Distributed Privilege Enforcement in PACS

Christoph Sturm[1], Ela Hunt[2], and Marc H. Scholl[3]

[1] Department of Informatics, University of Zurich
sturm@ifi.uzh.ch
[2] Computer and Information Sciences, Strathclyde University
ela.hunt@cis.strath.ac.uk
[3] Department of Computer & Information Science, University of Konstanz
marc.scholl@uni-konstanz.de

Abstract. We present a new access control mechanism for P2P networks with distributed enforcement, called P2P Access Control System (PACS). PACS enforces powerful access control models like RBAC with administrative delegation inside a P2P network in a pure P2P manner, which is not possible in any of the currently used P2P access control mechanisms. PACS uses client-side enforcement to support the replication of confidential data. To avoid a single point of failure at the time of privilege enforcement, we use threshold cryptography to distribute the enforcement among the participants. Our analysis of the expected number of messages and the computational effort needed in PACS shows that its increased flexibility comes with an acceptable additional overhead.

1 Introduction

Peer-to-Peer (P2P) networks are used in many application areas. Especially in the data distribution (file sharing, IPTV) and data integration area including Peer Data Management Systems (PDMS) [1], their use introduced new challenges for access control. Because traditional access control mechanisms are based on a central authority, which is missing in a P2P environment, new approaches are needed.

In previous work [2, 3] we proposed an access control system for PDMSs without data replication. Here, we extend the application range of our P2P Access Control System (PACS) by supporting replication. Replication is a core feature in P2P applications. It enables fault tolerance in the face of disappearing peers and ensures load balancing within the network.

Replication in a P2P environment has several implications for the access control mechanism. First, as there is only limited trust between the participants, replicas cannot be distributed in plain text. The data is encrypted and stored on the replica servers without giving the replica server the possibility to decrypt it. This method is widely used in database-as-a-service approaches e.g. [4, 5]. Second, the enforcement of privileges has to be distributed, as otherwise the data owner remains the single point of failure and a potential bottleneck. Again, because of limited trust between the participants, the enforcement cannot be transferred to a single peer.

The contribution of this paper is to develop a distributed privilege enforcement mechanism for P2P networks, by extending PACS to support data replication. This implies the distribution of privilege management and privilege enforcement. The distribution of privilege management is already a key feature of PACS. Therefore, the main contribution here is to distribute privilege enforcement. To provide this, we use client-side enforcement, where each data object is encrypted with its own private key. The distribution and generation of these keys is done collaboratively by a group of peers (elected by the data owner) using threshold cryptography [6]. In this way, powerful access control models like Role Based Access Control (RBAC) [7] and administrative distribution are supported.

The paper is organized as follows. In Section 2 current access control approaches for P2P networks are presented. Section 3 introduces the access control component of PACS, together with the requirements for the data storage and the threat model. Section 4 describes the details of privilege enforcement in PACS. Thereafter, cost and performance estimations of the proposed enforcement mechanism are shown. The paper concludes with a discussion and future work.

2 Related Work

Research projects that address access control in P2P networks can be classified according to their distributed privilege enforcement method, as presented in the following.

The first option is to burden the data owner with privilege enforcement. This matches the classical server-side enforcement as the data owner is always the data provider. Enforcement is distributed in the network since the individual data owners, rather than a central authority, enforce the privileges on their data. When the data is replicated the enforcement of the privileges on that data must be done by the replication server. In a P2P network this is rarely practical, as trust between the participants is limited. Projects that use this enforcement technique are P-Hera [8], Berket et al. [9] and our own approach PACS [2, 3].

The second option is to enforce the privileges on the client. In this case, the data owner encrypts the data before it is stored on the replica servers. Only clients that possess the corresponding key can decrypt the data delivered by replica servers. The decryption keys are distributed by the data owner according to the privileges held by the clients (e.g. [5]). P2P approaches based on client-side enforcement were developed for P2P file systems. Sirius provides read and write access control [10]. Pacisso [11, 12] additionally supports owner changes and the possibility to define object groups. Both approaches operate at file granularity and do not support advanced access control models such as RBAC and administrative distribution. The access control mechanism proposed by Miklau and Suciu [4], designed for simple P2P file sharing, is based on encryption and key distribution. The privileges are defined using a logical model so that the granularity is not restricted to files. Again, no administrative distribution or

definition of groups and roles is supported. In client-side enforcement efficient key management is a problem, as a revoke triggers data re-encryption. Additionally, privileges are bound to decryption keys, which limits the access control models that can be supported. The advantage of this option is that replication of confidential data is supported even in unreliable environments.

Sandhu et al. [13] use trusted computing technology for access control in P2P networks to avoid the shortcomings of cryptographic enforcement. Their solution is also the problem, as a trusted reference monitor on the client is mandatory.

The third option is to transfer enforcement to a group of peers, called delegates in the following. The data owner elects delegates which enforce the privileges for a data object on its behalf. The delegates execute a t of n (where $t < n$) voting scheme to decide collaboratively whether a particular request should be permitted or not. The client can only access the data if t of n delegates approve the request. This mechanism strengthens the robustness of the enforcement mechanism and reduces the trust required in single delegates. Threshold decryption techniques [6] do not disclose the decryption key to the delegates or the clients. The delegates do the data decryption collaboratively and can enforce arbitrary privileges. The disadvantage is that the data has to be decrypted and delivered to the client by at least t delegates, which causes high communication costs. Pacisso [12] uses threshold signatures to enforce write permissions and Saxena et al. [14] show the use of this technique especially for mobile ad hoc networks. In addition, several research systems use threshold cryptography for access control enforcement in distributed databases (e.g. [15]), but none of them is applicable to P2P networks.

