 4, with 8GB RAM. In the current implementation *SocialCloudShare*, the *KM*, and the *PFS* are instantiated in the same instance of an Apache Tomcat servlet container and they run under different namespaces.

Message Encryption Size. Figure 15 depicts the overhead, in terms of length of messages, implied by messages encryption. The considered messages are those exchanged during the login, upload, and download phases (see Figures 8, 10, 11).

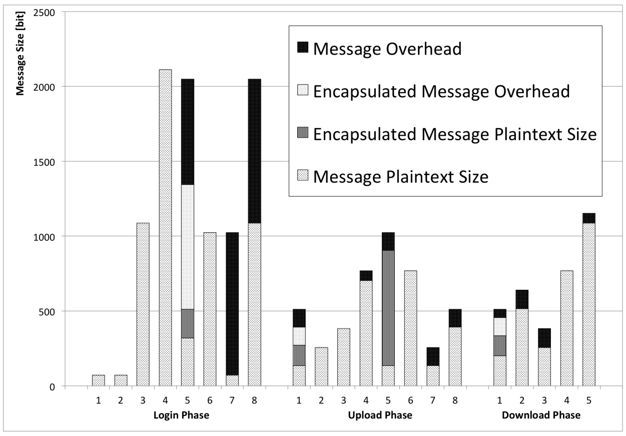


Figure 15. Messages encryption size overhead

Each bar in Figure 15 represents a single message, in term of message size. Each message is denoted by a *message plaintext size*, that is, the size of the message before encryption and a *message overhead*, that is, the size of the message once encrypted. Some message includes an encapsulated message (see message 5 in Figure 8, messages 1,2 in Figure 10, and message 1 in Figure 11) that requires a further encryption phase prior to message encryption. For those messages, Figure 15 reports the *encapsulated message plaintext size* as well as the *encapsulated message overhead*. As such, the *message plaintext size* for those messages is composed of the message plaintext size, the encapsulated message plaintext size, and the encapsulated message overhead.

Finally, it is important to note that messages in the login phase (see Figure 8) are encrypted using an asymmetric key encryption scheme.[16] This motivates the higher overhead introduced by messages encryption in such phase. Messages exchanged during upload and download phases, on the other hand, are encrypted exploiting AES-128 algorithm.

Message Encryption/Decryption Average Time. Tables 1 and 2 report the time consumption given by messages encryption. This experiment has been carried out monitoring the time required by encryption primitives to encrypt/decrypt the corresponding messages; for each message the encryption/decryption phase has been repeated 10 times. As such Tables 1, 2 report the minimum and the maximum time obtained in this experiment, along with the time average and the standard deviation.

With the exception of message 5 of the upload phase (Figure 10), the encryption/decryption primitives

[16] In particular, we exploit RSA-1024 for this phase .

Protocol	Msg	Time [ms]			Standard
		Min	Avg	Max	Deviation
Login	5	3.0	6.0	9.0	1.89
	7	1.0	2.2	3.0	0.6
	8	2.0	3.0	6.0	1.61
Upload	1	6.0	14.4	24.0	7.68
	3	3.0	8.1	12.0	3.01
	4	6.0	15.0	24.0	8.16
	5	14.0	28.0	56.0	16.57
	7	2.0	5.2	8.0	2.4
	8	4.0	8.0	16.0	4.0
Download	1	6.0	16.2	24.0	7.61
	2	5.0	8.0	20.0	4.58
	3	3.0	8.1	12.0	3.01
	4	6.0	13.2	24.0	7.96
	5	9.0	26.1	36.0	10.22

Table 1. Time required for message encryption

Protocol	Msg	Time [ms]			Standard
		Min	Avg	Max	Deviation
Login	6	4.0	4.8	6.0	0.98
Upload	2	4.0	10.0	16.0	4.1
	6	8.0	21.6	32.0	8.8
Download	2	4.0	9.2	16.0	4.02
	3	5.0	11.0	20.0	6.63
	4	5.0	10.8	20.0	6.38

