Personal Health Record System Protected using CP-ABE

Sebastian Banescu Hugo Ideler Simona Posea

Abstract

This paper describes the design and implementation of a distributed system used for storing and retrieving personal health records. The central repository is protected by the ciphertext-policy attribute-based encryption scheme. The implementation was done using Java and the jPBC [4] library for pairing-based cryptography.

1 Introduction

This section describes our choice of access control model. It presents arguments as to why the CP-ABE was chosen as the most suitable solution for storing health data in a distributed environment.

The Public Key Infrastructure (PKI) has the major disadvantage that one has to obtain the public key of the recipient before being able to encrypt a message for the respective user. This also implies that a user must have a public key from, or at least certified by a Third Trusted Party (TTP). In a Public Health Record (PHR) management system this would be difficult to achieve. For example, just imagine a patient wanting to share her data with all the doctors in some country, in order to allow for proper intervention in case of an accident. Considering such a scenario, one would have to get the public keys of all the doctors and encrypt the medical data with each of these keys. This would result in an significant processing overhead on the encryptor's part, very difficult or even impossible to perform considering the large number of both patients and doctors.

A more flexible solution proposed in literature is the Identity-Based Encryption (IBE). Compared to PKI, IBE is based on using known strings that uniquely identify a user in a group as public keys. The email or even the user's complete name (in the case of a small group) can be used as public keys in an IBE scheme. However, this does not entirely solve the problem mentioned before. A patient will still have to obtain a list with the unique identifier (to be used as public key) of all the doctors that should be able to decrypt the medical data. Even if this may seem easier than when using PKI, the number of encryptions that need to be performed remains the same.

The solution to this problem is provided by the Attribute-Based Encryption (ABE), which uses attributes in order to identify users. More specifically, a private key is associated with a set of attributes ω and can be used to decrypt only those ciphertexts encrypted with a public key, which is associated with the set of attributes $\omega' \subset \omega$.

There are several Attribute-Based encryption schemes proposed in literature, including the BSW scheme introduced by Bethencourt et al. in [2], the CN scheme of Cheung and Newport in [3] and the CP-ABE scheme proposed by Ibraimi et al. in [6]. The first two schemes present a set of disadvantages (e.g. BSW requires many pairings and exponentiation operations due to the large number of polynomial interpolation required to reconstruct the secret, the size of ciphertext and secret key in CN increases with the number of attributes in the system, etc.) which makes them not appropriate for a PHR system. The third scheme ([6]) was proved by the authors to

be efficient and secure, fact that motivates our choice for implementing it as a basis for the PHR system access control.

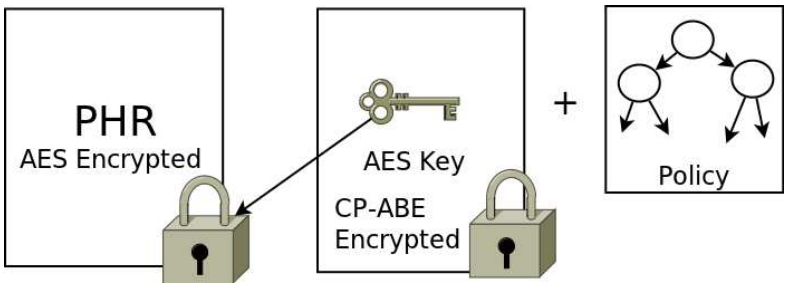

Figure 1: PHR Encryption Approach

However, our approach is not completely the same with the one in [6]. In order to allow for documents having variable length to be encrypted, we first encrypt the documents using AES in CBC mode, with PKCS#5 padding and after use the CP-ABE scheme to encrypt the AES symmetric key. This idea is graphically depicted in Figure 1. It can be noticed that the scheme does not support revocation. However, this can be easily achieved by slightly modifying the implementation and using a mediator as described in [5].

2 Theoretical Background

The scheme described in [6] includes 4 phases, two of them being performed by the Trusted Authority:

- **Setup** $\rightarrow$ (pk, mk): generates a public key pk and a master secret key mk to be used in the other three phases of the scheme;
- **KeyGen** (ω, mk) $\rightarrow$ (sk_ω): uses the master key previously built to generate a secret key sk_ω for a user having the set of attributes ω;

and the other two by the users:

- **Encrypt** (m, pk, τ) $\rightarrow c_\tau$: uses the public key pk generated by the Trusted Authority to encrypt a message m such that only users possessing specific attributes that satisfy the attribute-based access policy τ will be able to decrypt it;c_τ represents here the resulting ciphertext that also includes the access policy;
- **Decrypt** $(c_\tau, sk_\omega) \rightarrow m$: decrypts a ciphertext c_τ using the secret key sk_ω received from the Trusted Authority.