None of the presented approaches fulfills our requirement to support distributed enforcement for a powerful access control model and administrative delegation.

3 The Access Control Component

We first introduce PACS and then describe the distributed enforcement mechanism which is the main contribution of this paper. Our description abstracts from concrete access control models and data storages. In the following a user corresponds to a peer.

3.1 P2P Access Control System

PACS [2, 3] enables peers in a P2P network to establish global, decentralized access control. The access control mechanism is built bottom up, as single peers grant each other privileges on their private data. Through administrative delegation, privileges for non local data can also be granted. The privileges are stored in a distributed reliable directory, based on Castro et al. [16]. This guarantees that data stored in the privilege store is always available. PACS supports RBAC with administrative delegation and does not rely on any central authority or component. To make this possible, each participant needs a certificate

signed by a certification authority. The established public key infrastructure (PKI) enables secure authentication and ensures secure communication between the participants.

3.2 PACS Distributed Enforcement

To enable distributed enforcement in PACS, we propose a combination of delegate enforcement and client-side enforcement, as introduced in Section 2. The delegates do not decrypt the data object, but the data object decryption key. In that way we maintain the flexibility to enforce a variety of privileges and enable administrative distribution. The drawback is that clients get the decryption key and, therefore, the key generation and data re-encryption problem remains. In our solution, key distribution and key generation is done by the delegates collaboratively while data re-encryption is handled by the client that revokes a concrete privilege. We lose some of the flexibility of the delegate option, as we cannot use contextual information for authorization. Especially, it is impossible to support the evaluation of attributes during XACML [17] policy evaluation. These attributes can change anytime and the privilege enforcement component cannot keep track of these changes, as they are outside the scope of the access control component. Recall that every attribute change may result in privilege revocation, which must trigger new key generation and data re-encryption. In addition, if an application area relies on attribute evaluation, one can think of a wrapper that maps this attribute value to a permission inside PACS. In this way, PACS keeps track of the attribute changes and so key generation and data re-encryption can be triggered.

3.3 Data Storage

To make our approach more general, we separate privilege management and storage from data storage. Even though it is possible that both are the same, it is very likely that storing privileges requires higher security and is therefore more expensive. We make only basic assumptions about data storage. The storage has to provide a simple interface with two operations *put* and *get*. *put(ID, Data Object)* stores the data object under the given ID and *get(ID)* retrieves the data with the specified ID.

Internally, data storage can be based on a distributed hash table (DHT) [18] or any other storage structure. What is important is that each data object stored in the data storage is identified by a unique ID. As we operate in a P2P network, the network-wide data object ID consists of the unique peer ID of the data owner concatenated with the unique local name of the data object. The ID of the data object DO of peer $P1$ is defined as $ID_{DO_{P1}} = ID_{P1} + ID_{DO}$. To ensure basic security in the data store, data objects to be protected by PACS are inserted with a self-verifying identifier, ID^{SV}. An ID^{SV} for a data object is generated by creating a hash value over the concatenation of the object content and the object header $head_{DO_{P1}}$ as $ID^{SV}_{DO_{P1}} \equiv h(head_{DO_{P1}} + DO)$, where h is a secure hash function, such as SHA-1. The peers in the data store ensure that only data

objects with a correct ID^{SV} are stored. Clients that receive a requested data object can immediately verify whether the data object was changed without authorization.

Data objects protected by PACS have the following structure:

$$[head\{ID_{DO_{P1}}, ID_{DGIO}, V_{OKey_{DO_{P1}}}\}, DO]$$

Beside data content (which is normally encrypted), they carry a header that contains the $ID_{DO_{P1}}$, the ID of the Delegation Group Info Object (DGIO, explained in Section 4.3) and the version number of the object encryption key. This information is needed to check valid object deletions. As the self-verifying identifiers change with every data object modification, a special delete option is needed for outdated data objects. A deletion of DO is invoked by a $put(ID^{SV}_{DO_{P1}}, DGIO)$ call. Note that the verification of this request fails, as $ID^{SV}_{DO_{P1}} \not\equiv h(DGIO)$. But if the object is a new DGIO with the same ID_{DGIO}, including a new version number that is larger than the one stored in the current object, the request is accepted as a valid deletion. Before that, the correctness of the delivered DGIO object has to be checked by requesting the DGIO object from the privilege store.

3.4 Malicious Peers

As PACS is based on a P2P network that potentially contains malicious peers, we distinguish between good and malicious peers: a good peer always follows the specified procedures, otherwise it is a malicious peer. A malicious peer can act arbitrarily wrongly and can collaborate with other malicious peers. Data owners are assumed to work correctly as far as their own data objects are concerned. Authorized peers that revoke a privilege are assumed to execute the procedure properly. Only then the revocation is effective, which is in their own interest.