Table 2. Time required for message decryption

File Size	Time [s]			Standard
[Mb]	Min	Avg	Max	Deviation
0.2	1.92	2.01	2.08	0.05
0.4	3.66	3.94	4.14	0.15
0.6	5.76	5.94	6.16	0.13
0.8	7.2	8.07	8.72	0.56
1.0	9.71	9.99	10.3	0.18
1.2	11.47	12.03	12.58	0.41
1.4	13.8	14.05	14.28	0.16
1.6	14.77	16.05	17.36	0.85
1.8	17.16	17.83	18.57	0.46
2.0	19.54	20.01	20.59	0.33
2.5	23.53	25.34	26.45	0.96
3.0	28.3	30.33	32.34	1.25
3.5	32.53	34.79	38.08	1.96
4.0	36.62	39.69	43.43	2.56
4.5	41.34	45.74	48.62	1.96
5.0	48.07	49.64	51.51	0.98
5.5	52.86	54.23	56.88	1.18
6.0	57.5	60.12	62.65	1.76
6.5	60.79	64.33	68.89	2.27
7.0	68.89	69.88	70.94	0.66
7.5	69.58	74.74	80.77	4.16
8.0	77.33	79.99	83.43	2.03
8.5	77.47	83.77	92.24	5.05
9.0	87.06	90.06	92.98	1.85
9.5	90.64	95.26	98.39	2.80
10.0	91.11	99.35	108.1	5.42

Table 3. Time required for resource encryption

completed the execution in less than 40 milliseconds. In this experiment, we used a 64byte text file as uploaded resource to keep the simulation as light as possible. The average time for all messages encryption/decryption, thus, never reached a value higher than 30 milliseconds; as such this result let us state that the protocols may run with no impact on the user experience over the OSN.

Resource Encryption Average Time. Finally, Table 3 reports the results of the experiment to estimate the encryption time necessary to prepare a resource to be uploaded. Unlike messages, which are encrypted exploiting AES-128 algorithm, resources are encrypted exploiting AES-256 algorithm, in order to achieve a better security for resources, that have to be stored in the public Cloud. The first column in Table 3 reports the size (in Mbytes) of the resource to be encrypted, while the other columns gather the time interval, in seconds, necessary to perform the encryption. In this experiment, we used random-generated ASCII strings, with pre-determined lengths. The encryption phase has been repeated 10 times for each resource; as such Table 3 reports the minimum and the maximum time recorded during the experiment, along with the time consumption average and the standard deviation.

With those experiments, and the ones previously reported in [1] about the *Path Finder Service* performances, we can thus state that SocialCloudShare causes a slight overhead over users experience in the Social Network. As such, we believe that protocols and techniques proposed in this paper would give a remarkable improval to OSN privacy measures.

7. Related Work

The presented work is mainly related to the following research topics: path-preserving graph anonymization, crypto-based access control, and privacy preserving in Online Social Networks.

Present literature includes many work proposing graph anonymization techniques. Most of these works can be grouped into two separate categories: those which propose node clusterization tecnhiques (e.g., [3, 6, 15]), and those which flatten the graph topology by modifying it (e.g., [16, 18]). However, these works, make the common assumption that the graph topology can be entirely read by a centralized party that anonymizes the graph.

A slightly different approach is described by Terzi *et al.* in [12]. In this work, authors present a collaborative anonymization procedure that exploits only nodes' neighborhood information. However, even this work, likely the works mentioned before, presents a technique that anonymize the graph by modifying its topology. Unfortunately, an anonymization techniques that contemplates a topology modification is not suitable for ReBAC enforcement. Indeed, introducing new edges in the graph may bring to harmful data release, whereas removing edges may cause not to release resource that should be released according to access control rules in place.

The only work presenting a path-preserving anonymization technique is, to the best of our knowledge, [4]. Authors in [4] present algorithms that allow to compute privacy-preserving operations without editing the graph structure. However, the path finding procedure presented in [4] can handle only paths whose lenght is ≤ 2. As such, none of these works propose a path-preserving collaborative anonymization procedure like the one presented in this paper.

Literature offers several proposals of crypto-based access control for cloud-centric platforms. Many recent proposals exploit attribute-based encryption (see [14, 21, 26]). Authors in [9] propose a solution for regulating access to outsourced data by means of a proper distribution of encryption keys. In recent works have been proposed OSN plugins (e.g., see Scramble! [2], FaceCloak [19]) that prevent OSN providers to performa data mining by analyzing users' data by encrypting them. As such, resources can be shared as encrypted data and decrypted only by those users who exploit the same platform that has been used for encryption phase. None of these works, however, target the enforcement of ReBAC.

A different approach, that can be exploited to prevent OSN analysis over shared data, is to move users' resources to a data repository separate from the OSN (e.g., see Lockr [24] or Trust&Share [7]), where the social network provider has no access. Still, those proposals treat only aspects related to shared resources, and do not take into account to hide relationship data from OSN managers.

