Crypt-DAC: Cryptographically Enforced Dynamic Access Control in the Cloud

Saiyu Qi *State Key Laboratory of Integrated Service Networks (ISN)*
Xidian University, China
syqi@xidian.edu.cn
Yuanqing Zheng *Department of Computing*
The Hong Kong Polytechnic University
csyqzheng@comp.polyu.edu.hk

Abstract

Enabling cryptographically enforced access controls for data hosted in untrusted cloud is attractive for many users and organizations. However, designing efficient cryptographically enforced dynamic access control system in the cloud is still a challenging issue. In this paper, we propose Crypt-DAC, a system that provides practical cryptographic enforcement of dynamic access control. Crypt-DAC revokes access permissions by delegating the cloud to update encrypted data. In Crypt-DAC, a file is encrypted by a symmetric key list which records a file key and a sequence of revocation keys. In each revocation, a dedicated administrator sends a new revocation key to the cloud and requests it to encrypt the file with a new layer of encryption and update the encrypted key list accordingly. Crypt-DAC proposes three key techniques to constrain the size of key list and encryption layers as revocations continuously happen. As a result, Crypt-DAC enforces dynamic access control that provides *efficiency*, as it does not require expensive decryption/re-encryption and uploading/re-uploading of large data at the administrator side, and *security*, as it immediately revokes access permissions. We use formalization framework and system implementation to demonstrate the security and efficiency of our construction.

Index Terms

access control, cloud, revocation

I. Introduction

With the considerable advancements in cloud computing, users and organizations are finding it increasingly appealing to store and share data through cloud services. For instance, commercial cloud providers (such as Amazon, Microsoft, Apple, etc.) provide abundant cloud based services, ranging from small-scale personal services to large-scale industrial services. However, recent data breaches, such as releases of private photos [8], have raised concerns regarding the privacy of cloud-managed data. As such, a critical issue is how to enforce data access control on the potentially untrusted cloud.

In response to these security issues, numerous works [1]–[7] have been proposed to support access control on untrusted cloud services by leveraging cryptographic primitives. Advanced cryptographic primitives are applied for enforcing many access control paradigms. For example, attribute-based encryption (ABE) [9] is a cryptographic counterpart of attribute-based access control (ABAC) model [10]. However, previous works oversimplify revocation issues and mainly consider static scenarios in which access control policies are rarely changed. Besides, revocations can carry substantial overheads in practice. At a first glance, the revocation of a user's permission can be done by revoking his access to the keys with which the files are encrypted. This solution, however, is not secure as the user can just cache a local copy of the keys before the revocation. To prevent such a problem, files need to be re-encrypted with new keys. This requires the file owner to download the file, decrypt the file, and then re-encrypt the file before uploading back to the cloud, incurring prohibitive overhead at the file owner side.

Currently, only a few works investigated the problem of dynamic data access control. Garrison et al. [11] proposed two revocation schemes: immediate re-encryption (IMre), and deferred re-encryption (DEre). IMre requires an administrator to re-encrypt file with new keys in a revocation. This scheme incurs a considerable overhead as we discussed above. Differently, DEre delegates users to re-encrypt the file when they need to modify the file, relieving the administrator from re-encrypting file data by itself. This scheme, however, comes with a security penalty as the revocation operation is delayed to the next user's modification to the file. As a result, a newly revoked user can still access the file before the next writing operation. Wang et al. [23] proposed another revocation scheme: homomorphic re-encryption (HOre), in which the symmetric homomorphic encryption scheme [26] is used to encrypt the file. Such a design enables the cloud to directly re-encrypt file without decryption. However, HOre incurs expensive file read/write overhead as the encryption/decryption operation involves comparable overhead with the public key encryption schemes.

To overcome these problems, we present Crypt-DAC, a cryptographically enforced dynamic access control system on untrusted cloud. Crypt-DAC delegates the cloud to update encrypted file in permission revocations. In Crypt-DAC, a file is

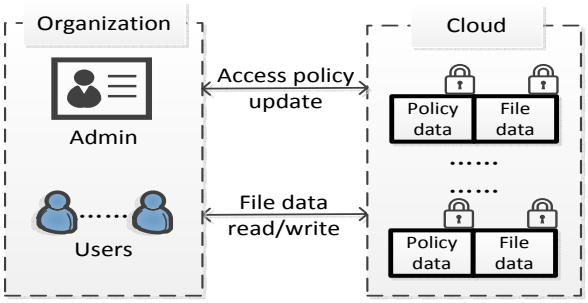


Fig. 1. Cloud enabled data access control.

encrypted by a symmetric key list which records a file key and a sequence of revocation keys. In a revocation, the administrator sends a new revocation key to the cloud, which encrypts the file with a new layer of encryption and updates the encrypted key list accordingly. Crypt-DAC is efficient as it avoids file transmission and mostly uses symmetric-key encryption. Crypt-DAC is also secure as it immediately revokes access permissions. Although the basic idea of layered encryption is simple, it entails tremendous technical challenges. For instance, the size of key list and encryption layers would increase as the number of revocation operations. To overcome such a problem, Crypt-DAC proposes three key techniques as follows.