As PACS is based on secure routing [16], it can handle up to 30% malicious peers in the network. We assume that the fraction of malicious peers within a delegation group equals the fraction of malicious peers in the network. It is the responsibility of the data owner to choose the delegates and the size of the delegate group n accordingly. In addition, PACS requires that the threshold $t \geq \frac{n}{2}$ as a security buffer.

The goal of an attacking malicious peer is to gain unauthorized access to stored objects and/or to prevent other authorized peers from accessing stored objects. Attackers are assumed to have only limited computational resources and hence cannot break the underlying cryptographic schemes. This means that an attacker cannot generate valid signatures and encrypt/decrypt data objects without having the right keys. An attacker can eavesdrop the network traffic as long as it is not encrypted, but cannot block communication between any two peers. We ensure that the communication between two peers is secure, by signing and encrypting messages. Challenge-response rounds are used to prevent replay attacks. Furthermore, we do not control the information flow after the data is delivered to the client. The data responsibility of PACS ends there.

4 Distributed Privilege Enforcement in PACS

Distributed privilege enforcement is based on a combination of symmetric, asymmetric and threshold cryptography [19], as presented below.

4.1 Object Encryption

Privilege granularity in PACS is on the object level. This is not a restriction, as the object can be defined as a tuple in a database table or a music file. Every data object o is symmetrically AES [20] encrypted by an individual object key $OKey_{o_p}$ (128 bit length), where p is the data owner. Each $OKey_{o_p}$ has a version number $V_{OKey_{O_P}}$. The data owner p provides the asynchronous private master key $MKey_p^{-1}$ and the corresponding public key $MKey_p$. The $OKey_{o_p}$ is generated by signing the concatenation of the data ID_{o_p} and $V_{OKey_{o_p}}$.

$$OKey_{o_p} = sig_{(ID_{o_p}+V_{OKey_{o_p}})}^{MKey_p^{-1}} = (h(ID_{o_p} + V_{OKey_{o_p}}))^{MKey_p^{-1}}$$

PACS uses RSA [21] as its signature algorithm. Enforcement is done by a delegation group. The data owner distributes the private master key $MKey_P^{-1}$ using Shamir's (t, n) secret sharing technique [22] [4]. Using this, we assign a share s_i to each delegate d_i. The secret is described by a polynomial $f(x)$ of degree of at most $t-1$, where $f(0) = MKey_P^{-1}$. The polynomial can be reconstructed by every combination of t participants, using the Lagrange interpolation formula. All coordinates x_i are made public, whereas the generated shares are private. Share $s_i = f(x_i), x_i \in \mathcal{R}$ is delivered to delegate d_i encrypted with its public key, so only d_i can read it. Based on Desmedt [19], a message m is then partially signed by delegate d_i which computes

$$sig_m^{s_i} = (h(m))^{s_i} (mod\ n')$$

where n' is the RSA modulus of the public key $MKey_p$. A full signature of m, $sig_m^{MKey_p^{-1}}$, can be calculated out of t partial signatures by first calculating the coefficients of the polynomial as

$$k_i = \prod_{i=1;j\neq i}^{t} \frac{x_j}{x_j - x_i}, \forall i \in [1, \dots, t]$$

and then

$$sig_m^{MKey_p^{-1}} = \prod_{i=1}^{t} (sig_m^{s_i})^{k_i} (mod\ n')$$

[4] For RSA, some modifications are needed. See [23] for details.

Share Verification. Until now the generation of $sig_m^{MKey_p^{-1}}$ from the partial signatures fails if one of the t partial signatures is malicious. As we cannot verify the correctness of each partial signature on its own, it is very cumbersome to identify the malicious one and replace it with another correct partial signature from another delegate. To enable the verification of partial signatures, we use the robust threshold RSA method (see Gennaro et al. [24]). During share generation the data owner generates a random public sample message w and its corresponding partial signature $sig_w^{s_i}$ for all shares s_i. These partial signatures as well as the complete signature of the sample message $sig_w^{MKey_p^{-1}}$ are made public. The verification procedure is then as follows

1. The requester has received the partial signature of the message m, $sig_m^{s_i}$, from delegate d_i
2. The requester chooses two random numbers i, j and computes the challenge $Q = m^i \times w^j$ and sends this challenge to the delegate d_i
3. The delegate signs the challenge with its partial key $sig_Q^{s_i}$ and returns it to the requester
4. The requester now checks if $sig_Q^{s_i} = sig_m^i \times sig_w^j$. If this is true, the requester accepts $sig_m^{s_i}$ as partial signature of the delegate d_i

Share-Share Generation and Verification. A share s_i is only known to the delegate d_i and the data owner. When a delegate leaves the network unexpectedly and the data owner is not available, there is no possibility to restore the lost share. To cope with this, the so called share-shares are generated during share generation. For each share s_i a Shamir's (t, n) secret sharing schema is created. The procedure is as follows [12]

1. For each delegate d_i, the data owner takes its share s_i and applies the (t, n) secret sharing schema. The polynomial $f_i(x)$ is of degree $(t-1)$ and $f_i(0) = s_i$
2. The owner evaluates $f_i(x), x_i \in \mathcal{R}$ for n points $[x_1, \ldots, x_n]$ and gets as result n share-shares $[s_{i,1}, \ldots, s_{i,n}]$ with $f_i(x_j) = s_{i,j}$
3. The owner now sends to each delegate d_i the share-shares $[s_{1,i}, \ldots, s_{n,i}]$. So d_i gets as share-shares the points with the ith x value from every polynomial. For each share-share, hash values are generated and publicly stored. They are used for share-share verification during share reconstruction

When a delegate d_j is no longer available and the share should be reconstructed, every remaining delegate d_i takes its share-share $s_{i,j}$ and sends it securely to the new delegate. After the new delegate has received t valid share-shares (the share-shares can be verified using the generated share-share hash values), it can reconstruct s_i using the Lagrange interpolation formula. Thereafter, the old delegates send the new delegate their share-share $s_{j,i}$. After receiving all share-shares, the new delegate has also recovered the share-shares of d_j.

