Extensions to Ciphertext-Policy Attribute-Based Encryption to Support Distributed Environments

Daniel Servos,
Department of Computer Science, Lakehead University,
Thunder Bay ON P7B 5E1, Canada
E-mail : dservos@lakeheadu.ca

Sabah Mohammed*
Department of Computer Science, Lakehead University,
Thunder Bay ON P7B 5E1, Canada
E-mail : mohammed@lakeheadu.ca
*The author for correspondence

Jinan Fiaidhi
Department of Computer Science, Lakehead University,
Thunder Bay ON P7B 5E1, Canada
E-mail : jfiaidhi@lakeheadu.ca

Tai hoon KIM
Department of Computer Engineering, Glocal Campus,
Konkuk University, Korea
E-mail :taihoonn@kku.ac.kr

Abstract: We present several extensions to the ciphertext-policy attribute-based encryption (CP-ABE) scheme, first introduced by Bethencourt, et. al. (2007), to support operation in a distributed environment with multiple attribute authorities. Unlike other efforts in creating a multi-authority attribute-based encryption schemes our extensions allow for each authority to be designated a subset of attributes and work independently in creating user keys (i.e. a user need only contact a single authority for their secret key). Additionally, we show that the presented extensions have a minimal impact on performance compared to standard CP-ABE and that both the performance of CP-ABE and our extensions can be improved by computing parts of the user and authority keys in parallel. We also discuss the use of CP-ABE in protecting data confidentiality in public cloud environments.

Keywords: Attribute-Based Encryption, Distributed Computing, Cloud Computing, Cryptography, Access Control

Reference: to this paper should be made as follows: Servos D., Mohammed S., Fiaidhi J., and Kim T. H. `Extensions to Ciphertext-Policy Attribute-Based Encryption to Support Distributed Environments', Int J. Computer Applications in Technology, Vol. x, No. x, pp. xxx-xxx

Biographical notes: Daniel Servos is a Graduate Student with the Department of Computer Science, Lakehead University working on Cloud Computing security and dependability.

Sabah Mohammed is a Professor with the Department of Computer Science, Lakehead University, Ontario-Canada. Dr. Mohammed research is focused on Medical Informatics and Cloud Computing. He is also an Adjunct Professor with the University of Western Ontario-Canada.

Jinan Fiaidhi is a Professor and Graduate Coordinator with the Department of Computer Science, Lakehead University. She is also an Adjunct Research Professor with the University of Western Ontario. Her research interest is on Ubiquitous Learning.

Tai hoon Kim is a Professor with Department of Computer Engineering, Glocal Campus, Konkuk University, Korea. He is a vice-president of Science and Engineering Research Support soCiety. Dr. Kim research focus is on Security.

1 Introduction

The increasingly popular cloud computing paradigm brings new opportunities to reduce hardware, maintenance and network costs associated with the traditional infrastructure required to offer large scale internet based services or even smaller localized application and storage solutions. However, with the dynamic scalability, reduced risk and potential cost savings comes a loss of control that creates new challenges for adopting cloud based infrastructure.

In particular, in the case of the health care industry, the need for cost efficient and low maintenance Electronic Health Record (EHR) systems is clear (Urowitz et al., 2001). However, while frameworks and architectures exist for managing and scaling access to EHRs in the cloud (Itani, Kayssi and Chehab, 2009) (Chow et al., 2009), data privacy and enforcing access control policies which comply with local and global privacy laws are significant barriers blocking adoption of public cloud offerings.

Ensuring data privacy on a potentially untrustworthy public cloud is still one of the open research problems in cloud computing (Mohammed, Servos and Fiaidhi, 2010) (Mohammed, Servos and Fiaidhi, 2011) Traditional data privacy measures (such as employing traditional symmetric encryption schemes) are ineffective on the cloud platform, while current research efforts tend to focus on solutions requiring additional trusted computing or cryptographic coprocessors hardware (Zhang, Cheng, and Boutaba, 2010) (Armbrust, et al. 2009) not yet offered by many public cloud providers.

Attribute based encryption, and in particular ciphertext-policy attribute-based encryption (Bethencourt, Sahai, and Waters, 2007) offers the potential for access policies to be embedded in encrypted documents and enforced both on and off the cloud, independently of the system which stores the documents. This is accomplished by assigning users' sets of "attributes" (strings associated with bit encoded values) which describe their role in relation to the embedded access policy and encrypting documents not with a key but a policy that must be met for decryption of the document to occur (this process is further detailed in section 2).

This paper outlines a novel extension to ciphertext-policy attribute-based encryption (CP-ABE) for protecting records both on and off the cloud. Our extensions add support for multiple distributed attribute authorities, capable of generating user keys, which share some given subset of attributes for which they are authorized. We introduce a new hierarchal authorization data structure (section 3.1) for attribute authorities which dictate the private and shared set of constant and variable attributes a given authority will have permission to delegate to users. A new not equals operation (section 3.5) for CP-ABE is detailed and its implications described, performance improvements (section 4.3) are discussed and tested, and a means for creating policies based on a user's origin (to which attribute authority they belong) is detailed (section 3.6). We present an implementation of our scheme (section 4.1) and evaluate its performance against Bethencourt, et al.'s (2007) CP-ABE implementation. A security evaluation and discussion is also presented in section 4.4.

2 Ciphertext-Policy Attribute-Based Encryption

2.1 Description

Ciphertext-policy attribute-based encryption (CP-ABE) (Bethencourt, Sahai, and Waters, 2007) offers a novel encryption scheme which continues the work on attribute based encryption to enable a complex tree based access policy to be embedded in the ciphertext rather than the key as with Key-Policy Attribute-Based Encryption (KP-ABE) (Goyal et al., 2006). CP-ABE also introduces "variable attributes" which use a set of traditional attributes to represent a value that can be evaluated with more complex operations (including >, <, <. > and =). This allows for users to be assigned attributes (e.g. "IS_DOCTOR" or "AGE = 36") and documents to be encrypted with complex access policies based on these attributes such as "(IS_DOCTOR OR IS_LAB_TECH) AND AGE >= 18 OR (IS_SYSTEM_ADMIN AND USER_LEVEL > 6)".

