Secure Data Management Project Report:
Enforcing Cryptographic Access Control on a PHR system

Riccardo Gennaro - s3534219
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Bogdan David - s3433927
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Péter Svelecz - s3542629
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Rik van de Haterd - s2590859
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Our system models a Personal Health Record (PHR) system involving multiple types of parties: patients, doctors, insurance companies, health clubs, employers, and hospitals. In this system: Patients can insert personal health data into their own records and grant read access to parts of their health records to other parties. Hospitals can insert patient health data for any of their patients. Health clubs can insert training data for any of their members. To meet these requirements, we implement a Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) scheme, as multiple authorities—such as hospitals and health clubs—are central to our project. Our system employs a hybrid encryption approach combining Advanced Encryption Standard (AES) and MA-CP-ABE to ensure that only parties with legitimate access can read and write data, and that health data is securely stored. In this setup, patient data is encrypted using AES for secure storage in the database. The AES encryption keys are then encrypted using MA-CP-ABE, which means that a party can decrypt the data only if their attributes satisfy the policy under which the data was encrypted. This ensures that only authorized parties can access the patient data. However, since CP-ABE schemes do not inherently distinguish between read and write access, we implemented a challenge-response protocol. This protocol ensures that only parties with adequate attributes may modify the data stored in the database, effectively controlling write access and maintaining data integrity.

Keywords: Personal Health Record, PHR, Access Control, MA-CP-ABE

1. Functional Requirements

In designing a Personal Health Record (PHR) system, it is crucial to consider the various types of users and their interactions with the system. This allows for the quick identification of functional dependencies and potential security implications, particularly regarding access control and data integrity. In our scenario, we categorize users into three primary groups: **active** users, **passive** users and **authorities**.

1.1 Active users

Active users have the capability to modify data within the system. They can add new entries, edit existing information, and, where appropriate, delete data. In the context of the PHR system, active users include:

- **Patients:** Patients can insert personal health data into their own records. They have control over their health information and can decide which parts of their records, such as health or training data, are accessible to other parties.
- **Doctors:** Doctors can insert patient health data for any patient treated by them. This includes medical records, treatment histories, and diagnostic results relevant to the patient's care.
- **Health club trainers:** Health club trainers can insert training-related health data into the records of patients who are members of the club.

1.2 Passive users

Passive users are limited to reading data from the system. They do not have permission to modify, add, or delete data. Their interaction is confined to viewing information that the patient has granted them access to. The passive users in our scenario are:

- **Insurance Company representatives**: Insurance representatives can be provided with read access to parts of a patient's health record.
- **Employer representatives**: Employer representatives may receive read access to certain health information of the patient if this is permitted by the patient in question.

1.3 Authorities

Authorities are responsible for issuing and delegating certain rights to their representatives. The authorities in our scenario are:

- **Hospitals**: Hospitals govern the rights of the doctors and their respective patients.
- **Health clubs**: Health clubs govern the rights of the health club trainers and their patients.
- **Insurance**: Insurance companies govern the rights of the insurance representatives and patients
- **Employers**: Employers govern the rights of the employer representatives and their employees (patients).

2. Security Requirements

Due to the nature of the data stored in the system, access to it shall be granted only to parties with legitimate interest. In this scenario, the following restrictions must be considered:

1. Access to the health record can only be granted by the record's owner.
2. Passive users can only be granted READ access.
3. Active users can be granted both READ and WRITE access.

To enforce that only the health record's owner can read its data and that no other party can understand it without the proper credentials, the data will be encrypted in such a way that anyone with an appropriate key can read and interact with it. As the access control is enforced purely on a cryptographic level, the decryption key will need to satisfy the following properties:

1. It must refer to only ONE type of data (general record, training record, etc.).
2. It must be unique for every combination of (user, health record).

As cryptographic access control often only grants full read/write control we will have to extend the solution to include authorization of users, to ensure that unauthorized users have access to rights they don't deserve. Furthermore, the database system must be considered a TRUSTED party and will act as the main authority for distributing keys (both encryption and decryption).

