# Dynamic Location Information in Attribute-based Encryption Schemes

Iwailo Denisow, Sebastian Zickau, Felix Beierle, and Axel Küpper
Service-centric Networking
Technische Universität Berlin | Telekom Innovation Laboratories | Berlin, Germany
iwailodenisow@mail.tu-berlin.de | {sebastian.zickau|beierle|axel.kuepper}@tu-berlin.de

***Abstract*—Attribute-based encryption (ABE) allows users to encrypt (cloud) data with fine-grained Boolean access control policies. To be able to decrypt the ciphertext, users need to have a private key with the associated attributes. If the attributes satisfy the formula, the plaintext can be recovered. In this paper, ABE is extended with dynamic attributes. This allows attributes to be added to an existing private key. A server component named Attribute Authority is introduced. By using these dynamic attributes, it is now possible to have the decryption depend on data that changes often, such as location information of a mobile device. Two schemes were developed that convert location data into usable ABE attributes. To demonstrate our results, an Android application was implemented and evaluated in a field test.**

***Index Terms*—Location-based Access Control, Attribute-based Encryption Schemes, Dynamic Attribute Information, Cloud Data.**

## I. Introduction and Motivation

There is a vast and growing amount of private and confidential cloud data [1]. Often there is the need to share this cloud data. Some of the newly emerging technologies simplify the way this can be done. One relatively recent development is *attribute-based encryption (ABE)* [2]. It allows the user to specify an *access policy* within the process of data encryption. The *access policy* can be used to control who has access to the data. To determine if a user can access a file, the *attributes* within user's *private key* are checked against the *access policy*. They can contain some of the characteristics of users, e.g., the date of birth, gender or domain/application specific attributes, such as the position in the company hierarchy, e.g., employee, manager, or external contractor.

However, one of the main problems with ABE is that the *attributes* that are given to a user are static. To add or change an *attribute*, the user has to be given a new *private key*. This prohibits the use of *attributes* that change often, such as data related to time: time of day or days of employment. It would also be very beneficial to be able to express that certain files can only be decrypted in specified time periods. Another property that changes often is the location of a mobile user. Due to an ever-increasing amount of mobile devices, users now have the ability to gather information about their location at almost any time. This makes it desirable to use this information as part of the encryption scheme, as a new approach to *location-based access control (LBAC)* [3]. The infrastructure of ABE currently has no facilities to support *location attributes*. The main goal of this work is extend the expressiveness of the access policies by implementing *dynamic location attributes*. To support *location attributes* in a cloud computing environment, where multiple users need access to the attributes, a client-server architecture is required. Since *location* information is primarily available on mobile devices, a client application needs to be developed. One common problem of *location-based services (LBS)* is that the location information sent by a mobile device can be manipulated. In this work, the reliability or verification of positioning information are only of secondary importance, and it is presumed that the locations are correct or that the involved parts of the system can be trusted.

The following section will give an overview of the related work and the basic technologies that have been used. Sections III and IV describe our main contributions and the implementation. In Section V the solutions are evaluated. Section VI concludes the paper and gives an outlook on future work.

## II. Related Work

This section will give an overview of related work and the technologies used during the realization and evaluation.

### A. Attribute-based Encryption

ABE is a family of asymmetric encryption schemes that depend on the use of pairing-based cryptography. The first scheme of its kind was devised by Sahai and Waters and published in 2006 [4]. In an ABE scheme there are three different types of keys. In a first step, a *key authority* generates a *secret master key* and associated *public keys*. The *key authority* should be a trusted entity since it has access to the *secret master keys*. These *secret master keys* can be used to generate *private keys*. A *private key* can be used to decrypt data. The *public keys* can be used to encrypt data, and may be distributed freely. With ABE, it is possible to encrypt data with a fine-grained access control policy. This is achieved through the use of *attribute sets* that are given to the users. *Private keys*, which can only be created with the *secret master key*, contain such *attribute sets*. When encrypting a file, the user needs to specify a *policy*, a Boolean formula describing what *attributes* are required for decryption, e.g., *((employee and hiringdate < 2012) or manager)*. This policy describes that a file is only accessible either by managers or employees that have been hired before 2012. If the *attribute set* of the *private key* satisfies this Boolean formula, then access to the plaintext is granted. One big advantage of ABE is that the generation of *private keys* and the decryption is decoupled. *Private keys* generated with certain *attributes* will always be able to decrypt data for which they fulfill the Boolean formulas, without ever communicating to the *key authority* again.