First, Crypt-DAC proposes *delegation-aware encryption strategy* to delegate the cloud to update policy data. For a file, the administrator appends a new revocation key at the end of its key list and requests the cloud to update this key list in the policy data. The size of the key list however increases with the revocation operations, and a user has to download a large key list in each file access. To overcome this problem, we adopt the key rotation technique to compactly encrypt the key list in the policy data [16]. As a result, the size of the key list remains constant regardless of revocation operations.

Second, Crypt-DAC proposes *adjustable onion encryption strategy* to delegate the cloud to update file data. For a file, the administrator requests the cloud to encrypt the file with a new layer of encryption. Similarly, the size of the encryption layers increases with the revocation operations, and a user has to decrypt multiple times in each file access. To overcome this problem, we enable the administrator to define a tolerable bound for the file. Once the size of encryption layers reaches the bound, it can be made to not increase anymore by putting more trust on the cloud. As a result, the administrator can flexibly adjust a tolerable bound for each file (according to file type, access pattern, etc.) to achieve a balance between efficiency and security.

During the life cycle of a file, its encryption layers continuously increase until a pre-defined bound is reached. Crypt-DAC proposes *delayed de-onion encryption strategy* to periodically refresh the symmetric key list of the file and remove the bounded encryption layers over it through writing operations. In specific, the next user to write to the file encrypts the writing content by a new symmetric key list only containing a new file key, and updates the key list in the policy data. With this strategy, Crypt-DAC periodically removes the bounded encryption layers of files while amortizing the burden to a large number of writing users.

Altogether, Crypt-DAC achieves efficient revocation, efficient file access and immediate revocation simultaneously. We have implemented Crypt-DAC as well as several recent works [11], [23] on Alicloud. Real experiments suggest that Crypt-DAC is three orders of magnitude more efficient in communication in access revocation compared with IMre, and is nearly two orders of magnitude more efficient in computation in file access compared with HOre. Finally, Crypt-DAC is able to immediately revoke access permissions compared with DEre.

The remainder of this paper is organized as follows. In Section II, we introduce our system model and assumptions, background on $RBAC_0$ and the cryptographic techniques used in our system design. Section III identifies several critical issues for cryptographically enforced dynamic access control, from which we derive the principles of Crypt-DAC. Section IV describes the design details of Crypt-DAC. In Section V, we formally analyze the security of Crypt-DAC. In Section VI, we compare the performance of Crypt-DAC through experiments. In Section VII, we discuss related works. Section VIII details our conclusions.

II. Background and assumptions

In this section, we first define the system and threat models. We then define the classes of cryptographic primitives upon which Crypt-DAC is based.

A. System model

Our system model is depicted in Figure 1. We consider a scenario where companies contract with a commercial cloud provider (e.g., Alicloud, Microsoft Azure) to outsource enterprise storage. As a result, there are three types of entities in our system model: a cloud provider, an access control administrator, and a large number of users. The cloud provider is responsible for the data storage and management. The data includes file data of users in the company, as well as policy data regulating access policies for these files. Both the policy/file data are encrypted prior to being uploaded to the cloud provider. The access

control administrator is responsible for managing access policies of the file data. It assigns/revokes access permissions by creating, updating, and distributing cryptographic keys used to encrypt files. Users may download any policy/file data from the cloud, but are only allowed to decrypt and read files according to their access permissions. Finally, all parties communicate via pairwise secure channels (e.g., SSL/TLS tunnels).

B. Threat model

In our threat model, we consider that the administrator is honest. The users may be malicious and try to access the file data out of their access permissions. The cloud provider is assumed to be honest-but-curious. Honest means that the provider honestly follows the commands of the administrator/users. Since any errors due to the provider's misbehavior will harm its reputation, we believe that the provider has incentive to follow the commands required by its customers. However, the provider may be curious to passively gathering sensitive information to acquire commercial benefits and it is hard to be detected.

C. Security goals

We aim to provide confidentiality and access control for the cloud-hosted file data.

