KPABE Photo Sharing

A secure photo sharing service with key-policy attribute based encryption

E. Meloni

R. Zippo

University of Pisa, 2018

Index

Index 1
1 Introduction 2
2 The KPABE suite 2
2.1 KPABE and CPABE comparison 2
2.2 Attributes 2
2.3 KPABE Suite 3
2.3.1 Setup 3
2.3.2 Keygen 3
2.3.3 Encrypt 3
2.3.4 Decrypt 3
2.4 Application example 4
2.5 Additional work on KPABE suite 4
3 Architecture 4
3.1 Client-Server communication 5
3.1.1 Server authentication 5
3.1.2 Client authentication 5
3.1.3 Key distribution 5
3.2 Photo sharing scheme 6
3.2.1 Encryption scheme 6
3.2.2 Shared space 6
3.2.3 File organization 7
3.2.4 File formats 7
4 Formal proof 8
4.1.1 Client Private Key exchange 8
4.1.2 Picture exchange 10
5 Additional security measures 11
5.1 Database protection 11
5.2 Memory Protection 11
5.3 Storage protection 11
6 References 12

KPABE Photo Sharing

A secure photo sharing service with key-policy attribute based encryption

1 Introduction

The application here described is a secure photo sharing service which uses KPABE encryption to enforce access to pictures exclusively to allowed users. The application is written in C# for the Windows platform and depends on a KPABE command line implementation developed in C by Yao Zheng [1].

The report is composed of 5 sections. In section 2 we introduce KPABE cryptographic primitives and we show a scenario in which KPABE can be applied. In section 3 we describe the architecture of the Photo Sharing system, detailing client-server communication and how pictures are shared. In section 4 we provide formal proof of key exchange and secure photo sharing using BAN logic. In section 5 we describe some additional security measures of secondary importance.

2 The KPABE suite

KPABE, short for Key-Policy Attribute Based Encryption, is a type of ABE, as the name suggests. ABE is a cryptographic primitive based on public-key encryption, on which the secret key is dependent on attributes assigned to the ciphertext or the key, depending on the type of ABE. Attributes are chosen from a set called the *Attribute Universe*. One important feature of ABE is that it is collusion-resistant, i.e. an adversary with several keys can decrypt a cyphertext if and only if at least one of them can decrypt it.

2.1 KPABE and CPABE comparison

For KPABE, a set of attributes is assigned to the ciphertext, while any private key has a policy assigned. Only ciphertext that satisfy the key policy can be decrypted by that policy. This is opposed to CPABE (Ciphertext-Policy ABE) where the policy is assigned to the ciphertext and the key is assigned a set of attributes. CPABE is closer to Role-based Access Controls, while KPABE can be used in fields such as secure forensics analysis and target broadcast [2]. Ciphertexts encrypted with CPABE grow considerably in size when the policy grows. KPABE in contrast has constant size ciphertext disregarding the attribute set assigned to it.

2.2 Attributes

The implementation that we use classifies attributes in two categories: boolean attributes and numerical attributes. Boolean attributes are identified only by a string, the attribute name. Numerical attributes are identified by a string, the name, and an integer, the bit resolution.

Boolean attributes are evaluated true if they appear in the attribute set of the ciphertext. When numerical attributes are inserted in an attribute set, they must be assigned an unsigned integer value, representable with the attribute bit resolution.

An example of a KPABE attributes and policies is the following:

We suppose that the Universe is composed only of the set $U = \{engineer, branch\}$, where branch is a numeric attribute and engineer is a boolean attribute. User A has a key with policy "`engineer and branch > 1`" while user B has a key with policy "`engineer or branch = 1`". With these keys, a ciphertext marked with $\{engineer, branch = 2\}$ can be decrypted by user B but not user A.

2.3 KPABE Suite

The suite is composed by 4 primitives: *Setup, Keygen, Encrypt, Decrypt*. Setup and Keygen is supposed to be used by a TTP to manage users' private keys. Encrypt and Decrypt are used by the users.