This is not always a desirable property, since that means the revocation of keys is not possible without re-encrypting the data or redistributing every *private key*. Newer ABE approaches can account for this problem with revocation through the use of a proxy [5] or through policy delegation [6], which allows updating the policy of a file to a more restrictive policy requiring only the *public key*. However, because of the decoupled nature of ABE, the attributes are static. They can only be changed by distributing a *new key*. This is why *attributes* cannot easily be used for location information the location of a mobile device changes frequently. An important characteristic of ABE is the prevention of collusion attacks. As an example, a file that is encrypted with the policy ($A$ and $B$) and two users. The first user has a *private key* with the *attribute* $A$ while the second user has a *private key* with the *attribute* $B$. Each one of them is not able to decrypt the file, since both *attributes* taken separately don't satisfy the formula. It is then a desirable property of ABE that both users cannot combine their keys to form a single key which will be able to decrypt the file. And in fact, ABE does fulfill this property, called collusion resistance, as described in [2]. There are two variants of ABE. In *Ciphertext-Policy ABE* (CP-ABE), the Boolean formula (also called the policy) is saved in the ciphertext (the encrypted file). The *attributes* that need to satisfy are saved in a *private key*. In *Key-Policy ABE* (KP-ABE), these two roles are directly reversed: the *private key* holds the formula and the ciphertext possesses *attributes*. There are five basic functions in CP-ABE:

- **Setup** - Generates a *public key* and an associated *secret master key*.
- **Encrypt** - Encrypts given data with a *policy*. A *public key* is needed.
- **Keygen** - Creates a *private key* with attributes. To do this a *secret master key* is needed. This *private key* is associated with the *secret master key* and its *public keys*.
- **Decrypt** - Needs a ciphertext and a *private key*. Decrypts the ciphertext only if the attributes in the *private key* satisfy the *policy* and the *private key* is associated with the *public key* that has been used to initially encrypt the file.
- **Delegate** - Needs only a *private key*. Generates a new *private key* with a subset of the *attributes*.

There are two types of atomic formulas: *normal attributes* and *numerical attributes*. A *normal attribute* is just a string of letters and digits that begins with a letter, e.g., employee, manager, workgroup301, male, or female. A *numerical attribute* however has a name and an associated value. As an *attribute* in a *private key* it would look like this: *hiringdate = 2014*, *birthyear = 1990*. Only integers in the range from $0$ to $2^{64} - 1$ are supported. In access policies these numerical attributes can be compared to numbers with the following operators: $=, <, >, \geq, \leq$, e.g., *hiringdate* $\leq$ *2014*, *birthyear = 2000*, *accesslevel* $\geq$ *8*. It is not possible to compare two different numerical attributes of a user, so the following is not possible: *count_children* $>$ *count_pets*. The second argument of a numerical comparison always needs to be a number that is known at the time of encryption.

For the access policy itself, there are three possible operators: *or*, *and*, and *of*. These operators can be combined to form complex formulas. A file might be encrypted with the following access policy: *((employee and hiringdate* $<$ *2014 and accesslevel* $>$ *3) or (employee and accesslevel* $>$ *5) or manager)*. As is usual for asymmetric encryption algorithms, ABE is supposed to be used as a key encapsulation scheme, i.e., as a hybrid encryption scheme [7]. This is a typical way to construct systems using asymmetric encryption since the asymmetric part is usually very slow. This is especially necessary for ABE, because of the computational complexity and the resulting time needed to encrypt or decrypt these keys.

## B. Dynamic Attributes

There have been a number of publications that discuss the addition of *dynamic attributes* to ABE [8][9]. However, each of these findings has some drawbacks.

For example, in Weber's approach from 2009 [8], the *dynamic attributes* are inserted into a specific point in the *policy tree*. In a static subtree, only the *static attributes* are saved, whereas in a dynamic subtree the *dynamic attributes* are saved. *Both* subtrees need to evaluate to true, because they are combined with an *AND*. These *dynamic attributes* are actually part of the policy, but only this single construction is made to be able to handle *dynamic attributes*. This lowers the expressiveness of ABE by requiring that the *dynamic attributes* need to evaluate positively or the file cannot be decrypted. An example for a formula that would not be possible to implement using this approach: *(manager or (employee and (time* $>$ *8 and time* $<$ *16)))* where *time* is a *dynamic attribute*. In 2012, Weber [10] proposed a way to combine *location-based encryption* (LBE) [11] and ABE. This solution does not integrate the dynamic information into the *attributes* or into the policies of ABE. It rather acts as an additional layer on top of the normal encryption and decryption. As explained earlier, ABE is used in a hybrid encryption scheme and encrypts a relatively small amount of data. This encrypted data is used as a *symmetric key* for the encryption or decryption using symmetric algorithm, such as AES. In the proposed addition of dynamic locations to ABE by Weber this *dynamic key* is XORed with the result of the LBE, resulting in a final *key* that is then used to encrypt and decrypt. A problem with this solution is that *all* encrypted files using this method then require users that want to decrypt the file to be at the specified location. It is not possible to make this dependent on a *part of the formula*. For example, the policy *manager or (employee and location:office)* describes that a file can only be decrypted if the person is a manager or if the person is an employee that is currently in the office. This policy cannot be represented with Weber's approach.

## C. ABE Implementations

There are multiple implementations for ABE. Since one of the goals is to write a prototype application for a mobile Android device, it is necessary to check if these implementations can be run there. CP-ABE [2] was the first implementation of an ABE scheme. It was written by John Bethencourt in 2007. For its calculations the pairing-based cryptography library [12] is used. While it only supports CP-ABE, it features a parser to transform the Boolean formulas into an internal format. This implementation also has support for numerical attributes in the private key and their comparison operators in the encryption policies. Besides supporting $and$ and $or$ CP-ABE also supports the use of an $of$ operator.