In order to enhance the understanding of the scheme, a detailed description of each of the 4 phases is provided below:

2.1 Setup

The Setup phase is the first phase of the scheme and must be performed by the Trusted Authority. It includes the following steps:

- Generate a bilinear group $\mathbb{G}_0$ having prime order p and generator g;

- Build a symmetric bilinear map $\hat{e} : \mathbb{G}_0 \times \mathbb{G}_0 \rightarrow \mathbb{G}_1$;
- Define the total set of attributes $\Omega = \{a_1, a_2, ..., a_n\}$, which can be used to create access policies; the implementation for the PHR system defines attributes like: patient, doctor, pharmacy, insurance, health club, hospital, etc. However, this set of attributes can be easily extended;
- For each attribute $a_i \in \Omega$ generate a random element $t_i \in \mathbb{Z}_p$; additionally, generate the random parameter $\alpha \in \mathbb{Z}_p$;
- Compute $y = \hat{e}(g, g)^{\alpha}$, and $T_i = g^{t_i}$, $1 \leq i \leq n$ and $n = |\Omega|$.

Public key pk and master key mk become:

$$pk = \{e, g, y, T_{i,1 \leq i \leq n}\},$$
$$mk = \{\alpha, t_{i,1 \leq i \leq n}\}.$$

Once pk is known, the key must be published such that everyone can use it in order to encrypt messages. The master key mk will remain known only by the Trusted Authority.

In the current implementation, for the the construction of the bilinear map $\hat{e}$ the jPBC library introduced in [4] has been used. jPBC is a Java porting of the PBC library written in C [1, 7]. It supports 6 different types of elliptic curves (A, A1, B, C, D, E and F), as introduced by Lynn in [7]. For the implementation of the Access Control scheme a curve of type A has been used, due to its simplicity and applicability. The curves of type A are supersingular curves having the following form:

$$y^3 = x^3 + x,$$

being constructed over the field $\mathbb{F}_q$, for some prime $q = 3 \mod 4$. Thus, a description of the other types of curves is not provided here. However, we recommend the reader to refer to [4] for a better understanding of these curve type classification.

2.2 KeyGen

The Key Generation phase is performed by the Trusted Authority only when requested by a user. The algorithm includes the following steps:

- Generate a random element $r \in \mathbb{Z}_p^*$; this element is introduced in order to protect against collusion attacks;
- Compute the following set of elements: $d_0 = g^{\alpha - r}$ and $d_j = g^{r t_j^{-1}}$, for every attribute a_j given by the user;

The set of elements in d represents the secret key sk_{ω} to be returned to the user. Note that before generating the secret key for a user, the Trusted Authority must check the attributes possessed by the respective client.

2.3 Encrypt

In order to encrypt a message $m \in \mathbb{G}_1$ using the access policy τ, a user must follow the steps below:

- Select a random element $s \in \mathbb{Z}_p^*$ and set $c_0 = g^s$, $c_1 = my^s$;

- Assign a value s_i to each of the leafs nodes in the policy tree, such that the sum modulo p of the assigned values for the attributes in an AND set equals s. An example of performing secret sharing of s is shown in Figure 2;
- For each leaf attribute $a_{j,i} \in \tau$ compute $c_{j,i} = T_j^{s_i}$, where i represents the index of the attribute in the policy tree;

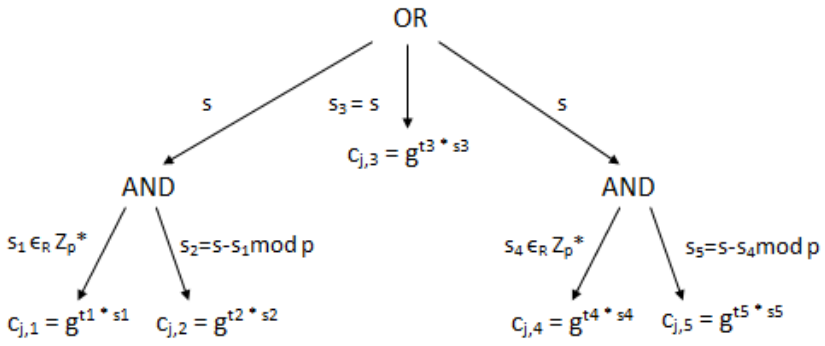

Figure 2: Assigning secret shares s_i to an access tree.