Confidentiality: our system stores encrypted data on the cloud, but never reveals the decryption keys to the cloud. This protects the confidentiality of the file data.

Read access control: our system uses cryptography to enforce access control so that users can only read file data according to their access permissions.

Write access control: for writing permission enforcement, our system relies on the cloud provider to validate write privileges of users prior to file updates.

D. Role based access control

We design and analyze Crypt-DAC based on the role-based access control model named (*RBAC*$_0$) [12], which is widely used in practical applications. *RBAC*$_0$ model describes permission management through the use of abstraction: roles describe the access permissions associated with a particular (class of) job function, users are assigned to the set of roles entailed by their job responsibilities, and a user is granted access to an object if they are assigned to a role that is permitted to access that object. More formally, the state of an *RBAC*$_0$ model can be described as follows:

– U: a set of users
– R: a set of roles
– P: a set of permissions (e.g., (*file*, *op*))
– $PA \subseteq R \times P$: a permission assignment relation
– $UR \subseteq U \times R$: a user assignment relation
– $auth(u, p) = \exists\, r$: $[(u, r) \in UR] \wedge [(r, p) \in PA]$: the authorization predicate *auth*: $U \times P \rightarrow \mathbb{B}$ determines whether user u has permission p

E. Cryptographic tools

Symmetric/public-key encryption scheme: Our construction makes use of symmetric-key encryption scheme (Gen^{Sym}, Enc^{Sym}, Dec^{Sym}) and public-key encryption scheme (Gen^{Pub}, Enc^{Pub}, Dec^{Pub}).

Key rotation scheme: Key rotation [16] is a scheme (Gen^{Ro}, *B-Dri*, *F-Dri*) in which a sequence of keys can be produced from an initial key and a secret key. Only the owner of the secret key can derive the next key in the sequence, but any user knowing a key in the sequence can derive all previous versions of the key. We next describe the algorithms of this scheme:

$Gen^{Ro}(1^n) \rightarrow (rsk, rpk)$: On input a security parameter 1^n, this algorithm outputs a secret-public key pair (rsk, rpk).

B-Dri$(k^i, rsk) \rightarrow (k^{i+1})$: On input a key k^i in a key sequence $\{k^i\}_{i=1}^t$ and a secret key rsk, this algorithm outputs the next key k^{i+1} in the sequence.

F-Dri$(k^i, rpk) \rightarrow (\{k^j\}_{j=1}^{i-1})$: On input a key k^i in a key sequence $\{k^i\}_{i=1}^t$ and a public key rpk, this algorithm outputs all previous versions of the key $\{k^j\}_{j=1}^{i-1}$ in the sequence.

III. Design principle

In this section, we begin with a basic construction of cryptographic access control enforcement, from which we derive a variety of issues for access revocation that must be addressed. We then give an overview of our system, Crypt-DAC, which addresses these issues.

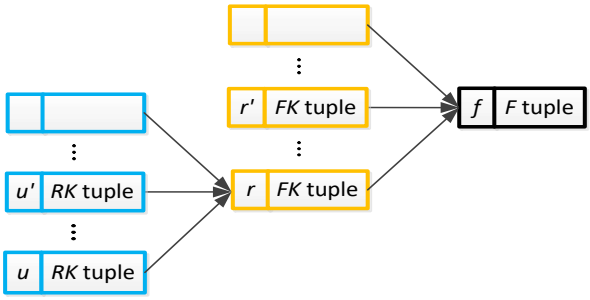


Fig. 2. Relations of *RK*, *FK*, and *F* tuples.

A. Basic construction

In a cryptographically enforced $RBAC_0$ system, a user u is associated with a user key $(ek_u, dk_u) \leftarrow Gen^{Pub}(1^n)$, a role r is associated with a role key $(ek_r, dk_r) \leftarrow Gen^{Pub}(1^n)$, and a file is associated with a file key $k \leftarrow Gen^{Sym}(1^n)$.

Access enforcement: For each $(u, r) \in UR$ in the $RBAC_0$ state (i.e., there exists a user u that is a member of r), the administrator distributes the role key of r to u through a role key (*RK*) tuple:

$$\langle RK, u, r, Enc^{Pub}_{ek_u}(dk_r)\rangle$$

This tuple provides u with cryptographically-enforced access to the decryption key dk_r of r. For each $(r, (f, op)) \in PA$ in the $RBAC_0$ state (i.e., there exists a role r with permission to f), the administrator distributes the file key of f to r through a file key (*FK*) tuple:

$$\langle FK, r, (f_n, op), Enc^{Pub}_{ek_r}(k)\rangle$$

This tuple provides r with cryptographically-enforced access to the file key k for f. For each file f, the administrator distributes f through a file (*F*) tuple:

$$\langle F, f_n, Enc^{Sym}_{k}(f)\rangle$$

This tuple contains a file name f_n of f and a ciphertext of f. The relations of *RK*, *FK*, and *F* tuples are shown in Figure 2.