Libfenc [13] is a library for functional encryption schemes. It has been written in 2010 and 2011 by Akinyele et al. and supports both CP-ABE and KP-ABE. Since it reuses the code of the parser from Bethencourt's CP-ABE implementation, it also support numerical attributes. This library is also written in C/C++. Development on this library stopped in 2011.

With Charm [14], there is a follow-up project. The major drawback for this library is, that, while there is parser for policies, it does not support numerical attributes. Further, the parser does not handle the $of$ operator. The predicate-based encryption library (PEBEL) [15] builds upon Charm. It extends the functionality presented in Charm by implementing a hybrid encryption scheme, where ABE is used as the key encapsulation scheme. It also implements numerical attributes and the accompanying operators.

The Java version of CP-ABE has been written in 2013 by Junwei Wang [16]. Instead of the pairing-based cryptography library it uses a Java port (JPBC [17]) of it. This port has been completely written in Java, making this library a prime candidate to use, since it can be deployed on any machine running Java, including Android. The internal formats of attributes and policies are described in Section III.

## III. Design

This section contains the design decisions that explain, how *dynamic attributes* and *location attributes* in an ABE scheme are used.

### A. Encoding Location Information

One of our goals was to integrate *location attributes* into ABE. To do this, the location has to be encoded in some way. In the following, we list geocoding systems considered. They encode the latitude and longitude of a position into another format. Some of the beneficial properties of these formats will be explained.

The most adequate solution for encoding location information is Geohash [18]. In 2008, Gustavo Niemeyer published a website[1] describing his work on encoding geolocations called Geohash. It is used to transform GPS coordinates into a short string. Being an open system its algorithm has been granting into the public domain. As such it is one of the most used systems for the encoding of locations. Geohashes are encoded in base 32, with the alphabet consisting of twenty-six letters from a-z and six digits from 2-7. Internal bits are used to specify the locations. Each letter describes the state of 5 bits. The Geohash of the location 52.51 latitude and 13.32 longitude with 6-character precision is: *u336xp*. The Geohash itself does not specify a point but an area, i.e., a square or a rectangle, respectively.

How the encoding and decoding works is best explained with an example. Let's say we wanted to encode the location that is at 52.51 latitude and 13.32 longitude. Because longitude and latitude are encoded in the same way, only the calculation for the latitude (52.51) will be shown in the example, see Table I. For the first bit the range of the latitude is set to the maximum (90, 90). Then depending on the latitude being greater than the mid of both ranges, the first resulting bit is set. The bit is only set when the latitude is greater than the mid of both values and not set otherwise. The range for the next iteration of the algorithm has to be adjusted. If the bit has been set, the latitude has been greater than the mid and all values below that are being discarded. If the bit has not been set, the latitude is smaller and the mid and all values greater are being discarded. This is done by replacing the value that is to be discarded with the middle of the previous iteration.

TABLE I: Example of how to encode latitude into a Geohash

| # Bit | latitude min | latitude mid | latitude max | bit result |
|---|---|---|---|---|
| 1 | -90 | 0 | 90 | 1 |
| 2 | 0 | 45 | 90 | 1 |
| 3 | 45 | 67.5 | 90 | 0 |
| 4 | 45 | 56.25 | 67.5 | 0 |
| 5 | 45 | 50.625 | 56.25 | 1 |
| 6 | 50.625 | 53.4375 | 56.25 | 0 |
| ... | ... | ... | ... | ... |

The longitude and latitude part of the hashes are combined by alternating between the two. The 0th and all other even bits belong to the longitude, while the 1st and all other odd bits belong to the latitude. This is done, because shortening the resulting string or bit string still yields an area where the initial location is in, but with a lower precision. Each bit added to the bit string decreases the area that is inscribed into a Geohash by half. Adding two bits, one to the longitude and one to the latitude part, results in an area that is four times smaller than before. When removing bits, the same is also true, removing two bits increases the size of the area by the factor four. This also means that areas that are in proximity to each other geographically tend to have common prefixes of the resulting Geohashes. The Geohash of the Ernst-Reuter-Platz in Berlin is *u336xp* and Eberswalde in the north of Berlin is *u33u9d*. While they are about 50km apart, they both share the common prefix: u33. So at least the first 15 bits of their Geohash are the same. Sometimes, locations close to each other do not share a common prefix. This happens at the borders of a Geohash, e.g., the meridian and the 180th meridian. However, since it is relatively easy to calculate the neighboring Geohashes, these can also be used to determine vicinity.