2.3.1 Setup

Setup is used to create the *Master Key* and the *Public Key*. The Master Key must be a kept a secret by the TTP, to avoid that an attacker creates private keys with overly permissive policies and decrypt any ciphertext. The Public Key must be published to any participant, because it is used for encryption and decryption. Setup takes as argument the Attribute Universe and outputs two files, one for the Master Key and one for the Public Key.

2.3.2 Keygen

Keygen is used to create *Private Keys*. Private Keys must be a secret between the TTP and the user, because any attacker who holds the key can decrypt anything that user can, without being detected. Keygen takes as arguments the Master Key, Public Key, and policy, and outputs one file with the Private Key.

2.3.3 Encrypt

Encrypt takes as arguments the Public Key, the plaintext and the attribute set to be assigned to it and outputs one file with the ciphertext. Ciphertext generated this way can be decrypted only by users whose Private Keys policy is satisfied by the ciphertext attribute set.

2.3.4 Decrypt

Decrypt takes as arguments the Public Key, the Private Key, and the ciphertext and outputs one file with the plaintext. This operation completes only if the Private Key policy is satisfied by the ciphertext attribute set.

2.4 Application example

KPABE Photo Sharing could be used in the following scenario for a secure access to confidential information. Suppose that several cameras are distributed in a corporate building and each of them takes a picture every second. Each of them has a numerical ID and the universe has numerical attributes $\{camera_{id}, timestamp\}$. Every picture they take is uploaded with KPABE Photo Sharing with attribute $camera\_id$ set to the camera ID and *timestamp* set to the current one.

If something happens near camera 5 and 6 around a certain hour, and further investigation is needed, security officers can be provided a private key with policy "`(camera_id = 5 or camera_id = 6) and (timestamp >= t1 and timestamp <= t2)`" and consult pictures relevant to the investigation without harming the privacy of workers recorded by other security cameras or out of the relevant timespan.

2.5 Additional work on KPABE suite

During our study of its tools, we found that the implementation by Yao Zheng had a few bugs and undocumented behaviors which we fixed and documented in a forked repository[1]. The main bugfixes are:

- Computation of max integer on N bits, which didn't work for N = 64, during policy creation and attribute parsing.
- `make` and `autoconfigure` files specified dependencies in an order which made `gcc` ignore some of them.

The differences between documented and actual behavior were mainly regarding syntax. We documented the safe syntax for each tool. Key issues are:

- Spaces are indicated to be optional, but can break the parsing in some cases.
- Numerical attributes are documented with optional fields which, if present, break the parsing.

3 Architecture

Our photo sharing system is based on

- KPABE suite for attribute-based encryption
- A file sharing system to synchronize a folder between clients
- A client-server architecture for KPABE keys distribution

Our work consisted in compiling the KPABE suite for the Windows platform and study its use, then build the Photo Sharing system in C#.

[1] Our forks: https://github.com/MeloniZippoProjects/kpabe, https://github.com/MeloniZippoProjects/libcelia

The system is composed of three modules:

- KP Services, a library of common tools, most notably
  - A wrapper class for the command line suite of KPABE. It implements the calls to the executable tools with a high-level interface.
  - Classes abstracting universe and attribute concepts, providing parsing and validation functions
  - Utility classes to provide secure handling of sensitive data both in memory and storage
- KP Trusted Party, a command line application which implements key distribution and authentication. It holds the system master key and all user's private keys. It provides two different interfaces
  - A command line interface to administrate the system. It provides commands to register and remove users, set their KPABE policy, etc.
  - A RESTful interface over HTTPS through which the clients can authenticate and request their keys.
- KP Client, a WPF application that allows users to authenticate themselves, upload, browse and save pictures and albums. The application heavily uses parallelization and caching to optimize performance.

3.1 Client-Server communication

All client-server communication is built on a REST interface through HTTPS. Both client and server implementations use the Grapevine library [3].

3.1.1 Server authentication

As part of the HTTPS protocol used by the REST interface exposed by the server, clients use TLS to authenticate the server via its certificate and establish a session key. HTTPS is mainly needed to protect the user credentials and its private key from a MITM attack.