Once all the three steps have been performed, the ciphertext becomes:

$$c_\tau = (\tau, c_0, c_1, c_{j,i}, \text{ s.t. } a_{j,i} \in \tau).$$

The authors in [6] describe the access policy as a tree having attributes as leafs and one of the following operators *AND, OR, X OUT OF Y* as non-leaf nodes. An example of such policy is displayed in Figure 3.

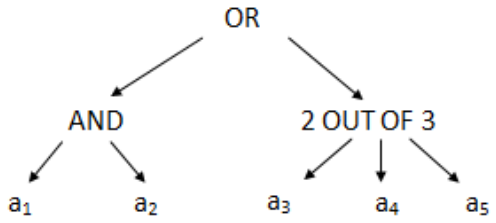

Figure 3: An example of Access Policy as used in [6]

However, for simplicity, in the implementation of the current process we express the policy using only AND and OR operators and having the following format:

$$\tau = (ANDSet_1) \text{ OR } (ANDSet_2) \text{ .....OR } ANDSet_m,$$

where:

$$ANDSet_i = a_{i1} \text{ AND } a_{i2} \text{ AND ....AND } a_{ij},$$
$$1 \leq j \leq n,\ a_{il} \in \Omega,\ 1 \leq l \leq j.$$

It can be easily noticed that the format adopted in the current implementation is equivalent with the format proposed in [6]. For example, the policy in Figure 3 can be rewritten as a policy with only AND and OR as operators, as shown in Figure 4. Although the policy expression may become larger due to the expansions required to transform the operators $OUT\ OF$ and the multi-level tree into simple 3-levels tree with AND and OR as only operators, this format allows for faster computations and easier implementation, being preferred over the initial one.

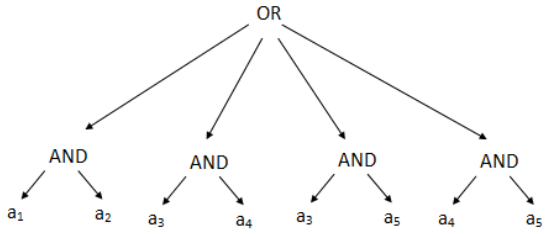

Figure 4: An example of Access Policy as used in the PHR system implementation.

2.4 Decrypt

Before being able to decrypt a message, a user must extract the minimum set of possessed attributes that satisfy the access control attribute-based policy associated with the ciphertext. This can be done either manually by the user or automatically, without requiring extensive computations. Let ω' be the minimum set of attributes necessary to decrypt a message. For example, supposing that a user possesses the following set of attributes:

$$\omega = \{a_3, a_5, a_6\}$$

and the access policy is:

$$\tau = (a_3) \text{ OR } (a_5 \text{ AND } a_6 \text{ AND } a_7)$$

then the minimum set of attributes required is:

$$\omega' = a_3.$$

The algorithm continues by decrypting the ciphertext c_τ using the formula below:

$$m' = \frac{c_1}{\hat{e}(c_0,d_0)\prod_{a_j \in \omega'} \hat{e}(c_{j,i},d_j)}.$$

We recommend the reader to refer to [6] for a proof of this formula.

Only if the set of attributes selected by the user satisfies indeed the access control policy associated with the ciphertext, will the quality

$$m' = m$$

be obtained.

3 Data Model

This section presents the abstract data model that our system is based on. The description includes the format of the data and how it is used to communicate between entities in the system.