### B. Dynamic ABE Attributes

*Dynamic attributes* are attributes that can be added to a *private key* after it has been created. This was not intended to

[1]http://geohash.org/

be possible. The naive approach to adding *dynamic attributes* is to regenerate the *private keys* every time an attribute needs to be added. This would be relatively easy to implement, but it would require that there is a database that contains all the users and their information. This is because when a *private key* is created again, the *attributes* that would need to be created for the users need to be saved somewhere. This approach would also require some kind of authentication since one would need to know which user is requesting the new *private key*. The following approach seeks to overcome these constraints. Every *private key* has a unique value $D$ and a *set of attributes*. This unique value is the part of the key that prevents collusion attacks. Using the value $D$ and the *secret master key* from which the *private key* was generated it is possible to reverse some of the computations and calculate the value used for the generation of attributes ($g^r$). This value makes it possible to generate additional *attributes*. During the setup stage of Bethencourt's ABE scheme [2] a *public key PUK* is generated with the values: PUK $= \left(\mathbb{G}_0, g, h = g^\beta, f = g^{1/\beta}, e(g,g)^\alpha\right)$. Where $\mathbb{G}_0$ is a bilinear group. The exponents $\alpha$ and $\beta$ are random. The *master key MK* is also generated and has the value: MK $= (\beta, g^\alpha)$. During the *key generation* stage of the scheme a *private key PRK* gets assigned the value: PRK $= \left(D = g^{(\alpha+r)/\beta}, \forall j \in S : D_j = g^r \cdot H(j)^{r_j}, D'_j = g^{r_j}\right)$. Where $r$ and all $r_j$ are random and $H$ is a hash function. Since $r_j$ is random for each *attribute* and $g$ is known through the *public key*, only the value $g^r$ is needed to compute *new attributes*. Based on the value $D$ of the *private key*, the following rearrangement of the formula is possible:

$$D = g^{(\alpha+r)/\beta} \tag{1}$$

$$D^\beta = g^{\alpha+r} \tag{2}$$

$$g^r = D^\beta / g^\alpha \tag{3}$$

The resulting Equation (3) shows that only $D$, $\beta$ and $g^\alpha$ are required to calculate $g^r$. That means only the value $D$ of the *private key* and both values of the *master key* are required to compute *new attributes*. This approach minimizes the communication needed between the owner of the *private key* and the *key authority*. If the *key authority* has the *master key* only the $D$ value of the *private key*, and additional data for the generation of the *attributes*, such as *location information*, needs to be transferred. Generating a key with hundreds or even thousands of attributes can take a lot of time. With this approach only the *added attributes* need to be computed, the already *existing attributes* in the *private key* can be used without any alteration and are not touched upon or modified. This system is designed to run without any external authentication. The general assumption is that *generated attributes* are considered to be publicly available. Users can easily create a new *private key* with the generated *attributes* itself. For example, a file is encrypted with the *policy currentTime < 1451602800*, which specifies that the file can only be decrypted until the beginning of 2016. The *attribute currentTime = <current unix time>* can be retrieved from the *key authority*. Because *currentTime* is the only *attribute* that is required to decrypt the file, anyone that has access to the *key authority* can create a new *private key* with the *generated attributes* from it. The file should thus be considered to be publicly accessible. However, when the file is encrypted with *currentTime < 1451602800 and employee* and the attribute *employee* cannot be generated by the *key authority*. In this way it is ensured that users who want to decrypt the file already had a *private key* beforehand. When encrypting a file that should not be publicly accessible, additional restrictions in the policy formula are needed to authorize the users who try to decrypt the file.

## C. Location Attributes

To understand how *location information* can be added as an *attribute*, it is explained how the *internal attributes* of the ABE implementation work.

*1) Internal Format:* The JCPABE library uses an internal format that has been adopted from Bethencourt's implementation [2]. There is only one *the $k$ of $n$* operator. The operator evaluates to true if at least *$k$ of the $n$* subformulas evaluate to true. The atomic formula in a *policy* only evaluates to true if the *attribute* with the same name is in the *private key*. The internal format uses the *reverse polish notation*. The operations that are described in Section II-A are implemented with this operator. The *and* operator in this format is emulated by setting the *$k$ to $n$*, requiring that *all* subformulas evaluate to true. The *or* operator is emulated by setting *$k$ to* 1, thus requiring that at least *one* of the subformulas evaluate to true.

*2) Geohash Solution:* To implement location attributes, we utilize Geohashes [18]. As discussed in Section III-A, they have multiple beneficial properties, such as a *binary representation*, the length determines the *accuracy*, and they share a *common prefix*.

These properties can be used in ABE. For the usage of Geohashes in a policy the following syntax has been chosen to specify a permitted location area: *<attributename>:<latitude>:<longitude>:<precision>*. The last part of this format specifies the precision of the Geohash with the number of bits used. The process of converting this string to the internal representation will be explained with an example. A user wants to allow a file to be decrypted in northern Poland and eastern Germany. He needs to enter the latitude and longitude of a location and a precision. In this case he choses Ernst-Reuter-Platz in Berlin, Germany as a starting point (latitude: 52.51, longitude: 13.32) and a precision of 10 bits. The resulting policy has the form: *location:52.51:13.32:10*. With the latitude, the longitude and the precision in bits a Geohash is created. The binary representation of the Geohash has the form: *1 1 0 1 0 0 0 0 1 1*.