4.2 Making a Data Request in PACS

The data request procedure is shown in Figure 1. We illustrate the case of the first request for a particular object made by a client. Note that we make a distinction

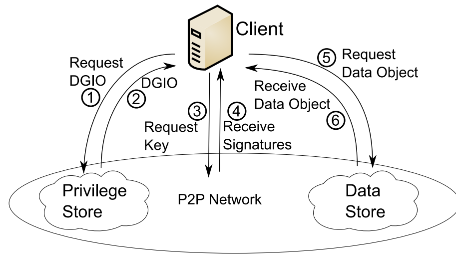

Fig. 1. A Data Request in PACS

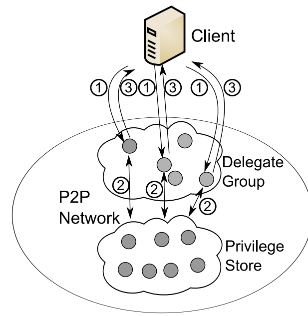

Fig. 2. Delegates Process an Access Request

between the privilege store and the data store. This is only a logical difference, as the two stores may run different storage protocols and different properties on the stored data can be guaranteed. So the same peers may participate in both logical networks. The client starts by sending a request for the Delegation Group Info Object (DGIO) for a particular object (Figure 1, step 1). The ID of the DGIO is derived from the object ID. The DGIO stores the information about the delegation group responsible for the object (for details see Section 4.3). Having received the DGIO, the client now knows which delegates to contact to get the decryption key. The decryption key in PACS corresponds to a threshold signature [6] over the object ID together with a version counter (cf. Section 4.1). The client has to ask a sufficient number of delegates to return a partial signature (steps 3 and 4). This is shown in detail in Figure 2. The client p sends the request to several delegates (step 1). Each delegate individually verifies the privileges held by p by requesting those from the privilege store (step 2). A good delegate returns the partial signature only if this verification succeeds (step 3). Privilege verification is executed by the delegate in the same way as a server does it in the sever side enforcement. After the client receives enough partial signatures, it constructs the complete signature that corresponds to the $OKey$ (see Section 4.1). Having the key, the client decrypts ID^{SV} from the DGIO and requests the encrypted data object from the data store. After receiving the object, the client decrypts the data and the request is satisfied. If the key is already known to the requesting client (caching), steps 3 and 4 in Figure 1 are skipped.

4.3 Delegate Management

Delegate management is part of privilege management. All information about the delegates is stored in a Delegation Group Info Object (DGIO). Because the information is as critical as the privileges stored in PACS, it is also stored in the privilege store. To achieve maximum flexibility, we allow a peer to have multiple delegation groups (e.g. one can think of a DGIO for every object). The peer just has to compute a unique master key pair $[MKey_p^{-1}, MKey_p]$ for every delegation group. To keep the description simple, we assume for the rest of the paper that

every peer has exactly one delegation group and therefore one delegation object. There exists an agreed naming schema for the DGIO so that every interested peer can compute the ID_{DGIO} of the right DGIO, using the data object ID.

Delegation Group Info Object. The DGIO object consists of a header part, an object part and a delegate part. The header is defined as:

$$[ID_{DGIO}, V_{DGIO}, sig_{DGIO}^{MKey_p^{-1}}, MKey_p, ID_p, PKey, w, sig_w^{MKey_p^{-1}}]$$

where $sig_{DGIO}^{MKey^{-1}}$ is the signature of the whole DGIO (excluding the signature itself), which prevents unauthorized changes. $PKey$ is the public key of peer p and w is the sample message needed for partial signature verification. As every delegation group has its own master key, the public master key $MKey_p$ is also included. V_{DGIO} is the DGIO version number required to detect stale DGIO copies.

The object part contains the object entries the delegation group is responsible for. A single object entry is defined as $[ID_{o_p}, enc_{ID_{o_p}^{SV}}^{OKey_{o_p}}, V_{OKey_{o_p}}]$. $ID_{o_p}^{SV}$ is the actual self-verifying identifier needed to retrieve o_p from the data store. It is stored in the DGIO to ensure that only authorized peers can create a new version of the object. $ID_{o_p}^{SV}$ is encrypted with $OKey_{o_p}$ to ensure that only authorized peers can see it. ID_{o_p} is stored for the $OKey_{o_p}$ creation together with a version number of the object key $V_{OKey_{o_p}}$.