For the cloud, and other instances where a remote system may not be trusted to enforce access control, CP-ABE offers a promising alterative. With access polices embedded in the cipher text, encrypted documents may be stored and replicated across multiple untrusted system, while keeping the plain text secure under the given policy. Also, unlike traditional public key or symmetric encryption, where a document may only be decrypted by a single key (either the user's private key or the symmetric key used to encrypt the document), CP-ABE offers an alternative where each user has their own secret key (based on the attributes they have been assigned) which can decrypt any document for which there attributes pass the embedded access policy.

CP-ABE accomplishes this through five cryptological functions, SETUP(), ENCRYPT(PK, M, A), KEY_GENERATION(MK, S), DECRYPT(PK, CT, SK) and DELEGATE(SK, S) (see appendix A for detailed definitions of each). Operation of the system proceeds as show in Figure 2.1:

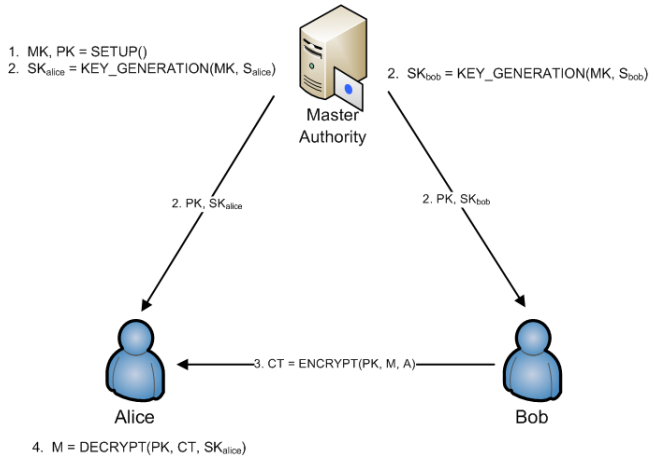

Figure 2.1: Operation of the CP-ABE cryptographic protocol.

1. During the initialization phase the SETUP function is called by the master authority to create the public (PK) and master (MK) keys. The public key is published publicly and the master key is kept in secret by the master authority and only used during user key generation.
2. After initialization, the master authority may generate a secret key (SK) for a user of the system using the KEY_GENERATION function which requires the master key and the set of attributes (S) assigned to the user by the authority and send the key to the user over a secure channel.
3. At any time after the public key has been published, a user may encrypt a document (M) for a given access policy (A) using the ENCRYPT function with the public key. The resulting ciphertext (CT) may be stored or shared with any system while remaining under the given access policy.
4. Once a user has a secret key, the public key and a cipher text they wish to decrypt they may do so using the DECRYPT function iff the set of the attributes in their secret key pass the access policy used during encryption.
5. Once a user has a secret key, they may delegate a subset of the attributes in their secret key to a third party using the DELEGATE function.

2.2 Issues for Distributed Systems

For distributed systems CP-ABE presents a significant bottleneck in the form of a single master authority responsible for all user key generation which proves problematic for both scalability and security reasons (i.e. if the master authority is compromised all encrypted documents are). Also it is required that a trusted party be made responsible for running the master authority and assigning all attributes to users of the system. In cases where a distributed system involves multiple parties (such as an EHR system which shares records between multiple health care institutions) a single party may not be available that is capable of securely, accurately and efficiently assigning attributes and generating user keys for all users.
Our extensions aim to resolve these issues by allowing each party (or domain) to have control over their own set of attributes, their own user key generation and attribute assignment while still sharing a common public key and set of attributes with the other domains making sharing of encrypted documents between domains possible. Additionally, this allows for a more scalable and trusted system as user key generation and attribute assignment is moved from a single point of failure and bottlenecking to multiple attribute authorities divided among the domains that comprise the system.

3 Our Extension's Constructions

Our construction extends the SETUP, DELEGATE and KEYGEN functions of the CP-ABE model to enable multi-authority user key assignment and delegation, while keeping the encryption (ENCRYPT) and decryption (DECRYPT) functions the same to enable backwards compatibility with CP-ABE. Each attribute authority is delegated a set of attributes for which it has authority over (power to further delegate the attributes to users) from an offline master authority. The master authority is only required during the initial creation of a new user authority or attribute.

The following sub-sections outline the extensions to each function. As with CP-ABE's constructions; G_0 is a bilinear group of prime order p and size k for which g is the generator of G_0 and $e: G_0 \times G_0 \rightarrow G_1$ denotes the bilinear map.

3.1 Authority Hierarchy

The authority hierarchy is a logical layout of all attribute authorities in the system and their parent/child relationships to each other. A child authority is granted all attributes of its parent and ancestors up to the root authority. Thus the root authority contains the subset of attributes shared by all authorities in the hierarchy. An example hierarchy is shown in Figure 3.1.

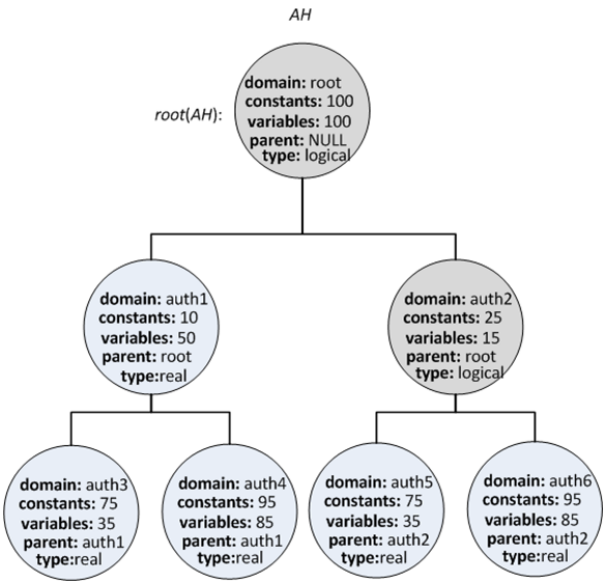

Figure 3.1: Example authority hierarchy with logical authorities root, and auth2, and real authorities auth1, auth3 .. auth6.