3. Software Architecture

3.1 Access Control Scheme

Our project implements a Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) scheme. MA-CP-ABE is a cryptographic scheme that allows users to access encrypted data based on a policy embedded in the ciphertext. In traditional ABE schemes, the access is solely based on the user attributes. In MA-CP-ABE however, the access is also based on a policy that includes attributes governed by multiple independent authorities. Each authority manages specific attributes and delegates access to the encrypted data based on satisfying conditions set in the encryption policy. This distribution of authority is ideal for a PHR system where there are multiple actors and authorities that have varying levels of access to sensitive data. The three main reasons why MA-CP-ABE is a good solution for this system are:

1. **Distributed attribute control:** From the requirements, we gathered that there is a need for multiple authorities such as hospitals, employers and health clubs. These different entities can make attributes that correspond to their representatives, allowing for decentralized control and data governance.
2. **Flexible access policies:** Users can use these unique attributes to define complex but flexible access policies. These policies allow for fine-grained control of who has access to specific data. For example, a patient can restructure the policy on his PHR to allow his insurer to view his medical data, which he was first unable to access.
3. **Privacy and security:**. Using the policies and attributes, the private data is encrypted securely. Furthermore, the patient can be certain that only authorized users can view his data. The goal of the scheme is to enforce secure and strict role-based access whilst maintaining privacy when multiple authorities are involved.

For the implementation of MA-CP-ABE, we make use of the Charm framework in Python. Charm is used for the prototyping of cryptographic schemes and provides a comprehensive library of cryptographic tools and algorithms. A more detailed explanation of the implementation using Charm can be found in Section 4.

3.2 Authentication

MA-CP-ABE inherently only allows for full read/write access, which is not in line with our project requirements. To tackle this, we implemented a challenge-based authentication system. Challenge-based authentication is a security method where a user proves their identity by responding to unique and one-time challenges. One achieves the proof without having the users require a key exchange, ensuring that keys remain private. The authentication works in three steps:

1. **Challenge generation:** The authenticator generates a random challenge and sends it to a user

2. **Response:** The user must respond to the challenge with a correct answer. This can be done by encrypting the challenge with a private key that only the authenticated user would know. The user then sends the answer back to the authenticator.
3. **Verification:** The authenticator checks if the response is correct. If so, the user is authenticated

Our specific implementation of this can be read in Section 4.

3.3 Hybrid-Encryption

To ensure security in an environment where messages and keys are stored in a database, we implement a hybrid encryption scheme by extending the MA-CP-ABE scheme with AES encryption. This creates an encryption approach where the message content is encrypted using AES, which in turn gets encrypted using the MA-CP-ABE key. The ABE-encrypted AES key is prepended to the AES-encrypted message. This way only the authenticated user can decrypt the AES key, with which he can decrypt the message.

3.4 Django

3.4.1 Front- and back-end

Django is a high-level Python web framework that allows for fast and practical development. In this project, Django serves as both the front- and back-end framework.

- **Front-end**: Django's template allowed us to quickly develop user interfaces that interact smoothly with our back-end logic.
- **Back-end**: Through Django, we implement a Model-View-Controller (MVC) architecture that handles our business logic and data manipulation. The models defined in Django represent the structure of the data and are directly linked to the database. The views amount to the front end, and the controller handles the business logic. Using the MVC architecture allows for a clear division of labour and modularity in our project, which improved development efficiency and adherence to good coding practices.

Using Django for both the front- and back-end of the project simplified our development process by maintaining consistency in language and framework. Furthermore, it allowed us to work seamlessly with our database, which is crucial for a PHR system.

3.4.2 Database

For the database of our project, we use an integrated SQLite3 database. SQLite is compatible with the Django framework and allows us to structure our data using Python models. The database is essential for storing and retrieving the parameters, keys and messages in the PHR system.

4. Details on Implementation

In this section, we outline the implementation of the core components and structure of the system's cryptographic and data management architecture. We begin with an overview of the MA-CP-ABE scheme, detailing the attributes and access policies that govern data security. Following this, we describe the challenge-based authentication that validates access rights. Finally, we examine the data model, which encompasses entities such as authorities, representatives and patients along with their interrelationships.

4.1 MA-CP-ABE

From the Charm framework, we implemented the `[abenc_maabe_rw15]` module. This module implements a public key ABE scheme with access policies. The security of the module is based on the complex q-type assumption.