For each bit in this sequence an internal attribute is created in the form: *<attributename>_geohash_<bitmarker>*. The bitmarker is just a string of *64 Xs* (an arbitrarily defined maximum precision for a Geohash) where the *X* at the position of the bit replaced with the value of the bit. The internal representation of the given area is displayed in Listing 1 (to fit the page, this example is given with 32 bits as the maximum precision). These *attributes* are then combined with an *and*. The internal representation the *of* operator is used.

Listing 1: Internal representation of a Geohash policy

```
location_geohash_1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_x1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xx0xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxx0xxxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxxx0xxxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxxxx0xxxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxxxxx0xxxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxx
location_geohash_xxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxx
10of10
```

*3) Geohashes as an Attribute:* Since users also want to be able to decrypt a file that has been encrypted with a location attribute, the private key must contain Geohash attributes. A location attribute during the key generation has the following syntax: *<attributename>:<latitude>:<longitude>*. The precision does not need to be specified, caused by the property that longer Geohashes give a higher precision. The user location is saved as precise as possible (64 bits). 64 bits would be enough to specify specific square centimeters. The high precision has the consequence that each Geohash in a private key uses 64 attributes. If a key is generated with the *attribute location:54.3665:18.6335* (Gdansk, Poland) then the first 30 bits of the Geohash look like this: *1101000011 1100110011 0001000001*. For every bit of the 64 bits *one attribute* is created in the same way as in the policy. Note that both Geohashes have a *common prefix*; the first 10 bits of the private key location and the 10 bits of the Geohash in the policy are the same. Since the *location attribute* in the private key has a higher precision than the area, which is guaranteed because the location attribute in a private key is always 64 bits and both Geohashes share a common prefix. The newly created private key can be used to decrypt files that have been encrypted with the example policy.

*4) Problems:* If users encrypt their files with a location, then the assumption may be, that every private key that has a location attribute of the vicinity may be able to decrypt the file. This is not necessarily true. In Figure 1, there is a marker just east of the central square. The marker represents the location that was entered to create the Geohash. The marker is on the edge of the Geohash, meaning that even locations just around of this marker will not have the same Geohash. This happens, because Geohashes operate on a fixed grid. It is not possible to arbitrarily move the rectangle. A solution to this problem could be to include all the *neighboring* Geohashes, as it is done in the prototype.

### D. Transformation Approach

Bethencourt's implementation of CPABE [2] contains support for *numerical attributes*. With the help of these *attributes* it is possible to add support for *location attributes* and corresponding areas in a *policy*. The reason why these *numerical attributes* can not be used directly, is that only the integers from 0 to $2^{64}-1$ can be represented. To build upon this existing feature, it needs to be possible to transform the *range* of the longitude and the latitude to the *range* of the *numerical attributes* with the condition that the comparison operators still works as expected. A *policy* to check if a user is in a given rectangle (*latitudeMin, longitudeMin, latitudeMax, longitudeMax*) can then be defined the following way: *(latitude $\leq$ latitudeMax and latitude $\geq$ latitudeMin) and (longitude $\leq$ longitudeMax and longitude $\geq$ longitudeMin)*. A possible transformation for the longitude would be:

$$\text{lon} : [-180, 180] \to \left[0, 2^{64} - 1\right] \cap \mathbb{N}$$
$$\text{lon}(x) = \left\lfloor \frac{x + 180}{360} \cdot \left(2^{64} - 1\right) \right\rfloor$$

For the latitude the same function could be used. But since the latitude only ranges from $-90°$ to $90°$ the result can be of a higher precision. Both transformations *are surjective*, since every value from 0 to $2^{64}-1$ can be the result. The functions *are not bijective*, because the cardinalities of the domain and the co-domain are not equal. That these transformations work with the existing operations of the *numerical attributes*, it needs to be shown that the following statements are true (apart from rounding errors):

$$a * b \Leftrightarrow \text{lon}(a) * \text{lon}(b), \text{ for each } * \in \{=, <, >, \geq, \leq\}$$
$$a * b \Leftrightarrow \text{lat}(a) * \text{lat}(b), \text{ for each } * \in \{=, <, >, \geq, \leq\}$$

The major advantage of this approach would be that areas are not limited to the grid of a Geohash and can be arbitrarily defined. This would make the neighboring Geohash solution obsolete, resulting in a reduced time needed for the encryption. During the key generation two *numerical attributes*, one for longitude and one for latitude, need to be created. Each *numerical attribute* is represented by about 70 *internal attributes*. This means for a *location attribute* in a *private key* about 140 *internal attributes* are needed, 76 more than the 64 attributes in the Geohash solution. The Geohash key generation should perform better.

## IV. Implementation

During the realization, three different components have been implemented: the JCPABE library, a server component named Attribute Authority (AA), and a mobile client application based on the Android platform.