The authority hierarchy data structure can be seen as a tree (AH) with root node root(AH) representing the root authority. Decedents of the root node represent authorities authorized to assign both attributes assigned to their ancestors (up to the root authority/node) and attributes from a private set available only to that authority and it's decedents in the hierarchy. Each node, x, in the hierarchy tree contains a unique name referred to as a domain (domain(x)), the number of constant attributes (constants(x)) assigned to the authority, the number of variable attributes (variables(x)) assigned to the authority, the parent of the authority (parent(x)) and a type of "real" or "logical" (type(x)). Nodes of type "logical" are considered to only be place holders for attributes shared among their decedents. A logical authority is not granted an authority key and exists only in the representation of the hierarchy AS. For example, the authority "root" in Figure 3.1 will cause the creation of 100 constant attributes and 100 variable attributes which will be shared with all descendants but will not be issued an authority key. Authority "auth1" will be assigned 10 constant and 50 variable attributes and will inherit all 200 attributes from the root authority. Similar "auth3" and "auth4" will be assigned their designated number of attributes and inherit all attributes from "auth1" (their parent) and the root authority (auth1's parent). Like the root authority, auth2 is logical and will not be granted an authority key but will share it's attributes with its descendants auth5 and auth6 (which also inherit attributes from the root authority via auth2).

3.2 Setup

Our setup function (Equation 3.1) takes the authority hierarchy tree (AH) and begins as CP-ABE does, using the same master key (MK) and public (PK) definitions (only differing in excluding f from PK and referring to it as the delegation key) but adds the generation of the set of attribute authority keys (ASK) based on the attribute set returned by the AuthAttSet function (Equation 3.2) for each authority in the given hierarchy. As with Equation A.1 (see appendix A), $G_0$is a chosen bilinear group of prime order p and $\propto, \beta$ are randomly generated integers in $\mathbb{Z}_p$. We define auth_index(x) as a function which returns an arbitrary but always unique integer greater than 0 and less than INT_MAX for a given authority hierarchy node x.

$$
\begin{aligned}
&PK, MK, f, ASK = Setup(AH):\\
&\quad \text{choose: } G_0 \text{ of prime order } p \text{ with generator } g\\
&\quad \text{choose randomly: } \alpha, \beta \in \mathbb{Z}_p\\
&\quad PK = \left(G_0, g, h = g^{\beta}, e(g,g)^{\alpha}\right)\\
&\quad f = g^{\frac{1}{\beta}}\\
&\quad MK = (\beta, g^{\alpha})\\
&\quad AS = AuthAttSet(root(AH), \emptyset)\\
&\quad ASK = \forall S \in AS:\ ASK_s = KEYGEN(MK, S, PK)
\end{aligned}
$$

Equation 3.1: Setup Function

$$
\begin{aligned}
&AS = AuthAttSet(x, parentset):\\
&\quad FOR\ 1\,..\,constants(x)\ as\ i:\\
&\quad\quad S_i = string(domain(x) + \text{"\_c"} + \text{i})\\
&\quad FOR\ 1\,..\,(variables(x) * 2)\ by\ 2\ as\ k:\\
&\quad\quad S_{constants(x)+k} = string(domain(x) + \text{"\_v"} + i + \text{" = 0"})\\
&\quad\quad S_{constants(x)+k+1} = string(domain(x) + \text{"\_v"} + i + \text{" = "} + INT\_MAX)\\
&\quad S_{constants(x)+(variables(x)*2)+1} = string(\text{"auth\_key = "} + \text{auth\_index(x)})\\
&\quad S = ConvertAtts(S)\\
&\quad P = ConvertAtts(\{string(\text{auth\_key} = + \text{auth\_index(parent(x))})\})\\
&\quad S = S \cup (parentset \setminus \text{P})\\
&\quad IF\ type(x) = real:\\
&\quad\quad AS = \{S\} \cup \forall z\ child\ of\ x:\ AuthAttSet(z, S)\\
&\quad ELSE:\\
&\quad\quad AS = \forall z\ child\ of\ x:\ AuthAttSet(z, S)
\end{aligned}
$$

Equation 3.2: Recursive AuthAttSet Function

In addition to creating the master and public keys, the Setup function calls the recursive function AuthAttSet to obtain the set (AS) containing sets for each authority containing the attributes to be assigned to the respective authority. An authority's attribute set is determined by recursively descending the authority hierarchy and creating a set of attribute names for each node (with attribute sets for descendants of that node being unioned with the parent node). Next the auth_key attribute is added (explained further in section 3.6), the variable attributes are converted to constant attributes (via ConvertAtts(S)), the set is added to AS, and AuthAttSet is called on all children of the node. With the set of attribute sets complete, the Setup function is able to compute the set of secret keys (ASK) for each authority using the KEYGEN function.

As attributes may not be initially assigned a particular meaning or purpose in the system, a generic attribute name is created which may be later mapped to a more appropriate human readable name. Constant attribute names are created by appending the authority's domain, the string "_c" and a number (1 through constants(x) inclusively). Variable attributes are named similarly (by appending the domain, string "_v" and a number) but are also given the value of 0 and INT_MAX (INT_MAX being equal to 2b - 1 and b being the number of bits allowed in the attribute values).

To satisfy the policy tree during decryption, variable attributes are split into multiple constant attributes each representing a possible value of a single bit of the variable's value. Thus, assigning the same variable attribute to an authority with a value of 0 and INT_MAX is equivalent to assigning it the constant variable's for every possible value of a bit, making up the variable attribute's value. This allows an authority to assign any value for a variable attribute to a user (by delegating the subset of constant attributes which make up the correct value in bits for the delegated value) while only having to hold a key containing b * 2 constant attributes (where b is the number of bits in a variable's value). An example of this can be seen in Table 3.1 where an authority with the variable attributes

"auth1_v0 = 0" and "auth1_v0 = 15" also contains the subset of constant attributes for "auth1_v0 = 10".

Variable Attributes	Constant Attributes
auth1_v0 = 10	auth1_v0_ flexint_1xxx auth1_v0_ flexint_x0xx auth1_v0_ flexint_xx1x auth1_v0_ flexint_xxx0
auth1_v0 = 0 auth1_v0 = 15	auth1_v0_ flexint_1xxx auth1_v0_ flexint_0xxx auth1_v0_ flexint_x0xx auth1_v0_ flexint_x1xx auth1_v0_ flexint_xx1x auth1_v0_ flexint_xx0x auth1_v0_ flexint_xxx0 auth1_v0_ flexint_xxx1

Table 3.1: Table showing the equivalent constant attributes for a given set of variable attributes. Assuming INT_MAX of 15 (i.e. 4 bit variable values).

