# Implementation of Ciphertext-Policy Attribute Based Encryption algorithm

Mateusz Olszewski

Jagiellonian University, Kraków, Poland
*Email address*: mtsz.olszewski@student.uj.edu.pl

Abstract. In this paper, we will provide an overview of the Ciphertext-Policy Attribute Based Encryption algorithm - method for encrypting data for remote server access, in a way that the data is safe, even if the server is compromised.

Then we will see how this algorithm can be implemented in the programming language of our choice (Golang) and how it performs on different test cases.

Most of this paper is based on the 2007 *Ciphertext-Policy Attribute-Based Encryption* work by J. Bethencourt, A. Sahai and B. Waters[**1**]

# Contents

## 1. Problem

As more and more data is being stored in the distributed client-server systems, keeping the information secure is even more urgent issue. There are several ways to keep the data encrypted and to give the access only for user who meets certain qualifications. In one way we can use public key encryption method to encrypt data to a particular user, however by doing so we are unable to handle more expressive types of access control such as attribute-based access control[**2**].

In an attribute-based access control we are implementing access control policies that can be customised using a wide range of attributes. For example suppose that the company Bajtex would like to give access to the encrypted data only for infosecurity division in Kraków and Suwałki or for an employee named Bajtazar or for the higher ups. We can specify access structure for accessing this information the following way:

(("DIVISION": "Infosecurity") AND ("LOCATION":"Kraków" OR "LOCATION":"Suwałki")) OR (NAME: "Bajtazar") OR ("POSITION" : "Director")

This way we don't need to know the identity of every person that will access our secured data, but still have control over who can access it. Traditionally it is enforced by employing a trusted server to store data locally and check if the user asking for access has proper attributes for this operation. However, when the data is stored at several locations it forces us to copy the stored data in order to ensure performance and reliability, but it also increases risk of stealing the data when the server is compromised. For these reasons we want to create a system in which the data is securely encrypted even if the server storing it is compromised.

## 2. Solution

In ciphertext-policy attribute-based encryption we are addressing all the mentioned issues in such a way that the user's private key is associated with a number of attributes expressed as strings. For each set of attributes we will specify an *access structure*, giving us an ability to encrypt and decrypt the messages over this structure. In mathematical terms access structure is described by the *monotonic access tree*, where nodes are composed of threshold gates and the leaves describe attributes. User with a given set of attributes will be able to decrypt the message encrypted with this tree only if these attributes satisfy a given set of nodes, which recursively satisfy a root node. There are two main types of nodes:

- *Attribute Node*: a leaf in a tree which represents a specific attribute required to access the encrypted data. It is satisfied only if the user has given attribute associated with this node.
- *Threshold Gate*: an interior node which represents various logical operations. Each of them is described by its children and a threshold value. *AND* gates are constructed as *n-of-n* threshold gates - meaning that to satisfy this node *every* child node of its will have to be satisfied too. Analogically we define *OR* gates as *1-of-n* threshold gates. This way to satisfy this node *only one* child node will have to be satisfied.

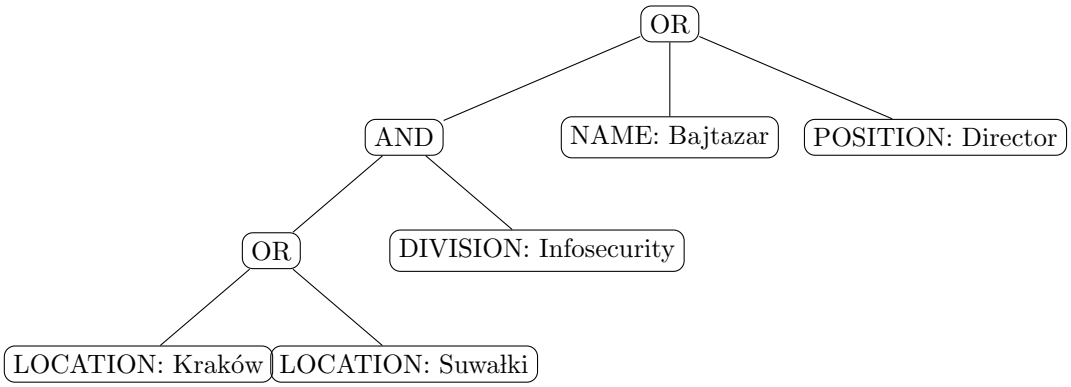


FIGURE 1. Example access structure mapped as tree

In this example user can decrypt the message encrypted on the given tree only if:

- Their name is Bajtazar
- Or they have position of director in the company
- Or they work in the infosecurity division and have location based in either of two cities: Kraków or Suwałki

We can see that with those attributes we can recursively satisfy root node, which in return authorises us to decrypt the message. Mathematical background of this implementation will be discussed in the following chapters.