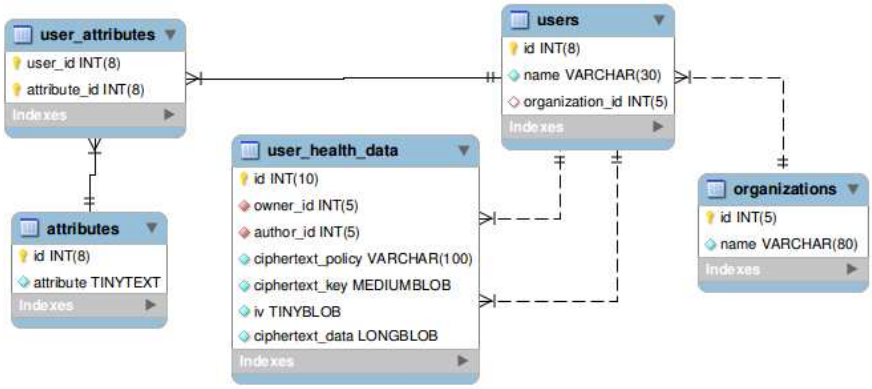

Figure 5: Database Schema Structure

3.1 Personal Health Records

The Personal Health Records (PHRs), which are the main data items that our system protects, are stored in a public database in encrypted form. PHRs are composed of several data records which are identified by a unique positive integer number. All health data records have an author, which is the user who added the record to the database (e.g. doctors, trainers, health club employees, etc.). However, record also has an owner represented by the person who the data refers to (e.g. patients). The actual contents of the health record are encrypted and stored in the form of a blob, i.e. a sequence of bytes.

The set of all data records that have the same owner make up that persons PHR. This implies a very flexible format for the data records that the system may store. Basically any type of file can be safely stored using our model. Therefore, the system allows hospitals, health clubs and other organizations to add the format that they prefer, without imposing the constraint of having the same format for all health data that is added. From our point of view this is more realistic and practical than the other mentioned alternative.

Another advantage of our approach is that the user can freely split sensitive information in any way s/he chooses fit and encrypt them with the desired policies. For instance emergency information should only be decrypt-able by a guardian, i.e. a trusted person designated by the owner. The user may simply communicate his preferred encryption policies to each organization s/he interacts with. Consider the case where Alice is registered at the *Lincoln Memorial Hospital* and she also goes to physical therapy sessions at the *Malibu Health Club*. She will most probably inform her trainer at the health club that she wants doctors from the recovery section of the *Lincoln Memorial Hospital* to see her health data after training sessions. Therefore, her trainer will encrypt her data records using the corresponding policy stated by Alice.

To allow such a high degree of flexibility we use the database schema illustrated in Figure 5 implemented using MySQL Server 5.1. We have not focused our time towards designing very elaborate user and organization structures and relations since this was not the focus of our project. Hence, tables representing users and organizations are rudimentary and contain only a name and a foreign key link. It can be observed from Figure 5 that the health data records contain:

- The unique record identification number.
- The IDs of both the owner and the author which are foreign keys referencing the *users*

table.

- The policy which was used to encrypt the AES key, in plain text.
- The CP-ABE encrypted AES key.
- The initialization vector (IV) used by AES in cipher block chaining (CBC) mode.
- The AES encrypted health data record.

Note that the attributes list containing textual identifier of the attributes are stored in a text file to avoid unnecessary database interactions during the setup phase. It important to remark that this attribute list is included in the database as well, this does not change the operation of our system in any significant way.

3.2 Keys

All keys generated by the *Trusted Authority* (TA) are stored files having binary format. When the TA is stared up, it checks if it can find the master key and the public key files. In case it doesn't find them it goes through the setup phase where it generates both of these keys, otherwise it reconstructs the key structures from the binary files.

The α part of the master key is stored first, followed by the each of the random numbers $t_i \in_R \mathbb{Z}_p, 1 \leq i \leq n$ (in ascending order of their index), that are associated with each attribute from the list provided in the attributes text file. The file containing the master key is generated by the TA in a secure environment and it is not disclosed to any other party.

The public key is also generated by the TA. The first element stored in the output file is the generator $g \in \mathbb{G}_0$, followed by $y = \hat{e}(g,g)^{\alpha} \in \mathbb{G}_1$ and then the list of $T_i = g^{t_i} \in \mathbb{G}_0, 1 \leq i \leq n$ in ascending order of their index. The public key file can then be published somewhere or disseminated to any party that wishes to have it.