3.3 User Key Generation

Unlike in CP-ABE, a user's key is not generated via the KEYGEN function but delegated off an attribute authority's key. This process (detailed in Equation 3.3, where function H is a hash function, $\tilde{r}$, and $\tilde{r}_k$ are random numbers and US is the set of attributes to be assigned to the user) is the same as the DELEGATE function from CP-ABE but includes the delegation key, f, as it is omitted from the public key due to the changes in the Setup function. Also, unlike attribute delegation in CP-ABE, where a key owner may only delegate the value of an attribute variable for which they were assigned, an attribute authority is able to assign any value of for an assigned variable attribute. This is made possible due to the way variable attributes are assigned in our Setup function (i.e. via the attribute authority being assigned constant attributes for all possible values of bits in a variable attribute's value).

$\mathrm{USK} = \mathrm{UserKeyGen}(\mathrm{ASK_i}, \mathrm{US}, \mathrm{PK}, \mathrm{f})$:

choose randomly: $\tilde{r} \in \mathbb{Z}_p$

$$\widetilde{D} = Df^{\tilde{r}}$$

FOR $\forall k \in \mathrm{US}$:

choose randomly: $\tilde{r}_k \in \mathbb{Z}_p$

$$\widetilde{D''}_k = D_k g^{\tilde{r}} H(k)^{\tilde{r}_k}$$

$$\widetilde{D'}_k = D'_k g^{\tilde{r}_k}$$

$$\mathrm{USK} = (\widetilde{D}, \widetilde{D''}_k, \widetilde{D'}_k)$$

Equation 3.3: UserKeyGen Function

Further delegation of attributes at the user level is controlled by limiting access to the delegation key. A user with the delegation key may further delegate their attributes into a new key using the same UserKeyGen function with their key and a subset of attributes from the set they were assigned, but may never add more attributes, change attribute values or combine the attributes with another users. Allowing users to further delegate their attributes is no more insecure than the possibility of users sharing a key or the information which it decrypts but has the advantage of enabling users to share only parts of their key (some subset of their assigned attributes) when necessary.

3.4 Encryption and Decryption

Encryption and decryption proceed similar to CP-ABE (so much so that it is backwards compatible with CP-ABE) but with a key difference from the CP-ABE implementation. As one of the performance enhancements presented in the CP-ABE implementation, an additional constant attribute is added for each variable attribute containing the decimal value of the variable. For example, for the variable attribute "auth1_v0 = 10" the constant attribute "auth1_v0_10" would be added. This allows for a performance increase when the policy tree contains an equals requirement on a variable attribute (e.g. requiring that auth1_v0 being equal to 10). As this would require each authority being assigned a constant attribute for each possible value from 0 to INT_MAX (adding significantly to the time to generate and the size of an authority key) we have omitted this enhancement and a policy tree is created which requires all attributes that make up the value in binary to be present (e.g. requiring that a key contains attributes for auth_flexint_1xxx, auth_flexint_x0xx, flexint_xx1x and flexint_xxx0 rather than just the attribute auth1_v0_10 being present).

3.5 Not Equals

The CP-ABE scheme presented by Bethencourt, et al. (2007) lacks a not equals operation. Such an operation is important for at least two cases; the first being excluding a particular user based on an attribute that most or all users possess some value for. For example if we assign every user in the system the variable attribute "user_id" for some unique value to that user (e.g. "user_id = 4727236"), not equals would then allow us to exclude a particular user from decrypting a file by creating an access policy such as

"user_id ≠ 9833344". The second case is for user and authority key revocation. As with the first case, a policy like "user_id ≠ 9833344" could be added to all encrypted documents to block a known-to-be-compromised key from accessing future files. Similarly, a statement like "auth_key ≠ 2" could be used to revoke access to a whole authority's user base in the case that the given authority becomes compromised.

For our proposes we may define "not equals" as "equal to any valid value but", meaning that a user must both have the given variable attribute and it must not be equal to the given value to pass the access policy (users missing the attribute completely would also be rejected). We may construct such an access tree as follows (for the example of "user_id ≠ 4" and an INT_MAX of 15):

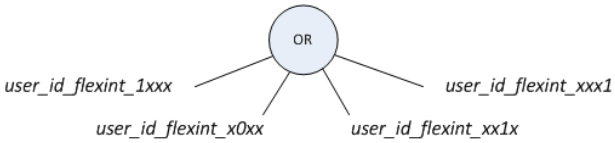

Figure 3.2: Access tree for user_id ≠ 4

That is essentially by converting the value to binary, "0100", inverting the 1s and 0s, "1011", and requiring the attribute for each bit. E.g. "1 of (user_id_flexint_1xxx, user_id_flexint_x0xx, user_id_flexint_xx1x, user_id_flexint_xxx1)".

3.6 User Origin

Our scheme provides a means of creating access policies based on a given user's origin (in this case to which attribute authority they belong) via the "auth_key" attribute added to every authority key in the AuthAttSet function (Equation). Every authority in the hierarchy is assigned a unique value of auth_key during authority key creation. It is expected that each attribute authority includes this attribute in each user's attribute set. However, even if a dishonest authority or user (via delegation of a subset of their key) omitted the attribute they would still fail to pass access policies requiring or excluding a given auth_key value (as the not equals operation requires the attribute be present and have at least some value).

As attribute authorities likely represent different institutions or departments in the system, this feature is useful for limiting access to or from a given institution/department when all other required attributes are shared and no existing attribute performs a similar role. Additionally, the "auth_key" attribute is used as part of our revocation system for revoking compromised authority keys (see section 3.7).

3.7 Revocation

3.7.1 User Key Revocation and Expiration

As with the CP-ABE scheme, user keys may be set to expire by including a variable attribute, an expiry date and ensuring that all encrypted documents contain a policy requiring that attribute to be greater than the current date. This would effectively limit expired user keys to decrypting documents encrypted before the key expired (which may already have been decrypted/compromised by the user). Revoking the user key then becomes simply a case of not renewing the key. Additionally in cases where users have no need to view documents past a set date, a lower limit may be placed on the expiry attribute to limit access to old documents.