8. Conclusions

In this paper, we present an implementation of the architecture presented in [1], where users' personal data are securely stored in public Cloud data storage and shared according to relationship-based access control rules defined by owners, tailored for the most popular of today OSNs, that is, Facebook. We plan to extend the work reported in this paper along several directions. First, we plan to extend the proposed privacy-preserving path finding to support more expressive access control rules. For instance, we intend to enforce also constraints on the trust of the required relationships. Moreover, we plan to improve the framework by implementing it in a distributed system, where the *Path Finder Service* is instantiated inside a Cloud provider realm (e.g., Amazon EC2).

9. Acknowledgements

The research presented in this paper was partially funded by the European Office of Aerospace Research and Development (EOARD) and the Air Force for Scientific Research (ASFOR). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

References

[1] D.A. Albertini, B. Carminati. Relationship-based Information Sharing in Cloud-based Decentralized Social Networks. In Proc. *ACM CODASPY*, 2014.
[2] F. Beato, I. Ion, S. Čapkun, B. Preneel, M. Langheinrich. For Some Eyes Only: Protecting Online Information Sharing.In Proc. *ACM CODASPY*, 2013.
[3] S. Bhagat, G. Cormode, B. Krishnamurthy, D. Srivastava. Class-based graph anonymization for social network data. In Proc. *VLDB Endowment*, 2009.
[4] J. Brickell, V. Shmatikov. Privacy-Preserving graph algorithms in the semi-honest model. In Proc. *ASIACRYPT*, 2005.
[5] G. Bruns, P. W. L. Fong, I. Siahaan, M. Huth. Relationship-Based Access Control: Its Expression and Enforcement Through Hybrid Logic. In Proc. *ACM CODASPY*, 2012.
[6] A. Campan, T. M. Truta. A clustering approach for data and structural anonymity in social networks. In Proc. *ACM PinKDD*, 2008.
[7] B. Carminati, E. Ferrari, J. Girardi. Trust&Share: Trusted Information Sharing in Online Social Networks. In Proc. *IEEE ICDE*, 2012.
[8] B. Carminati, E. Ferrari, A. Perego. Enforcing Access Control in Web-based Social Networks. In Proc. *ACM TISSEC*, 2009.
[9] S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati. Encryption policies for regulating access to outsourced data. *ACM Trans. Database Systems*, 2010.
[10] W. F. Ehrsam, C. H. W. Meyer, J. L. Smith, W. L. Tuchman. message verification and transmission error detection by block chaining. US Patent 4074066, 1976.
[11] Electronic Privacy Information Center, Facebook Privacy. online: *http://epic.org/privacy/facebook/*
[12] D. Erdös, R. Germulla, E. Terzi. Reconstructing Graphs from Neighborhood Data. In Proc. *IEEE ICDM*, 2012.

[13] Google official blog, A new approach to China. online: *http://googleblog.blogspot.com/2010/01/new-approach-to-china.html*. January 12, 2010.
[14] V. Goyal, O. Pandey, A. Sahai, B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Proc. *ACM CCS*, 2006.
[15] M. Hay, G. Miklau, D. Jensen, D. Towsley, P. Weis. Resisting structural re-identification in anonymized social networks. In Proc.*VLDB*, 2008.
[16] M. Hay, G. Miklau, D. Jensen, P. Weis, S. Srivastava. Anonymizing Social Networks. *Technical Report*, 2007.
[17] B. Kaliski. RFC 2315. online: *http://tools.ietf.org/html/rfc2315*, 1998.
[18] K. Liu, E. Terzi. Towards identity anonymization on graphs. In Proc. *ACM SIGMOD*, 2008.
[19] W. Luo, Q. Xie, U. Hengartner. FaceCloak: An architecture for user privacy on social networking sites. In Proc. *ICCSE*, 2009.
[20] M. Naor, B. Pinkas. Oblivious transfer and polynomial evaluation. In Proc. *ACM STOC*, 1999.
[21] S. Narayan, M. Gagné,R. Safavi-Naini. Privacy preserving EHR system using attribute-based infrastructure. In Proc. *ACM CCSW*, 2010.
[22] National Institute of Standards and Technology. Announcing the Advanced Encryption Standard (AES). online: *http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*, 2001.
[23] R. Needham, M. Schroeder. Using encryption for authenticating in large networks of computers. *Communication of the ACM 21*, 1978
[24] A. Tootoonchian, S. Saroiu, Y. Ganjali, A. Wolman. Lockr: better privacy for social networks. In Proc. *ACM CoNEXT*, 2009.
[25] J. E. Vascellaro. Google discloses Privacy Glitch. online: *http://blogs.wsj.com/digits/2009/03/08/1214/*.
[26] J. Zhang, Z. Zhang, A. Ge. Ciphertext policy attribute-based encryption from lattices. In Proc. *ACM ASIACCS*, 2012.