## 3. Background

Our algorithm consists of four core algorithms:

- **Setup**: We generate the public parameters $PK$ and the master key $MK$, given the implicitly chosen parameters.
- **Encrypt(PK,M,A)**: Given public parameters $PK$, a message $M$, and an access structure $A \subseteq U$ we run the Encrypt algorithm to encrypt the message $M$. The encryption process generates a ciphertext $CT$, such that only by possessing a set of attributes that is authorized in the access structure $A$, we are able to decrpyt the message $M$ from $CT$. This way ciphertext $CT$ also implicitly contains access structure $A$.
- **Key Generation(MK)**: From master key $MK$ we generate a private key $SK$
- **Decrypt(PK,CT,SK)**: User possessing a private key $SK$ and a ciphertext $CT$ can run the Decrypt algorithm to obtain the plaintext message $M$. Decryption is successful if the attributes associated with $SK$ satisfy the access policy $A$ in $CT$.

## 4. Formal definition

DEFINITION 1. (Bilinear Maps) *Let $\mathbb{G}_0$ and $\mathbb{G}_1$ be multiplicative cyclic groups of prime order $p$.*

*A bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_0 \to \mathbb{G}_1$ is a map that satisfies the following properties:*

*1. Bilinearity: For all $u, v \in \mathbb{G}_0$, $a, b \in \mathbb{Z}_p$, we have: $e(u^a, v^b) = e(u, v)^{ab}$*

*2. Non-degeneracy: There exist generator* $g \in \mathbb{G}_0$ *such that* $e(g, g) \neq 1$. *(not being identity element of* $G_1$*)*

We say that $\mathbb{G}_0$ is a bilinear group if the group operation in $\mathbb{G}_0$ and the bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_0 \to \mathbb{G}_1$ are both efficiently computable. We can notice that the map $e$ is symmetric since $e(g^a, g^b) = e(g, g)^{ab} = e(g^b, g^a)$.

An example simple billinear group can be mapping as $e(x, y) = 3^{xy}$, where $\mathbb{G}_0 = \mathbb{Z}_5$ and $\mathbb{G}_1 \leqslant \mathbb{Z}_{11}^*$. $\mathbb{G}_1$ has order of $p = 5$, the same as $\mathbb{G}_0$, because it is a subgroup of powers of 3 in $\mathbb{Z}_{11}^*$, that is with elements $\{1, 3, 4, 5, 9\}$.

We can see that it is billinear because it holds:

- Bilinearity - for any $u, v \in \mathbb{G}_0, a, b \in \mathbb{Z}_5$ we have $e(u^a, v^b)$. Then we have $u^a = u \cdot a$, because $u^a$ in $\mathbb{G}_0$ is adding $u$ to itself $a$ times. Then we have $e(u^a, v^b) = 3^{u \cdot a \cdot v \cdot b} = (3^{u \cdot v})^{a \cdot b} = e(u, v)^{a \cdot b}$
- For generator $1 \in \mathbb{G}_0$, we have $e(1, 1) = 3 \neq 1$

However this kind of mapping is not effective cryptography-wise. In our implementation we are using mapping:

- from $\mathbb{G}_0$, which elements are $y^2 = x^3 + x$ elliptic curve points over field $F_q$ for some random prime $q = 3 \ mod \ 4$.
- to $\mathbb{G}_1$, which is a subgroup of $F_{q^2}$ - extended field used in $\mathbb{G}_0$

The mathematics behind this mapping are quite complex, which are not in the scope of this paper. Explanation of the theoretical working of this library can be found in the phd dissertation of Benn Lynn[**3**], who created pairing-based cryptographic library, which we use in our implementation.