Each time a user requests a secret key it passes a set of attributes to the TA. Before generating the secret key and providing it to the user, the TA should check the validity of these attributes with respect to the requesting user. The way in which this check is done is beyond the scope of this project. Therefore, we will assume that by checking the contents of the *user_attributes* table from the database, the TA will know which attributes are valid for which user. This check is illustrated in Figure 6 is just a rudimentary check and it can be skipped like in the implementation of BSW [2], however we add it just as a proof of concept. Therefore, if the user requests a key for more attributes than s/he is entitled to, the TA will reply with an error message.

The secret key files which are sent each only to the requesting user are constructed by first storing the $d_0 \in \mathbb{G}_0$ and then the other d_is corresponding to the attributes $a_i \in \omega$ of the user. This file should be safely stored by the user and not shared with any other party.

4 Software Design

The architecture of our system is illustrated in Figure 6. Users have a generic implementation and can be instantiated by anyone having any role (e.g. patient, doctor, trainer, etc). The secret key request and response is implemented using a client-server architecture and the communication is realized though sockets, i.e. IP address and port number. Our future work includes authenticating this communication channel and making it secure from the point of view of confidentiality and integrity. However, this is not the focus of the current project.

Any user may use the public key to add encrypted heath records to the public database according to some chosen policy. However, only the user's whose secret key satisfies the policy

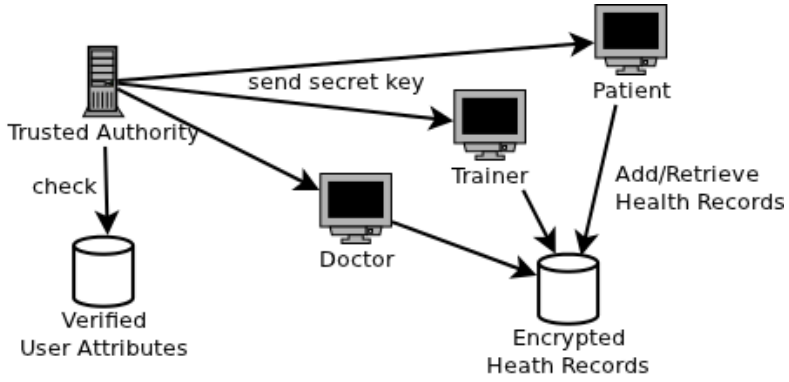

Figure 6: System Architecture

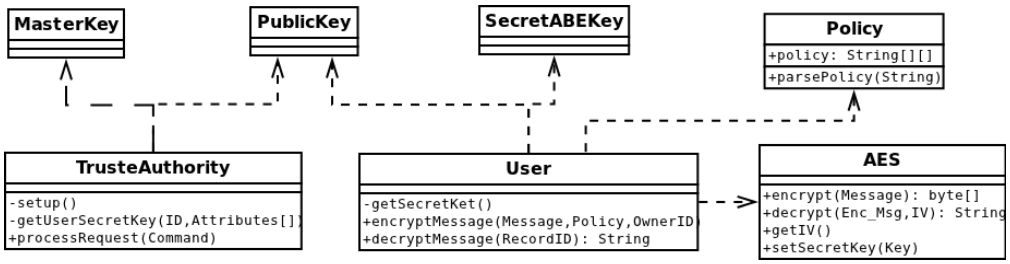

Figure 7: Core UML Diagram

associated to a ciphertext can decrypt the corresponding health record. This solution is perfect for distributed systems where the centralized storage can be accessed by anyone, but it may only be decrypted by certain users that have a corresponding secret key.

The UML diagram from Figure 7 offers a visual model of our software architecture core. The `TrustedAuthority` and the `User` classes are the central parts of the system. The former can perform the setup phase and can generate secret keys for users. The latter can use the secret keys to encrypt and decrypt AES keys, that are used for encrypting and decrypting the actual contents of the health records.

In the encryption process of the AES key using CP-ABE, the user needs an instance of the `Policy` class in order to associate it with the ciphertext. The policy is read by users that wish to decrypt the data in order to determine if this action is possible with the attributes that their secret key includes.