File access: If a user u with authorization to read a file f (i.e., $\exists r$: $(u, r) \in UR \wedge (r, f, Read) \in PA$) wishes to do so, u downloads an *RK* tuple for the role r to decrypt the decryption key dk_r by dk_u. u also downloads an *FK* tuple for the file f to decrypt the file key k by dk_r. Finally, u downloads a *F* tuple to decrypt the file f by k.

If a user u with authorization to write to a file f via membership in role r (i.e., $\exists r$: $(u, r) \in UR \wedge (r, f, RW) \in PA$) wishes to do so, u uploads a new *F* tuple encrypting the new file content f' to replace the existing *F* tuple. It is the responsibility of the cloud provider to check if u has the authorization to write to f. If yes, the cloud provider executes the writing operation.

Access revocation: The administrator may need to revoke a permission of a role, or revoke a membership of a user. We only describe the user revocation case as the role revocation case is analogous. Removing a user u^* from a role r entails: (1) re-encrypting a new role key of r stored in *RK* tuples, (2) re-encrypting new file keys stored in *FK* tuples accessible by r, and (3) re-encrypting files stored in *F* tuples by the new file keys. All these steps must be carried out by an administrator as only the administrator can modify the access authorization.

In the first step, a new role key of r is generated. This key is then encrypted into a new *RK* tuple for each remaining member u of r to replace the old *RK* tuple. This step prevents u^* from accessing the new role key of r. In the second step, a new file key is generated for each file f accessible by r. This key is then encrypted into a new *FK* tuple for each role r' (including r) that has access to f, and the new *FK* tuple is uploaded to replace the old *FK* tuple. This step prevents u^* from accessing the new file keys. In the third step, each file to which r has access is re-encrypted into a new *F* tuple by the new file key. The new *F* tuple is then uploaded to replace the old *F* tuple. This step prevents u^* from accessing the files by using cached old file keys.

We emphasize that it is important to re-encrypt the files by the new file keys as u^* may cache the old file keys of these files and try to access them after the revocation. For example, suppose u^* is assigned to three roles and can access 200 files. u^* can first cache all the file keys of the 200 files when he join the system. Later, u^* is revoked from one of its three roles and cannot access some the 200 files anymore. However, u^* can still use the cached file keys to access these files if they are not re-encrypted by new file keys in the revocation.

B. Design issues for revocation

The revocation in this basic construction is not suitable for realistic dynamic access control scenario due to its prohibitive overhead in file data re-encryption. Due to the multi-to-one property of access permission relations among users, roles, and

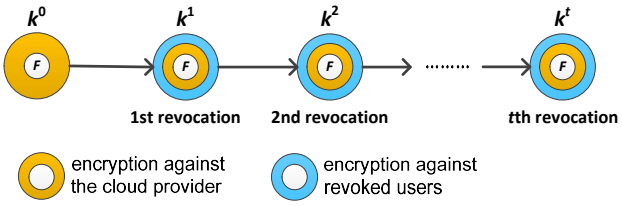


Fig. 3. Evolution of encryptions of a *F* tuple.

files in $RBAC_0$ model, the administrator needs to download, decrypt, re-encrypt, and upload a large number of *F* tuples, incurring potentially high bandwidth consumptions. For example, suppose the administrator needs to revoke the membership of u^* from *r* and *r* has permission to 100 files. Then, the administrator needs to re-encrypt all of the 100 files in the revocation.

Previous designs: In response, Garrison et al. [11] discussed two revocation schemes: immediate re-encryption (IMre) and deferred re-encryption (DEre). IMre requires the administrator to re-encrypt file data by itself in a revocation. This scheme completes the revocation immediately with a potentially high overhead as we discussed above. Differently, DEre relies on next users writing to the *F* tuples to re-encrypt the *F* tuples. This scheme, however, comes with security penalty as it delays the revocation to the next writing, creating a vulnerability window in which revoked users can continuously access the *F* tuples.

To alleviate the overhead of file data re-encryption, Wang et al. [23] proposed another revocation scheme: homomorphic re-encryption (HOre), in which the symmetric homomorphic encryption scheme [26] is used to encrypt the file data. Instead by re-encrypting the file data by itself, HOre enables the administrator to delegate the cloud to update *F* tuples from old file keys to new file keys without decryption. The problem of this scheme, however, is that the cost of homomorphic symmetric encryption is comparable with public key encryption schemes, incurring prohibitive computation overhead during file reading/writing.