4.1.1 Attributes

The attributes of the module are structured as follows:

`ATTRIBUTE@AUTHORITY_INDEX`

This format consists of the following three components:

1. **Attribute name:** Represents the role assigned to a user, such as `PATIENT` or `DOCTOR`.
2. **Authority name:** Represents the entity that is responsible for issuing and managing the attribute, indicated after the `@` symbol, such as `HOSPITAL` or `HEALTHCLUB`.
3. **Attribute index:** Represents an optional identifier which is appended after an underscore, which is used to distinguish between multiple instances of the same attribute issued by the same authority. This was crucial for our system as it would allow us to identify different users without having to make a special attribute for each one. However, this feature ended up not working forcing us to make specific attributes for different users.

4.1.2 Policies

An access policy is a string that outlines the necessary combination of attributes a user must possess to decrypt encrypted data and is constructed as follows:

`(PATIENT@PHR OR DOCTOR@HOSPITAL1_1) OR INSURANCEREP@INSURANCE2_1`

When encrypting a message, the data owner defines the access policy as a string and passes it to the encryption function. The `encrypt` method in the `MaabeRW15` class takes this policy string as an argument and embeds it within the ciphertext. Only users whose attributes satisfy the embedded policy can successfully decrypt the message.

4.2 Challenge authentication

4.2.1 Representative challenge

The challenge system for the server consists of two main functions: `get_challenge_auth_patient` and `post_challenge_auth_patient`. These functions work together to securely authenticate an authority representative (`rep_id`) when they attempt to access a patient's data.

4.2.2 `get_challenge_auth_patient`

This function generates an encrypted challenge for the authority representative to authenticate the patient. It performs the following steps:

1. **Retrieve PatientReps:** Fetches all `PatientRep` objects associated with the patient identified by `uuid`. Initializes an empty set `auth_set` to store relevant authority identifiers.
2. **Determine access parameters** Based on the `type` parameter, sets the variables:
   - If `type` is `"HEALTH"`:
     - `check_auth` = `"HOSPITAL"`
     - `check_attr` = `"DOCTOR"`
   - If `type` is `"TRAINING"`:
     - `check_auth` = `"HEALTHCLUB"`
     - `check_attr` = `"HEALTHCLUBTRAINER"`
3. **Identify relevant authority:** Iterates over each `PatientRep` and retrieves their attributes and extracts their authority. If `check_auth` is part of the authority identifier, adds it to `auth_set`

4. **Validate authority set:** If `auth_set` is empty, returns an error response indicating that the patient has no associated authorities of the required type.
5. **Generate challenge**
   (a) Generate a random group element `challenge` using `ma_abe.helper.get_random_group_element()`.
   (b) Construct the access policy string `policy` using `auth_set` and `check_attr`
   (c) Encrypts the `challenge` under the constructed policy using `ma_abe.helper.encrypt(challenge, policy)`
   (d) Serialize and encode the challenge to base64.
   (e) Store the tuple `(challenge, uuid, type)` in a global dictionary `challenge_map` with `rep_id` as the key.
   (f) Send the challenge as a JSON response to the challenged party

4.2.3 `post_challenge_auth_patient`

This function validates the response to the challenge provided by the authority representative. It follows these steps:

1. **Parse request data** Parse the JSON response and check for the presence of `b64_serial_challenge` and `b64_serial_rep_message`.
2. **Decode and deserialize:** Decode the challenge from base64 and deserialize to construct the `challenge` object.
3. **Verify challenge**
   (a) Checks if `rep_id` exists in the `challenge_map`. If not, resets `challenge_map[rep_id]` to `None` and returns an error response.
   (b) Compares the deserialized `challenge`, `uuid`, and `type` with the stored values in `challenge_map[rep_id]`. If any of the values do not match, resets `challenge_map[rep_id]` to `None` and returns an error response.
4. **Handle success** If the challenge is verified then the representative succeeded the challenge.
   (a) First, process the `b64_serial_rep_message` from the request data. Then create a `Message` object associated with the patient.
   (b) Then remove the challenge information for `rep_id` from `challenge_map`
   (c) Finally save the `Message` in the database and return a positive JSON response.