As our scheme adds a not equals operation we are also given the option of revoking access to a user by excluding an attribute value unique to that user. For example, if all users are given a unique value for the variable attribute "user_id" one may simply exclude a given user by adding a policy such as "user_id ≠ 123456" to deny a user with the id 123456 the ability to decrypt the document. Even if the user removed the "user_id" attribute from their key, they would still fail to pass the access policy due to the way not equals is defined.

3.7.2 Attribute Authority Revocation and Expiration

The addition of the "auth_key" attribute in AuthAttSet function (Equation) allows us to deny access to users whose key was generated by a set authority (i.e. their origin, see section 3.6) as we could an individual user via a "user_id" attribute. If an attribute authority were to become compromised, a notice could be posted in a public revocation list and future documents could be encrypted with the requirement that "auth_key ≠ 5" for example, if the authority with an auth_index of 5 was compromised. This would effectively prevent any user from that authority from decrypting future documents.

As with user keys, attribute authority keys could also be created with a set expiry attribute which would in turn be delegated on to its users. However, unlike the user key expiry date, which may be set to only days or hours in the future an authority key would have to be set to expire significantly longer in the future (possibly months or a year) as the process for creating authority keys is more costly and involves some level of manual intervention by a system administrator.

4 Implementation and Evaluation

4.1 Implementation

To test and fully evaluate our extensions to CP-ABE, a C++ based implementation was created by modifying Bethencourt, et al.'s (2006) CP-ABE implementation, to add our extended functions, features and distributed authority setup. As our implementation is based on the CP-ABE implementation it uses the same PBC library (http://crypto.stanford.edu/pbc/) for the algebraic operations and only supports Unix and Linux based systems.

Our Setup (Equation 3.1), AuthAttSet (Equation 3.2) and UserKeyGen (Equation 3.3) functions were added to the CP-ABE implementation. The constant attribute added for each variable attribute containing the decimal value of the

variable was removed (see section 3.4) and the implementation was split into three components (one for the master authority, one for attribute authorities and one for the users of the system). A hash table was added to store all components of an authority key in memory (mapping an attribute name to the values of Dj and D`j ($g^r \cdot H(j)^{r_j}$ and g^{r_j} respectively)) for the attribute authority component. This allows for the authority key to be read into memory during the initialization of the attribute authority rather than read from the hard drive for each user key request. As the authority key grows linearly in size with the number of attributes (Figure 4.1) this becomes a required optimization for systems that must fulfill a large number of user key generation requests.

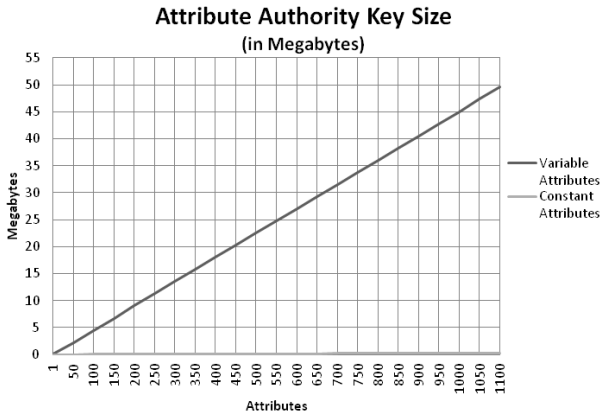

Figure 4.1: Attribute authority key size vs number of variable and constant attributes.

Finally, a Linux daemon was created for the attribute authority component which listens for connections on a local socket and responds to user key generation requests. A Java based client API was also created to communicate with attribute authority daemon as well as a Java API which uses the user component for encrypting and decrypting strings.

4.2 Performance Evaluation

To evaluate the performance of our extensions we examined our implementation in terms of number of constant attributes required to represent the same variable attribute, the time required to generate an authority key, the time required to generate a user key, and the size of the resulting user and authority keys. An unmodified version of Bethencourt, et al.'s (2006) CP-ABE implementation is used as a control and comparison for our results when possible.

Tests for the results in the following sections were performed on an Ubuntu Linux based system with the following specifications:

- **CPU:** Intel Core2 Quad CPU Q6700 @ 2.66GHz
- **RAM:** 4GB
- **Hard Drive:** 30GB
- **Network:** 10/100/1000Mbps

4.2.1 Attributes Required and Key Size

Unlike the keys used in CP-ABE, attribute authority keys require the attributes to create any possible value for a variable attribute. This leads the performance inequities between our extension and CP-ABE when the size of the authority keys, or the time required to generate authority keys, is compared. As shown in Figure 4.2, $b - 1$ more attributes are required for each variable attribute in an authority key than in an equivalent CP-ABE key (where b is the number of bits in a variable's value). However, the number of attributes required for a user key in our extension is one less than in CP-ABE.

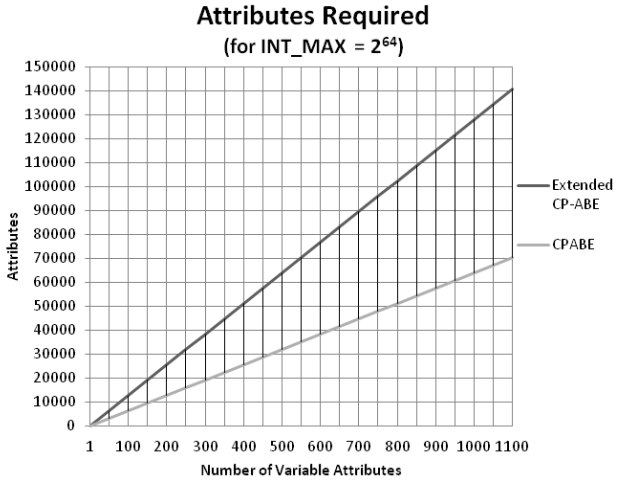

Figure 4.2: Constant attributes required to represent a given number of variable attributes in a authority key and a CP-ABE user key.

The number of attributes is also directly proportional to the size of the authority and user keys as shown in Figure 4.3. As with the number of attributes, despite the large size of the authority key, the size of a user key in extended CP-ABE is slightly smaller than the equivalent in CP-ABE.