5 User Interface

When starting the application as a user, one will be greeted by the screen in Figure 8. At this point, it is imperative that a connection with the database is possible before proceeding. After clicking on Populate the combo box becomes selectable as visible in Figure 9. From these list of users, the user selects his identity. Of course, as explained earlier, this is for demonstration purposes only – in reality the TA would check and validate the authenticity of the user before handing out secret keys. Once the identity has been selected, a helpful hint is displayed listing

Figure 8: Startup screen

Figure 9: Startup screen: after clicking populate

Figure 10: Startup screen: after selecting a user

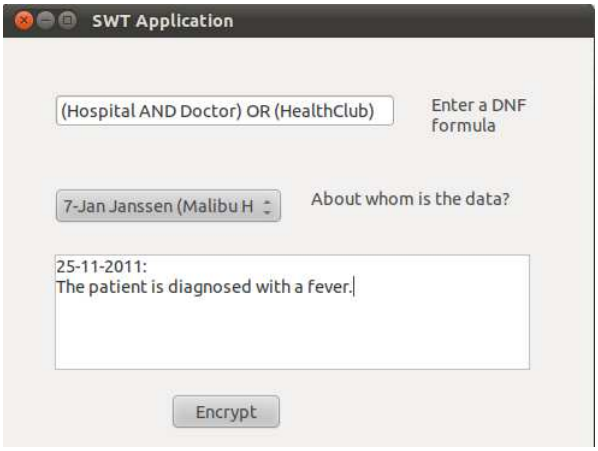

Figure 11: Encrypting a record

the user's attributes (Fig. 10).

Once the user has selected his or her identity, the user can choose to encrypt or decrypt. We will exhibit the former first. In Figure 11 it is displayed which steps are taken. The user chooses a decryption policy (here in Disjunctive Normal Form – but any formula can be rewritten in DNF), chooses about whom this data entry relates (note that this has no cryptographic significance) and finally enteres the text for the record.

Decryption is likewise made simple. As depicted in Figure 12, a table lists all encrypted records in the database. From this table, the user can select the records he or she wants to (attempt) to decrypt. The entries in the table show the number of the record, the subject (about whom the data relates), the author (who created this record) and the access policy. The access policy is shown in text, but this has no cryptographic significance (i.e., changing the text in this field will not allow you to decrypt anything you couldn't already before). As a reminder, there is a label showing the user which attributes he or she has. After clicking on Decrypt, the plaintext is shown in the bottom field.

Figure 13 shows what happens if the user tries to decrypt a record to which the user has insufficient attributes. The user is informed of the inability to decrypt (note that in this screenshot a different user is logged in than in the previous one).

Finally, adding a new user is also possible. In the startup screen (Fig. 8) there is a button New, clicking on this brings up a new window. As visible in Figure 14, the user can choose a name, attributes and the organization for this new user. Again, this feature is merely for demonstration purposes.

6 Conclusions

This paper presented a Java implementation of the CP-ABE scheme used for a distributed PHR storage and retrieval system. Our implementation is based on the paper of Ibraimi et al. [6] and includes two main modifications. The first change refers to encrypting the contents of the PHRs using AES and further encrypting the AES symmetric key using CP-ABE. The second modification consists in using a simpler format for the access policy. This allows for easier implementation and it can be proved that it is equivalent to the format used in [6]. During the