### 4.1. Access tree.

DEFINITION 2. (Access Structure) *Let* $\{P_1, P_2, \ldots, P_n\}$ *be a set of parties. Collection* $A \subseteq 2^{\{P_1, P_2, \ldots, P_n\}} = U$ *is* monotone *if* $\forall B, C$ : *if* $B \in A$ *and* $B \subseteq C$ *then* $C \in A$. *The* monotone access structure *is a monotone collection* $A$ *of non-empty subsets of* $\{P_1, P_2, \ldots, P_n\}$, *i.e.,* $A \subseteq 2^{\{P_1, P_2, \ldots, P_n\}} \setminus \{\emptyset\}$. *The sets in* $A$ *are called the* authorised sets, *and the sets not in* $A$ *are called the* unauthorised sets.

Throughout the paper, when we are writing about an access structure, we mean a monotone access structure.

In our construction private keys are identified with a set $S$ of descriptive attributes which are used to create access structure tree $T$ used for decryption and encryption. The tree consists of two types of nodes, as described in chapter two - *threshold gates* (and/or nodes) and *attribute nodes*. For nodes in access tree we define:

- $k_x$ as the threshold gate's threshold value - a minimum number of its child nodes needed to be satisfied, to satisfy this node. If the node is a *OR* gate $k_x$ equals 1. If the node is a *AND* gate $k_x$ equals $num_x$, where $num_x$ is the number of child nodes of $x$.
- $parent(x)$ as a parent of a given node
- $att(x)$ as an attribute associated with a given attribute node
- $T_x$ as a subtree rooted in a given node $x$
- $index(x)$ as a unique number of a node $x$ related to its parent. The access tree defines an ordering between the children of every node, that is, the children of a node $y = parent(x)$ are numbered from 1 to $num_y$. The function $index(x)$ returns such a number associated with the node $x$

**Satysfying an access tree**: if a set of attributes $\gamma$ satisfies the access tree $T_x$ we denote it as $T_x(\gamma) = 1$ We compute it recursively as follows. If $x$ is a non-leaf node, evaluate $T_{x'}(\gamma)$ for all children $x'$ of node $x$. $T_x(\gamma)$ return true only if at least $k_x$ children return 1. If $x$ is a leaf node, then $T_x(\gamma) = 1$ if and only if $att(x) \in \gamma$. User is able to decrypt ciphertext $CT$ only and only if there is an assignment of attributes from the private key to nodes of tree such that the tree is satisfied. The access tree is rooted at a policy node, which represents the overall access policy for the encrypted data.

**4.2. Formal construction of algorithms.** Besides the previously defined billinear maps and access trees we will we also define the Lagrange coefficient $\Delta_{i,S}$ for $i \in \mathbb{Z}_p$ and a set $S$ of elements in $Z_p$: $\Delta_{i,S}(x) = \prod_{j \in S, j \neq i} \frac{x-j}{i-j}$. Finally we define a hash function $H : \{0,1\}^* \to \mathbb{G}_0$ that randomly maps any attribute described as a binary string to a random group element