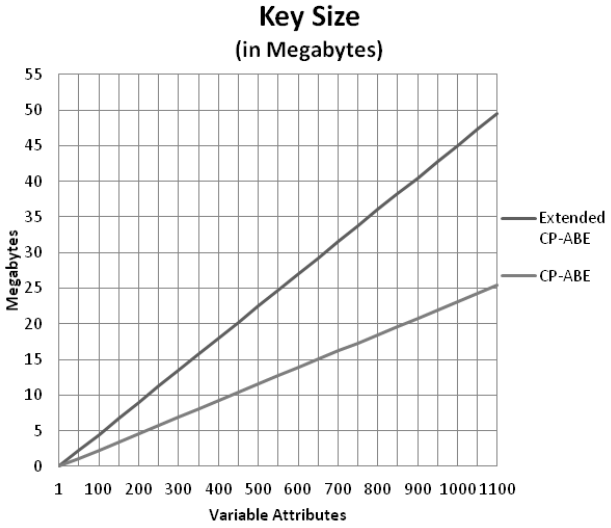

Figure 4.3: Size of an authority key vs. the size of a CP-ABE user key in megabytes for an INT_MAX of 264.

4.2.2 Time Required to Generate an Attribute Authority Key

The time in seconds to generate an authority key is shown in Figure 4.4. Authority key generation is linear with the number of attributes, though, still significantly longer than CP-ABE user key generation (Figure 4.5). However, the times are not easily compared as authority key generation is normally only performed once during the initialization of the master authority, while user key generation may be required frequently. As expected, the time required to create authority keys containing only constant attributes is significantly smaller than a key containing the same number of variable attributes as a variable attribute may be seen as b * 2 constant attributes.

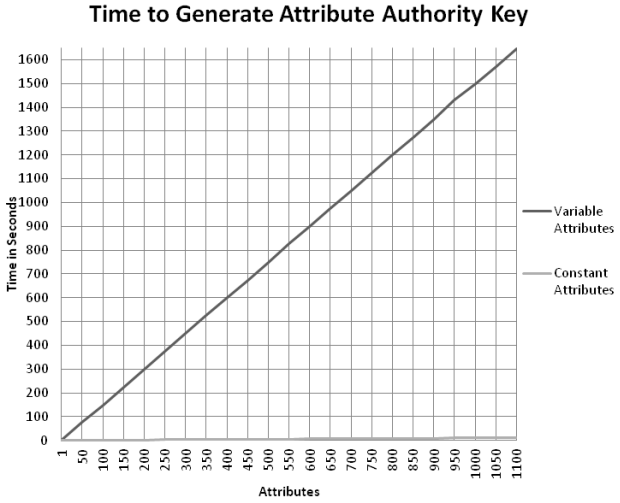

Figure 4.4: Time to generate an attribute authority key in seconds vs. number of variable and constant attribute.

4.2.3 Time Required to Generate an User Key

Figure 4.5 shows that the time to generate a CP-ABE user key is almost identical to the time required to generate a user key by delegation from an authority key via our UserKeyGen function. Additionally, Figure 4.6 shows that this will remain true despite the size of the authority key (assuming the number of attributes in the user key remains the same). Again the relationship between attributes and generation time remains linear.

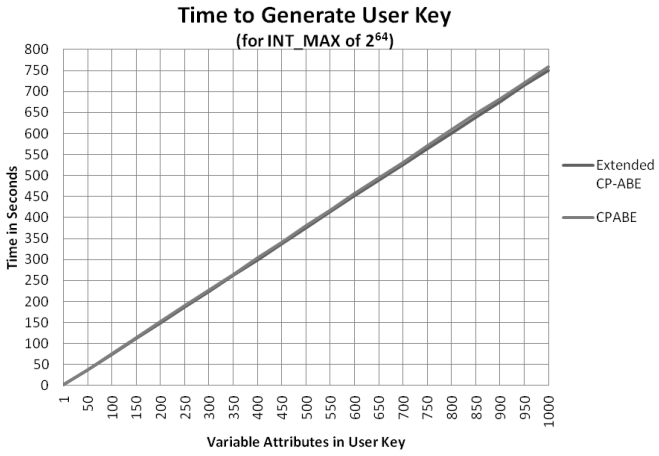

Figure 4.5: Time to generate user key in CP-ABE and Extended CP-ABE in seconds vs number of variable attributes.

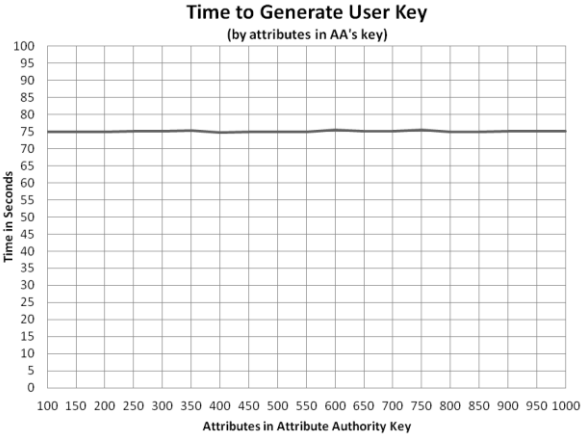

Figure 4.6: Time to generate an Extended CP-ABE user key in seconds vs number of attributes in the authority key for a constant number of attributes in the user key.

4.3 Potential Improvements

Our UserKeyGen function and the CP-ABE functions lend themselves well to parallel processing as there are many independent calculations required in computing the values of Di and D`i . The values of Di and D`i for each i in $\forall i \in$ US (UserKeyGen) or $\forall i \in$ S (KEYGEN) may be calculated independently and in parallel after the random value of r has been determined, so long as the proper order of the set is maintained. This allows for the generation of a particularly large attribute authority key to be set up as a massively parallel solution, reducing the generation time from hours to seconds. For user key delegation this allows for the use of multi core systems (which are becoming common place in both server and home hardware environments) to reduce Extended CP-ABE user key generation times to lower than the generation times in the standard CP-ABE implementation.