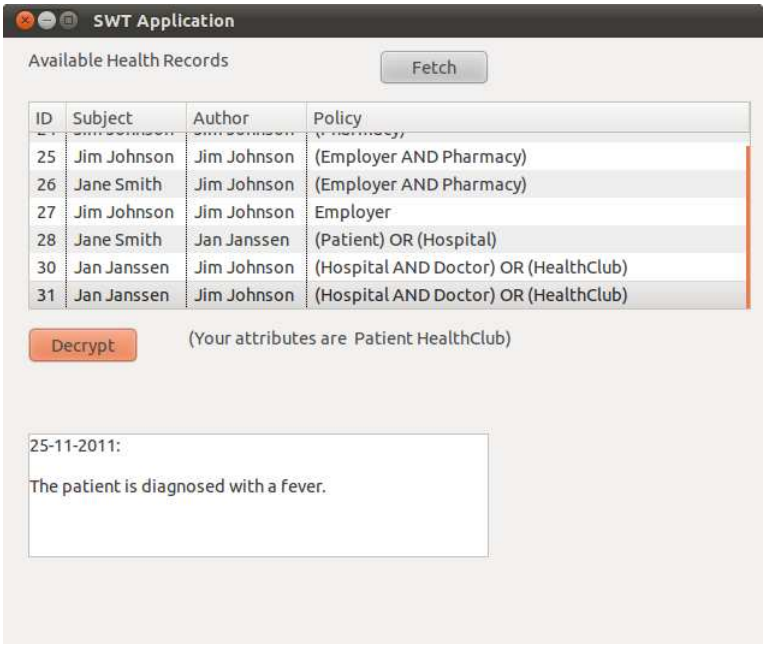

Figure 12: Decrypting a record

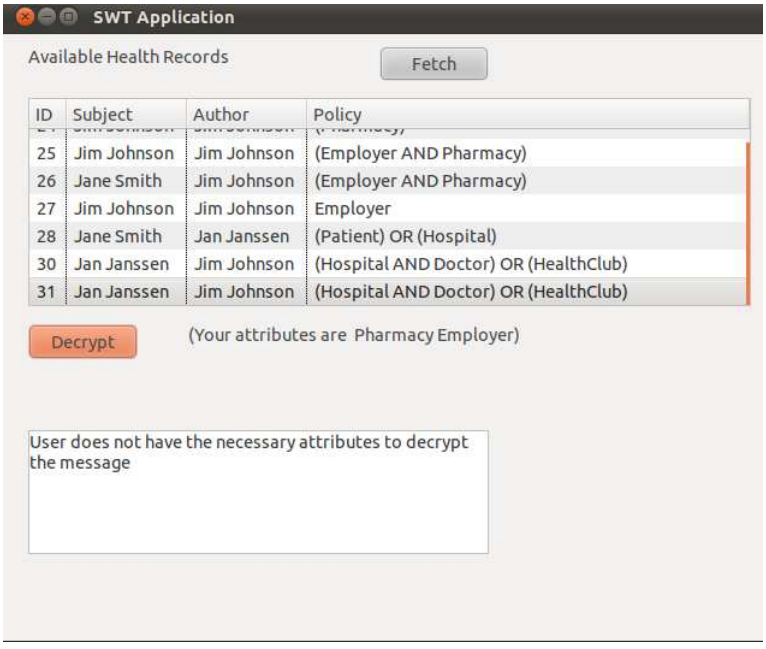

Figure 13: Failure of decrypting a record due to insufficient attributes

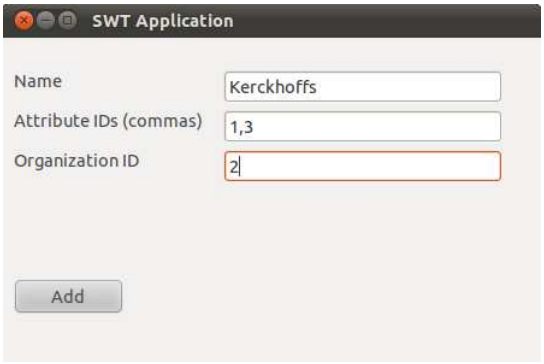

Figure 14: Adding a new user

implementation the major problems encountered were related to the use of the jPBC library for creating bilinear maps.

It can be argued that the scheme implemented in the current project does not support attribute revocation. However, this can be easily extended as proposed in [5]. This would simply imply generating an extra secret key for each user that would be stored by the mediator, and using it to perform partial decryption when requested. The authors of this work feel that this change does not imply significant additional work and could be easily implemented.

References

[1] PBC: Pairing Based Cryptography. http://crypto.stanford.edu/pbc/.

[2] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Ling Cheung and Calvin Newport. Provably Secure Ciphertext Policy ABE. Cryptology ePrint Archive, Report 2007/183, 2007. `http://eprint.iacr.org/`.

[4] A. De Caro and V. Iovino. jPBC: Java pairing based cryptography. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 850 –855, 28 2011-july 1 2011.

[5] Luan Ibraimi, Milan Petkovic, Svetla Nikova, Pieter Hartel, and Willem Jonker. Information Security Applications. chapter Mediated Ciphertext-Policy Attribute-Based Encryption and Its Application, pages 309–323. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] Luan Ibraimi, Qiang Tang, Pieter Hartel, and Willem Jonker. Efficient and Provable Secure Ciphertext-Policy Attribute-Based Encryption Schemes. In *Proceedings of the 5th International Conference on Information Security Practice and Experience*, ISPEC '09, pages 1–12, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] B. Lynn. On the Implementation of Pairing-Based Cryptography, PhD Thesis. http://crypto.stanford.edu/pbc/thesis.pdf.