3.1.2 Client authentication

The client can authenticate itself through a username and password. The trusted party component will generate credentials for the user on request of the admin via its command line interface. Passwords are stored in a database as salted hash, generated by PBKDF2 [4] with iteration count set to 20.000 and salt generated by the CSPRNG exposed by .NET library.

When the client authenticates, the server saves a 256 bits token and sends it to the client. The token then acts as a proof of identity that the client must show for each other operation. The token lifespan is set to 10 minutes.

3.1.3 Key distribution

Through the REST interface clients can obtain from the trusted party

- The KPABE universe, that is the set of attributes valid in the current system.
- The KPABE public key
- Their own KPABE private key

The universe, public and master key are set at server initialization and are considered immutable.

Private keys are instead generated every time a new policy is specified for the user, through the command line interface.

Users are initially created without any policy: this not only simplifies administration procedures, but can also be used in a scenario when data upload is to be permitted, but not decryption of any sort. In the previous application example cameras would need only the public key to upload the captured data. The client application, as is to be used by a user via GUI, does not support this mode and will close if no private key can be retrieved.

3.2 Photo sharing scheme

Photos are shared between clients through a shared space. This space must enforce some rules on the actions users can do to modify it. Files are organized in a folder hierarchy to easily separate keys from ciphertext.

3.2.1 Encryption scheme

The encryption scheme uses a combination of KPABE, AES symmetric encryption and hash chains. The choice of using both KPABE and AES encryption is due to performance as the former is inefficient on large data.

For both single photos and the first image of an album, a new random $\{key, IV\}$ pair is generated and used to encrypt the image with AES (in the example, Ape.png.aes). AES is used in CBC mode with 256-bit keys.

Then, the pair is saved to a file and encrypted using KPABE with the attributes specified for the image or album (in the example, Ape.keys.kpabe).

For album images after the first, hash chains are used: the $\{key, IV\}$ pair is given by hashing the pair of the precedent file. Use of hashing allows for a more efficient retrieval of a specific album image's key. The hashing function used is SHA256, in case of IV only the first 128 bits are used.

Note that the process is done twice: once for the image and once for its thumbnail. At the first step two independent pairs are generated and saved, and two independent hash chains follow in case of albums. The .keys.kpabe file contains both pairs.

3.2.2 Shared space

The system relies on a shared space accessible to all clients, where data is to be stored. Possible implementations could be based on SMB, FTP, or cloud services. In our implementation we abstract this component, asking only for the path through which access the folder using common API calls.

Security features that we expect to be provided by this component are:

- Clients are permitted read access to the relevant subfolders (*keys*, *items*) and all their contents
- Clients can create new files in those folders, but cannot modify, overwrite nor delete any already existent file
- Clients can create new subfolders in the *items* folder, but only the subfolder creator can create new files inside it

3.2.3 File organization

The shared space has two subfolders

- Items
- Keys

As the names suggest, inside subfolder *Items* are stored data files in encrypted format, while inside subfolder *Keys* are stored the keys necessary to access the data files.

Each item is recognized by its name, which must be unique. For each item, there is a key stored in *Keys* and either a file or a folder in *Items*. This is because photos can be uploaded both singularly and as albums: images of an albums are stored inside a folder with the name of the album.

To optimize rendering performance, photos are uploaded together with a downscaled thumbnail.

As an example, for a shared folder containing one photo and an album of two photos, the tree view would be the following:

- Items
    - Ape.png.aes
    - Ape.tmb.png.aes
    - Sfondi
        - Sfondi.0.png.aes
        - Sfondi.0.tmb.png.aes
        - Sfondi.1.png.aes
        - Sfondi.1.tmb.png.aes
- Keys
    - Ape.keys.kpabe
    - Sfondi.keys.kpabe

3.2.4 File formats

As can be seen in the example, we choose to append file extensions to indicate content and encryption scheme, namely

- .tmb for thumbnail files
- .keys for files containing AES keys
- .aes for files encrypted with AES
- .kpabe for files encrypted with KPABE. This last one is suggested by the KPABE suite

For images, we choose to impose the *png* image format to easier handling of naming schemes. The client upload window supports multiple image formats, and operates the appropriate conversion before the encryption process.