4.2.4 User challenges

The challenge authentication for the `Patient` to authenticate themselves follows a similar structure as that of the `AuthorityRep`, with two functions to get and validate the challenge.

4.2.5 `get_challenge_patient`

This function generates an encrypted challenge for the patient to authenticate themselves. It performs the following steps:

1. **Generate challenge:** Generate a random group element `challenge` using `ma_abe.helper.get_random_group_elem`
2. **Store challenge information:**Store the tuple `(challenge, patient_id, "PHR")` in `challenge_patient_map` with `patient_id` as the key.
3. **Construct access policy:** `policy` specific to the patient:
   - `policy = f"(PATIENT{patient_id}@PHR)"`

4. **Encrypt challenge:** Encrypt the `challenge` under the constructed policy using `ma_abe.helper.encrypt(challenge, policy)`.
5. **Serialize and encode challenge:**
   (a) Serialize the encrypted challenge using `serialize_encrypted_abe_ciphertext`.
   (b) Encode the serialized challenge to a base64 string `b64_serial_challenge`.
6. **Send challenge:** Return a JSON response containing `b64_serial_challenge` to the patient.

4.2.6 `post_challenge_patient`

This function validates the response to the challenge provided by the patient. It follows these steps:

1. **Parse request data:**
   (a) Parse the JSON request body.
   (b) Check for the presence of `b64_serial_challenge` and `b64_serial_rep_message`. If either is missing, return an error response.
2. **Decode and deserialize challenge:** Decode `b64_serial_challenge` from base64 to obtain `serial_challenge`. and Deserialize `serial_challenge` to reconstruct the `challenge` object.
3. **Verify challenge:**
   (a) Check if `patient_id` exists in `challenge_patient_map`.
   (b) If not, reset `challenge_patient_map[patient_id]` to `None` and return an error response indicating the challenge failed.
   (c) Compare the deserialized `challenge`, `patient_id`, and `"PHR"` with the stored values in `challenge_patient_map[patient_id]`.
   (d) If any value does not match, reset `challenge_patient_map[patient_id]` to `None` and return an error response.
4. **Handle success:**
   (a) Process `b64_serial_rep_message` from the request data and create a `Message` by calling `craft_message(b64_serial_rep_message, patient_id)`.
   (b) Remove the challenge information for `patient_id` from `challenge_patient_map` to prevent replay attacks.
   (c) Save the `Message` object to the database and return a positive JSON response.

4.3 Data model

The Django models represent our server-side PHR system to manage authorities, their representatives, patients and encrypted messages. Figure 1 shows the relations and attributes, which are explained in the following sections.

4.3.1 Authority models

The `Authority` model represents an entity that issues attributes. The authorities in our system are: `PHR`, `HOSPTIAL`, `INSURANCE`, `EMPLOYER`, and `HEALTHCLUB`, which include:

- An `id` and `name` for identification.
- One-to-one relationships with `SecKey` and `PubKey` models to associate keys uniquely with an authority.