### A. JCPABE

Our library implements the five basic functions described in Section II-A and the additional functionalities described in Section III. For the implementation of this library it has been decided to build upon the Java CPABE version. The most compelling argument for Java CPABE is its portability. Since every component has been written in pure Java 6, it can easily be run on Android-OS. We call our implementation JCPABE[2]. The following paragraphs will give an overview

[2]https://github.com/TU-Berlin-SNET/JCPABE

of the introduced additions and changes to the Java CPABE library.

*1) Policy Parser:* As explained in Section III, the C implementation that had been ported uses two different policy representations. All Boolean formulas are transformed into the internal format. Then this format, which is also human readable, is used to encrypt the file. The Java port of the CPABE implementation lacks this transformation. This is why we re-implemented the parsing capabilities of the C implementation in Java. It is now possible to traverse the parse tree and generate the internal policy that is used by the ABE part.

*2) Attribute Parser:* Adding more complex attributes such as numbers and locations also requires the parsing of these attributes when generating private keys. This is much simpler than parsing the policy, since the attributes neither have operations nor a certain order. Because of this simplicity, it was possible to implement the parsing with the help of *regular expressions*.

*3) Hashing of Attributes:* The origin implementation saved the name of the attribute in plaintext. This is a problem, because the name of an attribute may already disclose information to an attacker. For example, an attribute in a policy may be called "london". By analyzing the policy tree, it is now very easy to determine that this attribute is a location attribute. With plaintext attributes, it would also be possible to use the internal representation of the location attribute to determine which Geohash has been used to create these attributes in the first place. With this information, an attacker may either go to that location, or even try to spoof the location, to ultimately access a file which he should not have access to. All names of attributes in private keys and in the policy formulas of the encrypted files are saved after they have been hashed.

### B. Attribute Authority

The AA is an implementation of a key authority. It manages the public keys and the secret master keys. It is called AA because it is a web application that can distribute generated attributes. There are two separate parts to the AA. There is a REST API and a web application that serve as a Management Tool (MT). Both use the same SQLlite database to retrieve information.

*1) REST API:* The REST API is used for the communication with the client application. Its main functionality is to distribute generated attributes. It also has an index for the secret and public keys. In this system, there are two types of users: *authenticated* and *unauthenticated. Authenticated users* are associated with a username. An overview of the REST API methods:

Unauthenticated users:

- GET/connectiontest
- GET/publickeys
- GET/publickeys/<name of a key pair>
- GET/publickeys/<name of a key pair>/attributes
- POST/publickeys/<name of a key pair>/attributes

Authenticated users:

- GET/authtest
- GET/secretkeys
- GET/secretkeys/<name of a key pair>
- POST/secretkeys/<name of a key pair>
- DELETE/secretkeys/<name of a key pair>
- POST/privatekeys/<name of a key pair>

*2) Management Tool:* The MT is used to control everything related to the RESTful service. It is not meant to be used by an end user. The MT features user and key management. Users as well as master keys and their associated public keys can be created and deleted. It can be used to change passwords. The most important feature for the attribute generation however is the management of attributes for each key pair. This controls, which attributes users can generate through the AA's REST API. The *name* and the *type* of the attribute are needed for creation. Currently there are four *types* of attributes possible: static string, string, number, and location.

### C. Android Application

A proof of concept Android application has been developed. Since the application is tightly integrated into the architecture, nearly every action communicates with an AA. There are five entries in the main menu:

- **Attribute Authority** - Lists all registered AAs. It is possible to add or delete AAs. Adding an AA requires a name and a URL.
- **Master Keys** - Displays owned secret master keys, stored on AA level. Create and delete secret master keys.
- **Private Keys** - Shows all locally stored private keys. Create (with add ad-hoc or pre-defined attributes) and delete private keys.
- **Encrypt a File** - Set a public key, chose a file, create a policy (including area selection on a map); file is encrypted and stored locally.
- **Decrypt a File** - Set a private key and a decrypted a file. Requests new attributes from the AA.

## V. Evaluation

In this section, the initial goals are evaluated. The main goal was to increase the expressiveness of ABE by adding dynamic location attributes. A comparison between the two implemented solutions is also given. To show that the developed system works, a field test was conducted. Results of the performance tests are at the end of this section. Additional results can be found in a previous article [19].

### A. Expressiveness of ABE Policies

The expressiveness of the access control mechanism from ABE is one of the subjects of this paper. But how does ABE compare to other access-control mechanisms? In 1992, Ferraiolo and Kuhn published a paper about role-based access control (RBAC) [20]. In their paper, they describe an access control mechanism where users are given roles, and objects are only usable when having a certain role. ABE has greater expressiveness than RBAC, because every RBAC system can easily be expressed in ABE, i.e., every role is converted into an attribute. Users that have roles are given a private key with these attributes. Actions, files, and objects are secured with ABE with policies of the following form: *role1* or *role2* or *role3* or ... The only thing that is more difficult to translate is the concept of role hierarchies. Roles can be comprised of other roles. The easiest solution is to add each sub-role of a role as an attribute as well. Since ABE can express even more with the help of the *and* and *or* operators, the access control mechanism of ABE is superior to RBAC.