For key files, we use JSON to serialize/deserialize ItemKeys objects, which contain the two $\{key, IV\}$ pairs for the item. For JSON processing we used JSON.Net [5].

4 Formal proof

We give proof of confidentiality for the exchange of the Client KPABE Private Key and of a picture saved in the shared area.

A picture is encrypted through KPABE in such a way that every private key which is verified by the picture attribute set can decrypt it. Therefore, there can be many private keys associated to the same attribute set. We can use the abstraction of public key in the BAN Logic saying that P_c is the public key used to encrypt, generated from the set of attributes, and P_c^{-1} is one of the private keys with a policy verified by the attribute set used to generate P_c.

A KPABE private key can be generated only by a party which knows the KPABE Master Key. In our case only the KP Trusted Party knows the Master Key and never shares it.

An object can be encrypted with KPABE if the party knows the KPABE Public Key (not to be confused with the abstraction of a public key generated by the set of attributes). The Public Key can be public knowledge without any security problems.

Since BAN Logic did not consider an exchange of private keys, we extended the notation with the symbol $\mapsto^{P_C^{-1}} C$ meaning that P_c^{-1} is C private key.

4.1.1 Client Private Key exchange

Since we use SSL for communication, we can assume that the client authenticates the server and shares a fresh session key with the server. Replay attacks are avoided by SSL.

After login, the user sends its token along with its private key request. If the token is recognized by the Trusted Party, it sends the client private key.

The token can be idealized as a secret T between C and S

Assumptions

SSL guarantees the creation of a fresh session key. But only C is sure of S identity.

$$C \models C \overset{K}{\leftrightarrow} S, \qquad C \models \#\left(C \overset{K}{\leftrightarrow} S\right)$$

The token T is a shared secret between C and S

$$C \models C \overset{T}{\Leftrightarrow} S, \qquad S \models C \overset{T}{\Leftrightarrow} S$$

C trusts the server S to control the private keys.

$$S \models \mapsto^{P_C^{-1}} C, \qquad C \models S \Rightarrow \mapsto^{P_C^{-1}} C$$

Message 1: $C \rightarrow S: \{\langle R \rangle_T\}_K$

The server doesn't know that he's talking with C, so C sends its HTTP request R combined with its secret token T. Since $\langle R \rangle_T$ is encrypted with the session key, the confidentiality of T is preserved. Since S hasn't authenticated C yet, we use U as unknown.

$$\frac{S \lhd \{\langle R \rangle_T\}_K, \quad S \models U \overset{K}{\leftrightarrow} S}{S \lhd \langle R \rangle_T}$$

$$\frac{S \lhd \langle R \rangle_T, \quad S \models C \overset{T}{\Leftrightarrow} S}{S \models C|{\sim}R}$$

Message 2: $S \rightarrow C: \{P_c^{-1}\}_K$

Now S has authenticated C and it is ready to send C private key. The private key is encrypted with the session key, thus only C can read it. Therefore, P_c^{-1} confidentiality is preserved.

$$\frac{C \lhd \left\{\mapsto^{P_C^{-1}} C\right\}_K, \; C \models C \overset{K}{\leftrightarrow} S}{C \models S|{\sim} \mapsto^{P_C^{-1}} C}$$

Since K is fresh, also any value encrypted with it is fresh.

$$\frac{C \models \#\left(C \overset{K}{\leftrightarrow} S\right)}{C \models \#\left(\mapsto^{P_C^{-1}} C\right)}$$

Therefore, C can be sure that S believes that P_C^{-1} is C private key.

$$\frac{C \models S|{\sim} \mapsto^{P_C^{-1}} C, \quad C \models \#(\mapsto^{P_C^{-1}} C)}{C \models S \models \mapsto^{P_C^{-1}} C}$$

Conclusions

Finally, for the jurisdiction rule:

$$\frac{C \models S \models\mapsto^{P_C^{-1}} C, \quad C \models S \Rightarrow\mapsto^{P_C^{-1}} C}{C \models\mapsto^{P_C^{-1}} C}$$

Therefore, C knows its KPABE private key.

4.1.2 Picture exchange

We apply BAN Logic using the shared folder as the unsecure channel of communication. A file in this folder can be abstracted as a message between the uploader and the client which tries to read the file.