The delegate part contains delegate entries. There is one delegate entry for each group member. A delegate entry is defined as:

$$[ID_{d_i}, DKey, x_i, sig_w^{s_i}, \{x_{i,1}, \ldots, x_{i,n}\}, \{h(f_i(x_{i,1})), \ldots, h(f_i(x_{i,n}))\}, active]$$

An entry includes the ID_{d_i} of the delegate (so that one can contact it directly), its public key $DKey$ and which x_i coordinate is assigned to it as share s_i. The share-share coordinates $x_{i,j}$ and the hash values of the share-shares are also stored here. Lastly, the status of the delegate (active/deleted) is noted.

Handling of Delegate Changes. Delegates are peers in a P2P network. They may disappear without informing the other delegates in their group. If a remaining delegate d_i realizes the absence of a delegate d_j in its group, it initializes a delegate recovery procedure. To achieve this, d_i initiates a quorum (again, t of n) decision by contacting all remaining delegates to get their agreement that

- delegate d_j is no longer available and should be marked deleted in the DGIO
- d_{n+1} should be the new delegate that substitutes d_j

d_j is marked deleted to exclude this peer as a future delegate substitute. Having received a positive quorum means that d_i has received t valid partial signatures for the new modified DGIO. Therefore, d_i can reconstruct the whole signature $sig_{DGIO}^{MKey_p^{-1}}$, as shown in Section 4.1. $sig_{DGIO}^{MKey_p^{-1}}$ is stored in the DGIO and the

signed new DGIO is transmitted to the delegate group members. Before it is stored by d_i in the privilege store, the signature recovery process (Section 4.1) is started, to recover share s_j for the new delegate d_{n+1}. As the number of delegates needed to recover a lost share is t, we need at least $t+1$ good delegates to recover a share of a lost good delegate.

A last note on security: In our solution the shares and the share-shares do not change. This means that the share s_j owned by the disappearing d_j is still valid. If d_j joins the network again, it needs only $t-1$ partial signatures $sig_m^{s_i}$, where $i \neq j$ to get a valid signature $sig_m^{MKey_p^{-1}}$. Consequently, frequent delegation group changes weaken the security of the threshold decryption and the secure sharing schema. Proactive secret sharing [25] avoids this by changing the shares and share-shares frequently. For our purpose, it is sufficient when the data owner reinitializes the secret sharing by generating new shares and share-shares, depending on its personal security requirements. Generating new shares changes only the DGIO and the delegates. There is no need to re-encrypt the object data, as the $MKey_p^{-1}$ and thus the $OKeys$ remain the same.

4.4 Key Management

As explained in Section 4.2, the data owner generates a public master key pair $[MKey, MKey^{-1}]$ for each delegation group. After initialization through the data owner, the delegation group is responsible for $OKey$ management.

$OKeys$ are not stored by the delegates or the DGIO, rather, they are generated for every key request by partially signing the concatenation of object ID and $OKey$ version number $(ID_{O_P} + V_{OKey_{O_P}})$ stored in the DGIO. In that way only the authorized requesting client can generate the $OKey$ and cache it for future requests. Caching the keys reduces network traffic and processing load for the delegates. Because $OKeys$ are newly generated for every key request, a new $OKey_{o_p}$ is implicitly generated when $V_{OKey_{O_P}}$ changes.

4.5 Data Encryption

For data encryption in PACS three cases have to be distinguished: (1) initial data encryption on startup, when a new object is inserted into the network with access control enabled, (2) data re-encryption as a result of a privilege revocation and (3) modification of a stored data object.

Startup. Let us assume peer p wants to insert a new, protected object o into the network. At the very beginning, p has to create a delegate group, if no delegate group exists or the existing delegate group is not appropriate. To do this, p creates the master key pair $[MKey_p, MKey_p^{-1}]$.

Next, p generates the private object key $OKey_{o_p}$ for object o, by signing the ID of o (ID_{o_p}) together with a version number. As this is the start of the encryption, $V_{OKey_{o_p}} = 0$. The initial object key is therefore $OKey_{o_p} = sig_{(ID_{o_p}+0)}^{MKey_p^{-1}}$.

Then p encrypts o with $OKey_{o_p}$, generating the ciphertext object o_p^{-1} as $o_p^{-1} = enc_{o_p}^{OKey_{o_p}}$. p adds the header information $head_{o_p}$, as described in Section 3.3, to the encrypted object. Then p calculates the self-verifying identifier of o_p as $ID_{o_p}^{SV} \equiv h(head_{o_p} + o_p^{-1})$. Afterwards, p distributes $o_p^{-1} + head_{o_p}$ to the replica servers by calling $put(ID_{o_p}^{SV}, (o_p^{-1} + head_{o_p}))$ on the data storage.

What remains to be done is to build the delegate group, so that authorized peers can receive a decryption key in the absence of p. p selects n peers to do the privilege enforcement collaboratively. Next, the shares and share-shares are created and distributed to the delegates, following the procedure presented in Section 4.1. If p had to create a new delegate group, it must generate the DGIO for this new delegate group (see Section 4.3). Otherwise, the object information for o has to be added to the existing DGIO. This includes the $V_{OKey_{o_p}}$, and $enc_{ID_{o_p}^{SV}}^{OKey_{o_p}}$.

After storing the updated or created DGIO in the privilege store, peers receive the $OKey_{o_p}$ on request, if enough delegates approve the request according to the peers privileges. With the $OKey_{o_p}$, the client decrypts the $ID_{o_p}^{SV}$, then requests the data object o_p^{-1} with this ID and decrypts it.