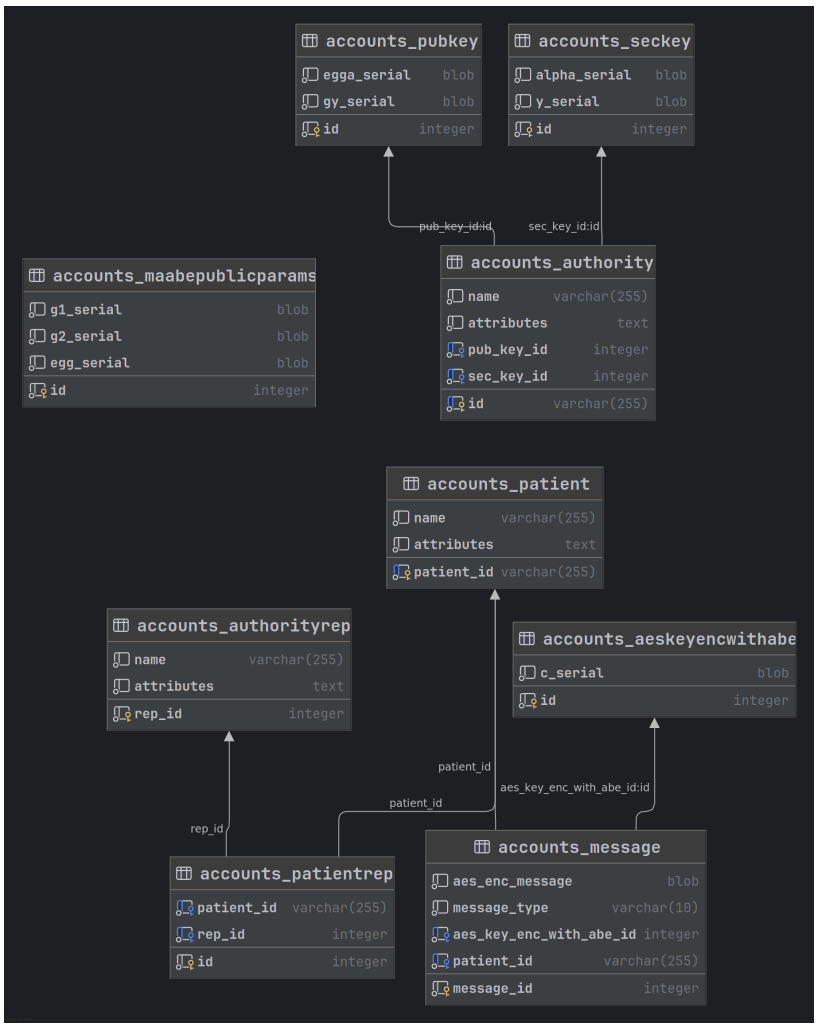

Figure 1: Server Side Model Diagram

- An `attributes` field as a JSONField to store the attributes of the authority.

4.3.2 Key models

`SecKey` and `PubKey` models contain the secret and public keys associated with an authority. Each key model contains binary fields to store serialized key components, such as `alpha_serial`, `y_serial` for `SecKey`, and `egga_serial`, `gy_serial` for `PubKey`. These fields are necessary to construct the key for attributes and user-secret-keys.

4.3.3 Authority representative models

The `AuthorityRep` model represents the representative that is acting on behalf of an authority. The representatives in our system are: `DOCTOR`, `INSURANCEREP`, `EMPLOYERREP`, and `HEALTHCLUBTRAINER`, which include:

- A `rep_id` as the primary key.
- A `name` for identification.
- An `attributes` field as a JSONField to store attributes assigned to the representative.

4.3.4 Patient models

The `Patient` model represents individuals receiving care. It includes:

- A `patient_id` and `name`.
- An `attributes` field as a JSONField to store patient-specific attributes.

The `PatientRep` model creates a linkage between `AuthorityRep` and `Patient`, showcasing representatives assigned to patients.

4.3.5 Encryption models

The `MAABEPublicParams` model holds the public parameters required for MA-ABE. It contains binary fields `g1_serial`, `g2_serial`, and `egg_serial` for serialized group elements needed for authority initialization, construction of keys and encryption. The `AesKeyEncWithAbe` model stores an AES key encrypted with ABE, containing a binary field `c_serial` for the serialized encrypted key.

4.3.6 Message models

The `Message` model represents encrypted messages associated with patients. It includes:

- A foreign key to the `Patient` model.
- A binary field `aes_enc_message` for the encrypted message content.
- A foreign key to `AesKeyEncWithAbe` for the encrypted AES key.
- A `message_type` field with choices ('HEALTH', 'TRAINING') to categorize messages.

4.3.7 Model relationships

- `Authority` is uniquely linked to its keys via one-to-one relationships, ensuring each authority has its own set of keys.
- `AuthorityRep` can be linked to its authority based on the attribute it has. We know that Doctor1 is a doctor at Hospital1 from: `DOCTOR@HOSPITAL1_1`.
- `PatientRep` serves as an associative model between `AuthorityRep` and `Patient`. It utilizes foreign keys to both models, creating many-to-many relationships.
- `Message` associates encrypted messages with patients and the corresponding encrypted AES keys, integrating the encryption mechanism into the data model.