An advancement of RBAC is the eXtensible Access Control Markup Language (XACML) [21]. It is an attribute-based access control scheme, which means that it can permit or deny the authorization based on arbitrary contextual information. This can include: actions the user wants to perform, objects the user wants to perform the action on, the identity or the time or the location of a requester, and many more. With XACML, it is possible to express many more scenarios than can be expressed with ABE. The main differences include the negation of attributes and the usage of functions on the attributes. It is for example possible to express that a user should not have a certain attribute at all. In ABE, it is not possible to compare two numerical attributes, while this is possible with XACML.

The scenarios that can be expressed in ABE are manifold. ABE's expressiveness can even outperform certain access control models. However, in contrast to the compared systems in this section, ABE has the advantage that the attributes are verified on the client side of the application. Thus, some of the processes of the ABE system can be done without any communication.

### *B. Comparison of both Dynamic Location Attribute Approaches*

Two different solutions for dynamic location attributes have been implemented. One is a proposed solution by Weber [10], the other has been detailed in this paper. The differences between the two are best explained when evaluating both solutions using an application scenario. In this scenario, the electronic health records (EHR) of patients are saved in an encrypted file. The encrypted file is only accessible by doctors and nurses. The nurses additionally also have to be at the hospital to access the file. For both approaches, the private keys of both users are the same. The nurse has a *nurse attribute* and the doctor has a *doctor attribute*. For our solution, the ABE policy of the file would look like this: *doctor* or (*nurse* and *location*:hospital) where *location*:hospital is an geographic area that encompasses the hospital.

With Weber's approach, the ABE policy would have the following form: *doctor* or *nurse*. To decrypt a file with this approach, a private key with satisfying attributes is not enough. An LBE key is also required to recover the plaintext. For this, the device used by the doctors and nurses also need to send their location to a server. If the location is inside the area of the hospital, then the right LBE key is returned.

The first major difference between these two approaches is, that in Weber's approach, the doctors also need to be at the hospital to view the data. This is because the location information is used to get an LBE key that is combined with the symmetric key, requiring that every decryption also specifies the LBE key. With the location attribute inside the ABE policy this is not necessary. A doctor can access the file without requiring him to send a location or even communicate with an AA at all. However, in the case of a time critical medical emergency, Weber's approach can have an advantage. A nurse wants to decrypt the EHR. The location of the nurse's device is sent to a server which then decides in fractions of a second if the location is valid or not. The time needed for the decryption is mostly dependent on the mobile device the nurse uses. In this situation, our approach is inferior. When a nurse tries to access the file during an emergency, the nurse's device will send the location to the AA to generate the missing attributes.

And in the case of a location attribute, 64 internal attributes need to be generated. In time critical situations, Weber's approach seems to be better suited. However, our approach preserves the expressiveness of ABE, allowing location attributes to be only used dependent on the policy formula.

### *C. Field Test*

To prove that the developed system actually works, a field test has been conducted. Both of the implemented approaches have been tested. A file has been encrypted twice, once for each system.

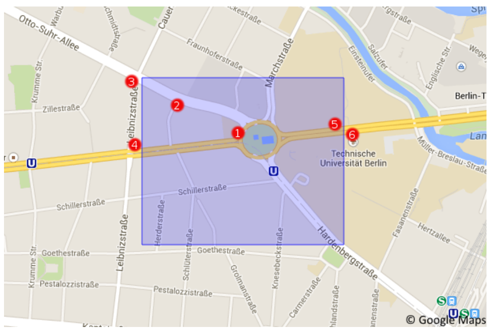

Fig. 1: Field testing positions / allowed area

For our approach, the file was encrypted with the policy: *attr and location:52.51304:13.32062:30*. The file that was tested with Weber's approach has been encrypted with the policy attribute. Additionally, an LBE key was entered during the encryption. The allowed area for the LBE implementation inside the AA has then been adjusted. For both approaches, only one private key has been used. The private key had the attribute *attr*. Both systems have been configured to only allow the decryption in the same area. The area can be described with the Geohash *u336xp* and is displayed in Figure 1. Also pictured in the same screenshot are the locations of the tests. At each location, an attempt to decrypt both files was made. The test has been conducted on a Nexus 4 Android device. Even being close to the edge of the allowed area still produced the correct results. This is due to the usage of GPS, which is the most accurate positioning mechanism the mobile phone supports. A major difference between the two systems is the time it takes to decrypt files.

With Weber's approach, the ABE decryption is significantly shorter, since only one attribute is needed to do that part of the decryption. With the other solution, at least 31 attributes are needed (one for the static attribute and at least 30 for the location attribute). The response times by the server are also much higher, since the AA needs to generate 64 attributes for the private key.

## D. Performance

To get an impression on how the number of location attributes is influencing the time to decrypt a file a performance test on a laptop with JCPABE was executed. The computer's attributes are: Intel CPU i5-2410M at 2.3GHz, 8GB RAM, Java-Version 8u45. The test run was performed with $2^0$ to $2^{10}$ location attributes. As Figure 2 is using a logarithmic representation on the x-axis, the linear results are not obvious at first sight. The different conjunctive location attributes were used in a disjunction. An application scenario for such a large amount of location attributes could be the usage of all medical practices locations within a city, e.g., Berlin.