Privilege Revocation. Data re-encryption is needed whenever a privilege on an object o is revoked by a client. A peer that possesses the privilege to grant privileges for object o to other peers has intrinsic read and write privileges on o. This is a constraint of our privilege management, because the re-encryption of an object is in fact a write operation. Before the actual revocation, the revoker requests the DGIO of object o, decrypts the $ID_{o_p}^{SV}$ with $OKey_{o_p}$ and calls $get(ID_{o_p}^{SV})$ on the data store. After receiving o_p^{-1}, r decrypts it with $OKey_{o_p}$ to get the plaintext object o. The $OKey_{o_p}$ can be requested from the delegates because the grant privilege includes read and write privileges.

Next, the revoker r revokes the privilege of peer b on object o. This operation has two parts. The first part is done by the PACS privilege management and is blanked out here. We assume that r has the appropriate privileges and therefore the removal succeeds. The second part is the generation of a new $OKey_{o_p}$ (called $OKey'_{o_p}$), followed by the encryption of o with the new $OKey'_{o_p}$. This is mandatory, since peer b may have seen and cached the $OKey_{o_p}$. Without a new $OKey'_{o_p}$ and re-encryption of o, b can still decrypt o_p^{-1} and get o in plaintext. The new $OKey'_{o_p}$ is generated by the delegates by incrementing the version counter $V_{OKey_{o_p}}$. To allow the delegates to prove the correctness of the revocation, the revoke command is first sent to the delegates for object o. Each delegate d_i that accepts the revoke returns a partial signature $sig_{(ID_{o_p}+(V_{OKey_{o_p}}+1))}^{s_i}$ privately to r. After r has received t valid partial signatures, it computes the new $OKey'_{o_p}$ as $OKey'_{o_p} = sig_{(ID_{o_p}+(V_{OKey_{o_p}}+1))}^{MKey_p^{-1}}$, following the description in Section 4.1. Only then, r submits the revoke command to the privilege store. Afterwards, the revoker r re-encrypts the data object o with the new $OKey'_{o_p}$,

$o_p'^{-1} = enc_{o_p}^{OKey'_{o_p}}$, and stores the newly encrypted object into the data storage. To do this, r computes the new self-verifying identifier $ID'^{SV}_{o_p}$ and executes $put(ID'^{SV}_{o_p}, (o_p'^{-1} + head_{o_p}))$.

Having stored the encrypted data object, the revoker informs the delegates about the update by sending the new encrypted self-verifying identifier of o ($enc_{ID'^{SV}_{o_p}}^{OKey'_{o_p}}$) to the delegates. Now the DGIO can be updated to contain the new version number $V_{OKey_{o_p}} + 1$ and the new encrypted $ID'^{SV}_{o_p}$.

The DGIO update is done either by one of the delegates or by the revoker. To make the DGIO valid again, it needs to be signed with the $MKey_p^{-1}$ key. Again, our threshold signature schema is used to generate the signature. All delegates d_i generate a partial signature of the new DGIO and send it to the coordinator (revoker r, or one of the delegates). If t of n delegates agree with the changes, the signature is complete and the DGIO can be stored in the privilege store with the valid signature.

After the re-encrypted object $o_p'^{-1}$ and the updated DGIO are stored, the outdated object o can be deleted by the revoker. The revoker sends a special *put* request $put(ID^{SV}_{o_p}, DGIO)$ to the data store, where $ID^{SV}_{o_p}$ is the self-verifying ID of the old object and DGIO is the new DGIO object. As explained in Section 3.3, this instructs the storage nodes to delete the object with ID $ID^{SV}_{o_p}$, as it is outdated.

Modifying an Object. Storing a modified object is similar to revoking a privilege. The difference is the interaction with the privilege store. When a peer p wants to update object o, it modifies the object and encrypts it with the $OKey_{o_p}$ (provided that p has already received o). Now p generates the new $ID'^{SV}_{o_p}$ and encrypts it with the $OKey_{o_p}$. This encrypted $ID'^{SV}_{o_p}$ is now sent to the delegates to request an update of the DGIO entry for object o to the new $ID'^{SV}_{o_p}$. The delegates only accept the changes to the DGIO if p has the required privileges to modify object o. If t delegates accept the changes to the DGIO and send back a partial signature of the new DGIO to p, a valid DGIO signature is created. The modified data object can now be stored in the data store using $put(ID'^{SV}_{o_p}, (o_p^{-1} + head_{o_p}))$. After the updated and signed DGIO is put into the privilege store, the old version of o can be deleted by executing $put(ID^{SV}_{o_p}, DGIO)$.

5 Complexity Analysis

Here, we evaluate the performance of PACS with client-side enforcement. We compare PACS with the pure client-side enforcement abbreviated as "client" and pure delegate enforcement shortened as "delegate", as introduced in Section 2. For the sake of brevity, we only compare the simple approaches and do not take into account any optimizations. We also abstract from the details of data

encryption (Section 4.5) and delegation group management (Section 4.3) to get a balanced and comprehensible comparison.