We present the following modifications (Equations 4.1, 4.2, 4.3 and 4.4) to our UserKeyGen function and the CP-ABE KEYGEN function to allow for parallel processing. For multi core, or multi CPU systems sharing the same memory, it is assumed that the area in memory is large enough to fit the resulting key that is created (e.g. an array of structures which will hold the value of Di and D`i) and the results are

placed correctly within the block of memory as computed. For distributed systems it is assumed that a central node will create a similar block of memory and store the resulting values correctly as computed. In both cases this should be a constant time, O(1), operation (as it is the same as inserting a value into an array). Additionally it is assumed that proper measures are put in place to properly produced secure random numbers in parallel.

$$
\begin{aligned}
&SK = KEYGEN\_PARALLEL(MK, S, PK):\\
&\quad randomize(r)\\
&\quad D = g^{(\alpha+r)/\beta}\\
&\quad B = Array\ Same\ Size\ as\ S\\
&\quad FOR\ \forall i \in S:\\
&Start\ Thread\ For\ keygen\_compute(B, r, g, i)\\
&\quad WaitForAllThreadsToFinish()\\
&\quad SK = (D, B)
\end{aligned}
$$

Equation 4.1: Parallelized version of the KEYGEN function.

$$
\begin{aligned}
&\text{keygen\_}compute(B, r, g, i):\\
&\quad randomize(k)\\
&\quad D''_i = g^r \cdot H(i)^k\\
&\quad D'_i = g^k\\
&\quad B_i = D''_i, D'_i
\end{aligned}
$$

Equation 4.2: keygen_compute function to be run in parallel.

$$
\begin{aligned}
&USK = UserKeyGen\_Parallel(ASK, US, PK, f):\\
&\quad randomize(\tilde{r})\\
&\quad \tilde{D} = Df^{\tilde{r}}\\
&\quad \tilde{B} = Array\ Same\ Size\ as\ US\\
&\quad FOR\ \forall i \in US:\\
&\quad Start\ Thread\ For\ userkeygen\_compute(\tilde{B}, \tilde{r}, g, i, D_i)\\
&\quad WaitForAllThreadsToFinish()\\
&\quad USK = (D, \tilde{B})
\end{aligned}
$$

Equation 4.3: Parallelized version of the UserKeyGen function.

$$
\begin{aligned}
&\text{userkeygen\_}compute(\tilde{B}, \tilde{r}, g, i, D_i):\\
&\quad randomize(k)\\
&\quad \widetilde{D''}_i = D_i \cdot g^{\tilde{r}} \cdot H(i)^k\\
&\quad \tilde{D}'_i = D'_i \cdot g^k\\
&\quad \tilde{B}_i = \widetilde{D''}_i, \tilde{D}'_i
\end{aligned}
$$

Equation 4.4: userkeygen_compute function to be run in parallel.

Using the same methodology and system as in section 4.2, we tested and compared the performance of the parallelized generation functions with both the standard Extended CP-ABE and CP-ABE key generation functions. Four simultaneous threads (each running on a separate CPU core) were used for the parallelized functions on the same Linux based system as used in section 4.2. The C++ POSIX threads API was used to provide threading functionality to our implementation. As shown in Figure 4.7 and 4.8, the parallelization of the key generation functions provides a signification improvement in the time required to generate both authority and user keys. This improvement allows key generation to be further scaled by adding additional CPU cores while maintaining a linear relationship with the number of attributes. It is likely that more modern systems with 6 or 8 CPU cores would show additional improvements in key generation time.

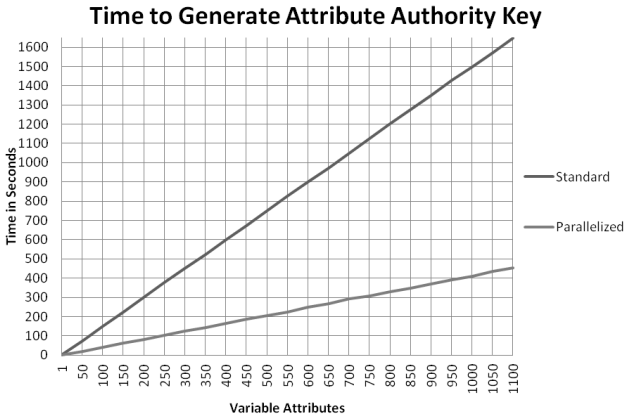

Figure 4.7: Time to generate authority key with standard and parallelized functions.

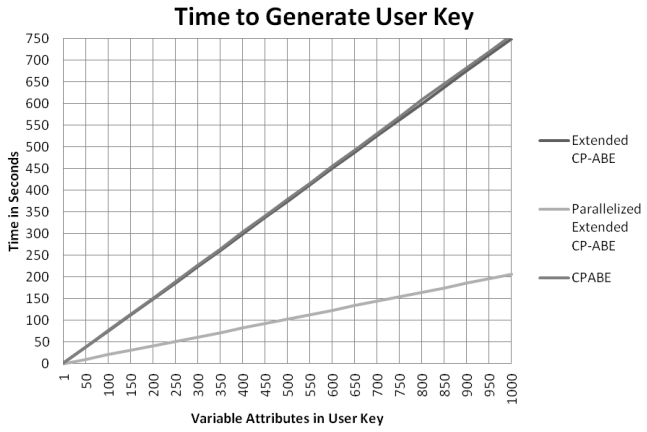

Figure 4.8: Time to generate a Extended CP-ABE user key with standard and parallelized functions (standard CP-ABE generation time also shown).

4.4 Security

As the encryption and decryption algorithms used in our extended CP-ABE scheme remain the same as presented in Bethencourt, et al. (2007), the security of extended CP-ABE may also be proven secure under the generic bilinear group model (Boneh, Boyen and Goh 2005) (as is shown in appendix A of (Bethencourt, Sahai, and Waters, 2007) for the decryption and encryption functions). As with Bethencourt, et al. (2007) and with Boneh and Franklin (2007)'s IBE schemes (Boneh, and Franklin, 2001) extended CP-ABE can be extended to be secure against a chosen ciphertext attack by applying the techniques from (Fujisaki and Okamoto, 1999).