- **Setup** We choose given parameters: bilinear group $\mathbb{G}_0$ of prime order $p$ with generator $g$, two random exponents $\alpha, \beta \in \mathbb{Z}_p$. The public key is generated as: $PK = (\mathbb{G}_0, g, h = g^\beta, f = g^{1/\beta}, e(g,g)^\alpha)$ and the Master Key is: $MK = (\beta, g^\alpha)$
- **Encrypt(PK,M,T)** We want to encrypt a message under the tree access structure $T$. We're doing it in the given steps:
  (1) For each node $x$ in the tree, we choose the polynomial $q_x$ with the degree $d_x = k_x - 1$, where $k_x$ is the threshold value of the node
  (2) Starting with the root node r, the algorithm chooses at random $s \in \mathbb{Z}_p$ and sets $q_r(0) = s$. For each other $d_r$ coefficients of the polynomial $q_r$ we are choosing them randomly.
  (3) Then for other nodes $x$ it recursively sets $q_x(0) = q_{parent(x)}(index(x))$ and chooses $d_x$ other coefficients randomly
  (4) Finally, let $Y$ be the set of leaf nodes in $T$. The ciphertext $CT$ is constructed by computing $CT = (T, \tilde{C} = Me(g,g)^{\alpha s}, C = h^s = g^{\beta \cdot s}, \forall y \in Y : C_y = g^{q_y(0)}, C'_y = H(att(y))^{q_y(0)})$
- **KeyGen(PK,MK,S)** For given set of attributes $S$ we want to output a key that identifies with this set:
  (1) We choose random $r \in \mathbb{Z}_p$ and then random $r_j \in \mathbb{Z}_p$ for each attribute $j \in S$
  (2) We compute the key as $SK = (D = g^{(\alpha+r)/\beta}, \forall j \in S : D_j = g^r \cdot H(j)^r, D'_j = g^{r_j})$
- **Decrypt(CT,SK)** We define our algorithm recursively:
  (1) At first we define a recursive algorithm DecryptNode(CT, SK, x) that takes as input a ciphertext $CT = (T, \tilde{C}, C, \forall y \in Y : C_y, C'_y)$, a private key SK, which is associated with a set $S$ of attributes, and a node $x$ from $T$.

(2) If the node $x$ is a leaf node then we let $i = att(x)$, $H(i) = g^{\gamma}$. If $i \in S$, then:

$$\begin{aligned}\text{DecryptNode } (CT, SK, x) &= \frac{e(D_i, C_x)}{e(D_i', C_x')} \\ &= \frac{e(g^r \cdot H(i)^{r_i}, g^{q_x(0)})}{e(g^{r_i}, H(i)^{q_x(0)})} \\ &= \frac{e(g^{r+\gamma \cdot r_i}, g^{q_x(0)})}{e(g^{r_i}, g^{\gamma \cdot q_x(0)})} \\ &= \frac{e(g,g)^{(r+\gamma \cdot r_i) q_x(0)}}{e(g,g)^{\gamma \cdot r_i \cdot q_x(0)}} \\ &= e(g,g)^{r q_x(0)}\end{aligned}$$

If $i \notin S$, then we define DecryptNode(CT, SK, x)=$\perp$

(3) If $x$ is a non-leaf node, the algorithm goes over all children $z$ of $x$ and computes the results of DecryptNode(CT, SK, z). Let's name this output as $F_z$ Let $S_x$ be an arbitrary $k_x$-sized set of these child nodes $z$, such that $F_z \neq \perp$. If none such set exists we return $\perp$. Otherwise, we compute, where $i = index(z)$ and $S_x' = \{index(z) : z \in S_x\}$:

$$\begin{aligned}F_x &= \prod_{z \in S_x} F_z^{\Delta_{i,S_x'}(0)} \\ &= \prod_{z \in S_x} (e(g,g)^{r \cdot q_z(0)})^{\Delta_{i,S_x'}(0)} \\ &= \prod_{z \in S_x} (e(g,g)^{r \cdot q_{parent(z)}(index(z))})^{\Delta_{i,S_x'}(0)} \quad \text{(by construction)} \\ &= \prod_{z \in S_x} e(g,g)^{r \cdot q_x(i) \cdot \Delta_{i,S_x'}(0)} \\ &= e(g,g)^{r \cdot q_x(0)} \quad \text{(using polynomial interpolation)}\end{aligned}$$

and return $F_x$ as a result.

(4) Finally, when we call DecryptNode(CT,SK,r) on the root node $r$ and we get $F_r = e(g,g)^{r q_n(0)} = e(g,g)^{rs}$, we can decrypt the message by computing: $\tilde{C}/(e(C,D)/F_r) = \frac{\tilde{C} \cdot F_r}{e(C,D)} = \frac{Me(g,g)^{\alpha s} e(g,g)^{rs}}{e(g^{\beta s}, g^{(\alpha+r)/\beta})} = \frac{Me(g,g)^{\alpha s + rs}}{e(g,g)^{s(\alpha+r)}} = M$

## 5. Discussion

**5.1. Security intuition.** In order to decrypt the message from $\tilde{C} = Me(g,g)^{\alpha s}$ an attacker must recover $e(g,g)^{\alpha s}$. In order to do this an attacker must pair $C$ from $CT$ with $D$ from some user's private key. This will result in the desired value $e(g,g)^{\alpha s}$ blinded by some value $e(g,g)^{\alpha s}$. This value can be blinded out if and only if the user has the correct key components to satisfy the secret scheme embedded in the ciphertext.