We estimate the number of messages needed, the amount of data that has to be transferred and the computational power needed by the participants to perform the enforcement. The effort for privilege management is not cosidered here. For simplicity, we assume that all data is stored in a DHT with the number of messages used to retrieve data being $log(n)$ where n is the network size. t is the number of delegates that need to be contacted, o is the data object and sig a signature. We abstract here from the overhead of secure communication between the participants as long as it is the same for all approaches. In addition, we assume that the DGIO has the same size as the key file (KF) in the client case. Similarly, the effort to create or modify a DGIO or a KF are the same. We distinguish between decryption (dec), encryption (enc) and signature generation (sig).

We compare the three approaches for an access, an insert, an update, a revoke and a grant request. We distinguish between the first (cold) request and the following requests (hot), as client and PACS enforcement use key caching. The results shown in Tables 1, 2 and 3 and Figures 3 and 4 are explained in the following.

	Cold Request	Hot Request	Insert	Revoke	Grant
Client	$2 * log(n)$	$log(n)$	$2 * log(n)$	$4 * log(n)$	$log(n)$
Delegate	$(t + 1) * log(n) + 2t$	$t * log(n) + 2t$	$2 * log(n)$	0	0
PACS	$2 * log(n) + 2t$	$log(n)$	$2 * log(n)$	$4 * log(n) + 2t$	0

Table 1. Number of Messages for the Different Enforcement Approaches

	Cold Request	Hot Request	Insert	Revoke	Grant
Client	$KF + o$	o	$KF + o$	$2KF + 2o$	KF
Delegate	$DGIO + 2t * o$	$2t * o$	$DGIO + o$	0	0
PACS	$DGIO + t * sig + o$	o	$DGIO + o$	$2 * (DGIO + t * sig + o)$	0

Table 2. Amount of Data to be Transferred in the Different Enforcement Approaches

	Cold Request	Hot Request	Insert	Revoke	Grant
Client	$2 * dec$	dec	enc	$2 * dec + enc$	enc
Delegate	$t * enc + 2t * dec$	$t * enc + 2t * dec$	enc	0	0
PACS	$t*sig+t*enc+(t+1)*dec$	dec	enc	$2t * sig + (2t + 1) * enc + (2t + 1) * dec$	0

Table 3. Computational Effort for the Different Enforcement Approaches

Cold Request. In the client enforcement approach, a DHT request retrieves the key and another DHT request retrieves the data object. DGIO and the object need a decrypt operation. In PACS we need to retrieve the DGIO. Then at least t delegates are contacted to create and return the partial signatures. The retrieval of the data object results in an additional DHT request. In the delegation approach we retrieve a kind of DGIO identifying the delegates for the object. Afterwards, at least t delegates are asked to return partial decrypted data objects securely (i.e. encrypted).

Hot Request. Here we consider only the second request by a client for an object. If the privileges have not changed (if no revoke occurred), the keys cached by the client and PACS can be used again.

Insert. An insert of a new object into an established access control mechanism triggers an update or creation of a KF/DGIO (DHT request). In addition the data has to be encrypted and stored.

Update. In all approaches the altered object is encrypted and afterwards inserted into the network, which causes $log(n)$ messages. As there is no difference between the approaches, this is omitted.

Revoke. In the PACS and client approach, privilege revocation results in requesting and re-encrypting the object. In the client approach, this corresponds to the request effort plus the insert effort. In PACS (if the revoker is not the data owner) t delegates need to be asked for a new key. This results in additional $2t$ messages. In the delegate approach there is no effort, as no one (beside the data owner) has the decryption key in plain text, so no re-encryption is needed.

Grant Granting a privilege causes an update of the key file in the client approach, which results in $log(n)$ messages. The PACS and delegate approach require no effort.

Figure 3 illustrates the predicted number of messages for various approaches in networks with up to 10,000 nodes. PACS is clearly superior to the delegate approach and when the cache is warm, it should perform as well as the client approach. Figure 4 shows the expected sizes of transferred data in kB. The volume load in the network in PACS is much lower than in the delegate method, and similar to the client enforcement method.

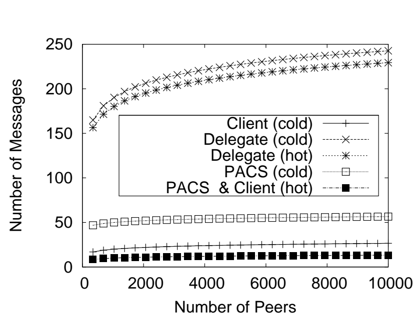

Fig. 3. Number of Messages for a Request with $t = 15$ and $log_2(n)$

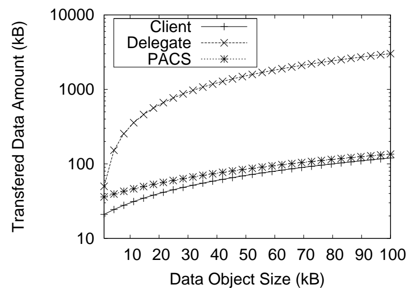

Fig. 4. Amount of Transferred Data for a Cold Request (sig = 1 kB, DGIO = 20 kB, $t = 15$, $log_2(n)$)

6 Discussion and Future Work

Client-side enforcement in PACS is a compromise between performance and power of the access control system. As can be seen from the complexity estimations (Figures 3 and 4), the pure delegate approach is too expensive in the amount of transferred data and communication cost. Especially the fact that each object has to be tranmitted t times is responsible for the limited scalability, as the communication costs linearly increase with the number of requests.