4.4 Abstraction

The `Authority` and `AuthorityRep` models are abstracted to reduce the need for multiple specialized models that contain roughly the same attributes. By generalizing the models we can accommodate for multiple representatives dynamically, which makes our system more flexible and scalable. Furthermore, it simplifies database complexity and migrations.

4.5 Client side

On the client side, we simply store the `user_secret_keys` of the Rep or the Patient, `MAABEPublicParams` and `authority_public_keys` to enable encryption en decryption.

5. Security Considerations

In this section, we assess the security features and limitations of the project's cryptographic architecture. We'll cover data confidentiality, integrity, and attack resistance.

Furthermore, we'll discuss certain dependencies, such as the reliance on trusted servers and our database as well as the Charm framework.

5.1 Security features

5.1.1 Data confidentiality

- **Hybrid encryption for sensitive data:** The project ensures that all sensitive PHR data is encrypted using a hybrid encryption scheme combining AES and MA-CP-ABE. This approach secures the data, ensuring that only authorized users can access and decrypt it. Furthermore, even if the database is compromised, the underlying data remains protected as the AES key can only be decrypted by users who meet the attribute policy requirements.
- **Access policies:** With the use of MA-CP-ABE, access policies embedded in the ciphertext restrict data access to those with specific attributes. This ensures that even if an unauthorized party gains access to the encrypted data, they cannot decrypt it without meeting the policy conditions.

5.1.2 Data integrity

- **Challenge-based authentication** The system uses a challenge-based authentication mechanism to verify users' identities. This prevents unauthorized users from accessing sensitive data, as only those who correctly respond to a unique challenge can proceed, without the need to publicly exchange keys.
- **Multi-Authority Key Distribution:** Each authority manages its own set of attributes, which prevents any single point of failure in key management and strengthens security by distribution access control.

5.1.3 Attack mitigation

- **Replay attacks:** Challenge-based authentication mitigates replay attacks by issuing one-time challenges, ensuring that only real-time responses are valid
- **Collusion Resistance:** Multiple unauthorized users cannot combine their attributes to gain access. Since each authority independently manages its attributes, collusion between authorities is difficult.

5.2 Dependencies

- **Database trust requirement:** The system assumes that the database is a trusted party for key storage and management. If the database is compromised, encrypted keys and metadata could be at risk, potentially exposing the system to attacks even if the data itself remains encrypted. **Important note:** We would like to point out that storing the list of attributes in the DB is only required to load the initial state of the application. That being said, the resulting system does not implement classical access control but a cryptographically enforced one.
- **Server trust requirement:** The dependence on a trusted server does pose notable drawbacks by centralizing risk, reducing privacy, and introducing a potential point of failure. For the scope of this project as a demo it is permitted, in a real-world scenario however, this is not desirable.
- **Charm framework** As a fast prototyping framework Charm works well, however we ran into a lot of difficulties with creating a full-demo system with the module functions that were implemented in `[abenc_maabe_rw15]`.

6. Known Limitations

Our project has a few limitations, most of which are present because of the nature of the project, which is to showcase how to incorporate the studied cryptographic techniques in

order to enforce access control in a Personal Health Record system. That being said, our system is not intended for production use and could be enhanced further by addressing the currently known limitations listed below.

1. **Possible MitM attack against challenges:** During the challenge phase, which is used to check whether a party has sufficient rights to modify data stored on the server, the response to the challenge and the modified data are sent together. The communication during this phase is not protected by TLS. Thus, an adversary can capture the new data that is to be written to the database and alter it. The only prerequisite for this attack to work is that the proof satisfies the challenge.

2. **Pulblic parameters in clear text:** The Public parameters of the ABE scheme are stored in clear text in the database. This means the database must be considered a trusted party when using our system.

3. **Unique AES keys per message:** In our solution, the content of every message (i.e. record in the database) is encrypted with its own AES key. This means that a new key has to be generated for every message. This could be made more efficient by creating a map that holds all Policy, AES key pairs and using the entries of such map when encryption messages according to their policies.

4. **No key revocation:** Our system does not support key revocation, meaning once a party is entitled to participate in the system, their credentials cannot be withdrawn.

5. **Rudimentary UI:** The User Interface provided by the system is meant to be used by someone familiar with the way the system works and does not provide an intuitive way to interact with the system.