**5.2. Efficiency.**

- The encryption algorithm requires two exponentiations for each leaf in ciphertext's access tree.
- The ciphertext size will include two group elements for each tree leaf.

- The key generation algorithm requires two exponentiations for every attribute given to the user, and the private key consists of two group elements for
- The decryption algorithm could require two pairings for every leaf of the access tree that is matched by a private key attribute and (at most) one exponentiation for each node along a path from such a leaf to the root. However, this can be optimised by various enhancements.

**5.3. Key-revocation and numerical analysis.** It's hard to easily implement key revocation system in cp-abe, since several different users might match the description policy. The usual approach is to append to each of identities/descriptive attributes a date for when the attribute expires. For example instead of using the attribute "Computer Science" we might use the attribute "Computer Science: Oct 17, 2006". However, if we wish for a party to be able to specify policy about revocation dates this way on given date scale, users would be forced to go often to the authority for new keys and maintain a large amount of private key storage.

One way to mitigate that issue is to allow a key authority to give out a single key with some expiration date $X$ rather than a separate key for every time period before $X$. When a party encrypts a message on some date $Y$, a user with a key expiring on date $X$ should be able to decrypt if and only if $X \geq Y$ and the rest of the policy matches the user's attributes. In this manner, different expiration dates can be given to different users and there does not need to be any close coordination between the parties encrypting data and the authority.

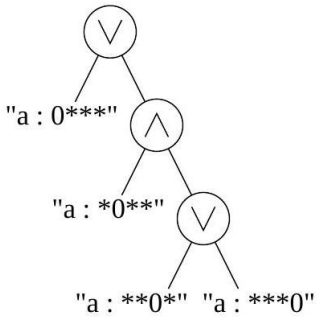


FIGURE 2. Policy tree implementing the integer comparison $a < 11$

This sort of functionality can be realised by extending our attributes to support numerical values and our policies to support integer comparisons. To represent a numerical attribute "a = k" for some n-bit integer k we convert it into a "bag of bits" representation, producing n (non-numerical) attributes which specify the value of each bit in k. As an example, to give out a private key with the 4-bit attribute "a = 9", we would in- stead include "a : 1***", "a : *0***", "a : **0*", and "a : ***1" in the key. We can then use policies of AND and OR gates to implement integer comparisons over such attributes, as shown for $a < 11$ in Figure 1. There is a direct correspondence between the bits of the constant 11 and the

choice of gates. Policies for $\leq$, $>$, $\geq$, and $=$ can be implemented similarly with at most n gates, or possibly fewer depending on the constant.

## 6. Implementation

As an effort to understand the concept of ciphertext-policy attribute based encryption even better I've implemented it as a library in golang.

Currently it's publicly available in my personal github repository[**4**]. It uses wrapper library for cryptographic pairings, which is based on Pairing-Based Cryptography (PBC).[**5**] It allows for user to setup access policy, generate user's private key and master secret keys based on their attributes. Then it allows for encryption and decryption of given text file. More information is in the *README.md*.

# Bibliography


[1] J. Bethencourt, A. Sahai and B. Waters, *Ciphertext-Policy Attribute-Based Encryption,* 2007 IEEE Symposium on Security and Privacy (SP '07), Berkeley, CA, USA, 2007, pp. 321-334, doi: 10.1109/SP.2007.11.

[2] Hu, Vincent C. and Kuhn, D. Richard and Ferraiolo, David F. and Voas, Jeffrey, *Attribute-Based Access Control,* in Computer, vol. 48, no. 2, pp. 85-88, Feb. 2015, doi: 10.1109/MC.2015.33.

[3] On the Implementation of Pairing-Based Cryptosystems, Ben Lynn - https://crypto.stanford.edu/portia/pubs/articles/L1194130135.html

[4] https://github.com/mtdrewski/go-abe

[5] Wrapper: https://pkg.go.dev/github.com/Nik-U/pbc
Base library: https://crypto.stanford.edu/pbc