5 Conclusions

This paper introduced extensions to Bethencourt, et. al.'s CP-ABE scheme for providing CP-ABE in a disturbed environment, in which multiple authorities are granted a set of constant and variable attributes which they may further

delegate to their users. Some subset of these attributes may be shared with other authorities such that access policies may be created that allow foreign users to decrypt documents. Additionally, a "not equals" operation was added to CP-ABE as well as a means for creating policies based on the user's origin (i.e. which attribute authority delegated their key). Details on how revocation and expiration are enforced where discussed.

The performance of a prototype implementation of our extensions based on Bethencourt, et. al.'s CP-ABE implementation was evaluated and found to scale linearly with the number of attributes. An additional improvement to the key generation and delegation algorithms to support distributed processing (e.g. on multiple CPU cores) was presented which further improved the performance of extended CP-ABE to the point of matching the original CP-ABE implementation.

Appendix

A. CP-ABE Functions from Bethencourt, et. al. (2007)

$PK, MK = SETUP()$:
choose: G_0 *of prime order p with generator g*
choose randomly: $\alpha, \beta \in \mathbb{Z}_p$
$PK = \left(G_0, g, h = g^{\beta}, e(g,g)^{\alpha}, f = g^{\frac{1}{\beta}}\right)$
$MK = (\beta, g^{\alpha})$

Equation A.1: Setup Function

$CT = ENCRYPT(PK, M, \tau)$:
chose randomly: $s \in \mathbb{Z}_p$
$q = CreatePolynomials(\tau_r, s)$
$Y = \forall leaf\ nodes \in \tau$
$CT = (\tau, \tilde{C} = Me(g,g)^{\alpha s}, C = h^s,$
$\forall y \in Y: C_y = g^{q_y(0)}, C'_y = H\big(att(y)\big)^{q_y(0)})$

Equation A.1: Encrypt Function

$q = CreatePolynomials(x, s)$:
"Starting with the root node [τ_r] the algorithm sets $q_r(0) = s$. Then, it chooses d_r other points of the polynomial q_r randomly to define it completely. For any other node x, it sets $q_x(0) = q_{parent}(x)(index(x))$ and chooses d_x other points randomly to completely define q_x."(Bethencourt, et al., 2007).

$SK = KEYGEN(MK, S, PK)$:
choose randomly: $r \in \mathbb{Z}_p$
$D = g^{(\alpha + r)/\beta}$
$FOR\ \forall j \in S$:
choose randomly: $r_j \in \mathbb{Z}_p$
$D''_j = g^r \cdot H(j)^{r_j}$
$D'_j = g^{r_j}$
$SK = (D, D'', D')$

Equation A.3: KeyGen Function

$M = DECRYPT(CT, SK, PK)$:
$A = DECRYPTNODE(CT, SK, PK, root(\tau))$
$IF\ A \neq \perp$:
$M = \frac{\tilde{C}}{\frac{e(C,D)}{A}}$
$ELSE$:
$M = \perp$

Equation A.4: Decryption Function

$A = DECRYPTNODE(CT, SK, PK, x)$:
$IF\ x\ is\ a\ leaf\ node$:
$i = att(x)$
$IF\ i \in S$:
$A = \frac{e(D''_i, C_x)}{e(D'_i, C'_x)}$
$ELSE$:
$A = \perp$
$ELSE$:

$\forall z\ child\ of\ x$: $F_z = DECRYPTNODE(CT, SK, PK, z)$
$S_x = \forall z\ child\ of\ x\ and\ F_z \neq \perp$
$IF\ S_x = \emptyset$:
$A = \perp$
$ELSE$:
$A = \prod_{z \in S_x} F_z^{\Delta_{i,s'_x}(0)} \quad where_{s'_x = \{index(z): z \in S_x\}}^{i = index(z)}$

Equation A.5: Recursive DecryptNode Function

References

Urowitz S., Wiljer D., Apatu E., Eysenbach G., DeLenardo C., Harth T., Pai H. and Leonard K. J., (2008) 'Is Canada ready for patient accessible electronic health records? A national scan', BMC Medical Informatics and Decision Making, vol. 8, no. 1, p. 33.

Mohammed S., Servos D. and Fiaidhi J., (2010) 'HCX: A Distributed OSGi Based Web Interaction System for Sharing Health Records in the Cloud,' in International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), IEEE/WIC/ACM, Toronto, ON.

Mohammed S., Servos D. and Fiaidhi J., (2011) 'Developing a Secure Distributed OSGi Cloud Computing Infrastructure for Sharing Health Records,' in AIS'11 Proceedings of the Second international conference on Autonomous and intelligent systems, Burnaby, BC.

Zhang Q., Cheng L. and Boutaba R., (2010) 'Cloud computing: state-of-the-art and research challenges,' Journal of Internet Services and Applications, pp. v. 1, i. 1, p. 7-18.

Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R., Konwinski A., Lee G., Patterson D., Rabkin A., Stoica I. and. A. M Zaharia, (2009) 'Above the Clouds: A Berkeley View of Cloud', UC Berkeley Reliable Adaptive Distributed Systems Laboratory, Berkeley.

Itani W., Kayssi A. and Chehab A., (2009) ' Privacy as a Service: Privacy-Aware Data Storage', in Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing,

Chengdu.

Chow R., Golle P., Jakobsson M., Masuoka R., Molina J., Shi E. and Staddon J., (2009) 'Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control', in Proceedings of the 2009 ACM workshop on Cloud computing security, Chicago.

Bethencourt J., Sahai A. and Waters B., (2007) 'Ciphertext-Policy Attribute-Based Encryption', in IEEE Symposium on Security and Privacy, 2007. SP '07. , Berkeley, CA.

Goyal V., Pandey O., Sahai A. and Waters B., (2006) 'Attribute-based encryption for fine-grained access control of encrypted data', in Proceedings of the 13th ACM conference on Computer and communications security (CCS '06), Alexandria.

Boneh D., Boyen X. and Goh E.J., (2005) 'Hierarchical identity based encryption with constant size ciphertext', EUROCRYPT, pp. volume 3494 of Lecture Notes in Computer Science, p. 440–456.

Boneh D. and Franklin M., (2001) 'Identity-Based Encryption from the Weil Pairing, Advances in Cryptology'.

Fujisaki E. and Okamoto T., (1999) 'Secure integration of asymmetric and symmetric encryption schemes', in CRYPTO '99 Proceedings of the 19th Annual International Cryptology, California.