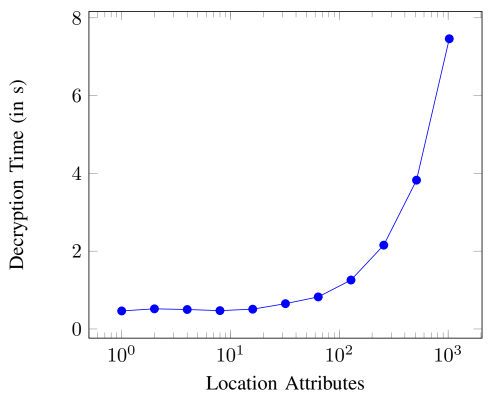

Fig. 2: Decryption time with increasing policy length (or)

## VI. Conclusion and Outlook

A system has been developed which users can use to securely share private data. In particular, users can also specify at which location the data is accessible. For this some additions to the existing ABE schemes have been designed. The contributions are the dynamic attributes and the proposed way to use location information integrated into an ABE scheme. A client application for the Android platform enables users to use these dynamic attributes for encryption and decryption of mobile cloud files. An already existing approach has been implemented and compared to the proposed system. During the work a second solution arose on how location information could be integrated into ABE. One problem with the way the dynamic attributes work is that a malicious user may accumulate attributes that are meant to be only used once. One possible solution would be to let ABE run in a trusted environment on the device. This environment could then make sure that additional attributes are discarded after a single use.

## Acknowledgment

The work presented in this paper was performed in the context of the *Curcuma*[3] project within the *Software Campus*[4] program. The project is funded by the German Federal Ministry of Education and Research (*BMBF*)[5].

[3]http://curcuma-project.net/

[4]http://www.softwarecampus.de/, (Förderkennzeichen 01IS12056), The authors are responsible for the content of this paper.

[5]http://www.bmbf.de

## References

[1] S. Zickau, M. Slawik, D. Thatmann, S. Uhlig, I. Denisow, and A. Küpper, "Tresor – towards the realization of a trusted cloud ecosystem," in *Trusted Cloud Computing*. Springer, 2014, pp. 141–157.

[2] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 321–334.

[3] S. Zickau, D. Thatmann, T. Ermakova, J. Repschlager, R. Zarnekow, and A. Kupper, "Enabling location-based policies in a healthcare cloud computing environment," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 333–338.

[4] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 89–98.

[5] S. Jahid and N. Borisov, "Piratte: Proxy-based immediate revocation of attribute-based encryption," *arXiv preprint arXiv:1208.4877*, 2012.

[6] A. Sahai, H. Seyalioglu, and B. Waters, "Dynamic credentials and ciphertext delegation for attribute-based encryption," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 199–217.

[7] R. Cramer and V. Shoup, "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003.

[8] S. G. Weber, "Securing first response coordination with dynamic attribute-based encryption," in *Privacy, Security, Trust and the Management of e-Business, 2009. CONGRESS'09. World Congress on*. IEEE, 2009, pp. 58–69.

[9] N. Doshi and D. Jinwala, "Updating attribute in cp-abe: A new approach," *IJCA Proceedings on International Conference in Distributed Computing and Internet Technology 2013*, vol. ICDCIT, pp. 23–28, January 2013.

[10] S. G. Weber, "A hybrid attribute-based encryption technique supporting expressive policies and dynamic attributes," *Information Security Journal: A Global Perspective*, vol. 21, no. 6, pp. 297–305, 2012.

[11] L. Scott and D. E. Denning, "A location based encryption technique and some of its applications," 2003.

[12] B. Lynn, "On the implementation of pairing-based cryptosystems," Ph.D. dissertation, Stanford University, 2007.

[13] M. Green, J. A. Akinyele, and M. Rushanan. libfenc: The Functional Encryption library. [Online]. Available: http://code.google.com/p/libfenc

[14] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, "Charm: A framework for rapidly prototyping cryptosystems," *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 111–128, 2013.

[15] J. de Muijnck-Hughes. (2013) pyPebel. [Online]. Available: https://github.com/jfdm/pyPEBEL

[16] J. Wang. (2013) CPABE. [Online]. Available: https://github.com/junwei-wang/cpabe

[17] A. De Caro and V. Iovino, "jPBC: Java pairing based cryptography," in *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*. IEEE, 2011, pp. 850–855. [Online]. Available: http://gas.dia.unisa.it/projects/jpbc/

[18] G. Niemeyer. (2008) Geohash. [Online]. Available: http://geohash.org

[19] S. Zickau, F. Beierle, and I. Denisow, "Securing mobile cloud data with personalized attribute-based meta information," in *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2015, pp. 205–210.

[20] D. Ferraiolo and R. Kuhn, "Role-based access controls," *Proc. of 15th NIST-NSA National Computer Security Conference*, 1992.

[21] OASIS Standard. (2013) eXtensible Access Control Markup Language (XACML) Version 3.0. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html