The best performance is achieved by client-side enforcement. However, this has limited access control power. In fact, the pure client-side enforcement shown here can only restrict read access to data objects and offers no administrative distribution, grouping of privileges, roles, etc. as provided in PACS. PACS pays for its additional access control power with every cold request but it performs better than the pure delegation approach in terms of communication costs and the size of data that is transmitted. PACS needs as much computational power as the pure delegate approach, but this is not that critical, as partial signature generation is done in parallel and every peer generates only one signature per request.

All the cryptographic primitives used in PACS have been proven secure elsewhere. As the combination of methods used in PACS does not release or utilize any additional information, also the combination of the cryptographic methods is secure. In such a dynamic environment one has to find tradeoffs between security and performance, and between the power of the access control and its generated overhead. In that way PACS is a compromise in both respects. It guarantees the security of data while producing acceptable additional effort. Furthermore, we get a powerful access control enforcement mechanism with minor restrictions, but with an acceptable overhead.

In the future, we plan to make several extentions to our work. First of all, PACS can be employed on an unreliable P2P network. For non-crucial data, especially data that can disappear for a certain amount of time, this may be less expensive and less complicated. Alternatively, we could dispense with the privilege store. The task and responsibility of storing the privileges reliably in the network can be handed over to the delegates that have to enforce the privileges anyway. In this way, PACS would become more lightweight, which would lead to an easier integration into existing applications based on P2P networks. Furthermore, we want to implement this extention in our PACS prototype and run comparative performance studies.

7 Conclusions

We presented an extention of PACS that enables the replication of confidential data and distribution of privilege enforcement. Our privilege enforcement mechanism is based on threshold encryption / signature and private key encryption with key distribution. The chosen combination enables PACS to enforce powerful access control models like RBAC with administrative distribution, which is impossible with the currently known approaches. Regarding the gained flexibility, the additional overhead of PACS is acceptable. With PACS, data in a P2P network can be protected almost as flexibly as in a centralized scenario. This may open up P2P networks for further application areas.

References

1. Halevy, A.Y., et al.: Schema Mediation in Peer Data Management Systems. In: ICDE. (2003) 505–516

2. Sturm, C.: Orchestrating Access Control in Peer Data Management Systems. In: EDBT Workshops. (2006) 66–74
3. Sturm, C., Dittrich, K., Ziegler, P.: An Access Control Mechanism for P2P Collaborations. In: DAMAP. (2008)
4. Miklau, G., Suciu, D.: Controlling Access to Published Data Using Cryptography. In: VLDB. (2003) 898–909
5. Bouganim, L., et al.: Client-Based Access Control Management for XML Documents. In: VLDB. (2004) 84–95
6. Desmedt, Y.G.: Threshold Cryptography. European Transactions on Telecommunications and Related Technologies **5**(4) (1994) 449–457
7. NIST: Role Based Access Control. ANSI INCITS 359-2004 (February 2004)
8. Crispo, B., et al.: P-Hera: Scalable Fine-grained Access Control for P2P Infrastructures. In: ICPADS. (2005) 585–591
9. Berket, K., Essiari, A., Muratas, A.: PKI-Based Security for Peer-to-Peer Information Sharing. In: P2P. (2004) 45–52
10. Goh, E.J., et al.: SiRiUS: Securing Remote Untrusted Storage. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2003. (2003)
11. Koç, E.: Access Control in Peer-to-Peer Storage Systems. Master's thesis, ETH Zurich (October 2006)
12. Koç, E., et al.: PACISSO: P2P Access Control Incorporating Scalability and Self-Organization for Storage Systems. Technical Report TR-2007-165, Sun Microsystems, Inc. (June 2007)
13. Sandhu, R., Zhang, X.: Peer-to-Peer Access Control Architecture Using Trusted Computing Technology. In: SACMAT. (2005) 147–158
14. Saxena, N., et al.: Threshold Cryptography in P2P and MANETs: The Case of Access Control. Comput. Netw. **51**(12) (2007) 3632–3649
15. Naor, M., Wool, A.: Access Control and Signatures via Quorum Secret Sharing. IEEE Trans. Parallel Distrib. Syst. **9**(9) (1998) 909–922
16. Castro, M., et al.: Secure Routing for Structured Peer-to-Peer Overlay Networks. SIGOPS Oper. Syst. Rev. **36**(SI) (2002) 299–314
17. Committee, O.A.C.T.: eXtensible Access Control Markup Language (XACML) Version 2.0 (2005) `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.
18. Stoica, I., et al.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: SIGCOMM 2001. (2001) 149–160
19. Desmedt, Y.G., Frankel, Y.: Threshold Cryptosystems. In: CRYPTO '89, New York, NY, USA, Springer-Verlag New York, Inc. (1989) 307–315
20. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)
21. Rivest, R.L., et al.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Commun. ACM **21**(2) (1978) 120–126
22. Shamir, A.: How to Share a Secret. Commun. ACM **22**(11) (1979) 612–613
23. Desmedt, Y., Frankel, Y.: Shared Generation of Authenticators and Signatures. In: CRYPTO '91, London, UK, Springer-Verlag (1992) 457–469
24. Gennaro, R., et al.: Robust and Efficient Sharing of RSA Functions. Journal of Cryptology **13**(2) (2000) 273–300
25. Herzberg, A., et al.: Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In: CRYPTO' 95. (1995) 339–352