Assumptions

C already obtained P_c^{-1} so C knows that P_c^{-1} is its public key.

$$C \models \mapsto^{P_C^{-1}} C$$

Based on how the hash chain work, we can also say that if one key in the chain is known, also the next can be computed.

$$\frac{C \lhd K_{obj_i}}{C \lhd K_{obj_{i+1}}}$$

Message 1: $U \rightarrow C : \{K_{obj_0}\}_{P_C}$

$$\frac{C \lhd \{K_{obj_0}\}_{P_C}, \mapsto^{P_C^{-1}} C}{C \lhd K_{obj_0}}$$

Using $\frac{C \lhd K_{obj_i}}{C \lhd K_{obj_{i+1}}}$ and starting from K_{obj_0}, C can compute $K_{obj_i}\ \forall i$

$$\frac{C \lhd \{K_{obj_0}\}_{P_C}, \mapsto^{P_C^{-1}} C}{\forall i\ C \lhd K_{obj_i}}$$

Message 2: $U \rightarrow C : (\{obj_0\}_{K_{obj_0}}, \ldots \{obj_N\}_{K_{obj_N}})$

$$\frac{C \lhd \{obj_i\}_{K_{obj_i}}, C \lhd K_{obj_i}}{C \lhd obj_i}$$

Conclusion

This can be repeated for every $i = 0..N$ and therefore we have the conclusion

$$\frac{C \lhd \{K_{obj_0}\}_{P_C}\ C \models \mapsto^{P_C^{-1}} C}{\forall i\ C \lhd obj_i}$$

5 Additional security measures

As the application handles sensible data during its execution, we adopted various solutions to minimize the attack surface, especially for side channel attacks. The areas we focused on are

- Database
- Memory
- Storage

5.1 Database protection

Since all keys are stored in the database, if the database is compromised, the attacker can compromise all user keys and the KPABE master key. To prevent this, we use Data Protection API offered by Windows operating systems [6] to encrypt keys before storage, so that they are accessible only within the scope of the Windows user running the server. DPAPI keys are completely managed by the OS and tools exist for migrating these keys on another server.

5.2 Memory Protection

To reduce the attack surface from memory scraping, the memory cache uses Data Protection API to encrypt the data, this time using process scope so that only the same client that encrypted the data can have the OS decrypt it.

For this purpose, we wrote the classes *SecureBytes* and *TemporaryBytes*. The former holds data encrypted with DPAPI. When the data is required for use it is copied on a new array held by a *TemporaryBytes* object, which is a disposable object that clears (all bytes set to 0) the array after use.

This is especially important client side, since to improve performances and response times the client uses caching to store AES keys and decrypted image data, which greatly increments the lifespan of such data in memory.

5.3 Storage protection

Since the KPABE command line suite only deals with files, there is the need of copying keys from the protected memory to a temporary file. To reduce the probability of someone compromising the keys by reading the temporary file, we keep them in storage only for the time needed for computation. When the files are not needed anymore they are deleted, and the bytes on the hard drive are shredded to prevent an attacker to read the previous content.

The shredding process is implemented with *SecureFile*, a disposable class similar to *TemporaryBytes*. The shredding code is taken from [7].

6 References

[1] Y. Zheng, "KPABE Suite," 2014. [Online]. Available: https://github.com/gustybear/kpabe.

[2] V. Goyal, O. Pandey, A. Sahai and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *13th ACM conference on Computer and communications security*, Alexandria, Virginia, USA, 2006.

[3] S. Offen, "Grapevine," 2016. [Online]. Available: https://github.com/sukona/Grapevine.

[4] B. Kalinski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," *RFC 2898*, September 2000.

[5] Newtonsoft, "JSON.Net," 2008. [Online]. Available: https://www.newtonsoft.com/json.

[6] Microsoft Corporation, "Windows Data Protection," October 2001. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms995355.aspx.

[7] J. Martensson, "Securely Delete a File using .NET," 2008. [Online]. Available: https://www.codeproject.com/Articles/22736/Securely-Delete-a-File-using-NET.