C. Our design

To overcome these limitations, Crypt-DAC develops new techniques using lightweight symmetric encryption scheme. In Crypt-DAC, a *F* tuple (file) is encrypted by a symmetric key list $\{k^0, k^1,..., k^t\}$ which records a file key k^0 and a sequence of revocation keys $k^1,..., k^t$. Crypt-DAC uses the innermost encryption layer to protect the file against the cloud provider and the outermost encryption layer to protect the file against revoked users. In the *i*th revocation, the administrator sends a new revocation key k^{i+1} to the cloud, which encrypts the file with a new layer of encryption. After this procedure, the revoked user cannot access the file as he cannot access k^{i+1}. The evolution of the encryptions of a *F* tuple is shown in Figure 3. Compared with previous designs, Crypt-DAC achieves efficient revocation, immediate revocation, and efficient file access simultaneously. For revocation efficiency, the administrator only needs to send keys to the cloud. For immediate revocation, the permissions of users are immediately revoked as the files are re-encrypted. For file access efficiency, the files are still encrypted by symmetric keys. To further constrain the size of key list and encryption layers for files, Crypt-DAC proposes three key techniques as follows.

1) Delegation-aware encryption: Delegation-aware encryption enables the administrator to delegate the cloud provider to update *RK* and *FK* tuples instead of creating and uploading new *RK* and *FK* tuples by itself. To constrain the size of the key lists, delegation-aware encryption adopts the key rotation technique to compactly encrypt each key list in one encryption in a *FK* tuple [13]. As a result, the administrator only needs to send one encryption for the cloud to update a key list. This design also improves file access efficiency as users only need to download compact *FK* tuples to access files. The design principle of the strategy is shown in Figure 4.

To revoke a user u^* from a role *r*, the administrator generates a new role key (ek_r^*, dk_r^*) of *r*. The administrator first delegates the cloud provider to update *RK* tuples. Suppose there are *n* users remaining in *r*. For each of these users *u*, the administrator delegates the cloud provider to update the *RK* tuple of *u*. To do so, the administrator uploads an encryption of $Enc_{ek_u}^{Pub}(dk_r^*)$. With this encryption, the cloud updates the *RK* tuple as:

$$\langle RK,\ u,\ r,\ Enc_{ek_u}^{Pub}(dk_r^*)\rangle$$

The updated *RK* tuple encrypts the new role key of *r*.

The administrator next delegates the cloud provider to update *FK* tuples. An *FK* tuple contains an encryption of a symmetric key list $\{k^0, k^1,..., k^t\}$ for a file f_n. With the key rotation scheme, the administrator can use a key rotation key pair (rsk_{f_n}, rpk_{f_n}) to generate the revocation key sequence, and compactly represent this key list as $k^0||k^t$. Given $k^0||k^t$, the next revocation key k^{t+1} can be derived by $k^{t+1} \leftarrow B\text{-}Dri(k^t, rsk_{f_n})$ and the whole revocation key sequence $k^1,..., k^t$ can be recovered by $k^{i-1} \leftarrow F\text{-}Dri(k^i, rpk_{f_n})$ $(2 \leq i \leq t)$. As a result, the complete format of an *FK* tuple is:

$$\langle FK,\ r,\ (f_n,\ op),\ c\rangle$$

$$c = Enc_{ek_r}^{Pub}(k^0||k^t||rpk_{f_n}||t)$$

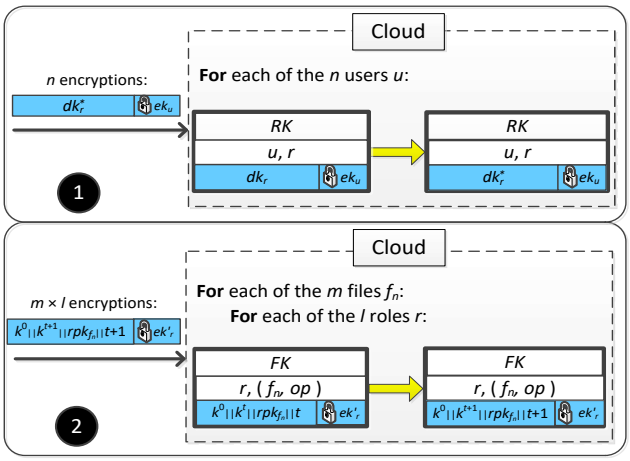


Fig. 4. Design principle of delegation-aware encryption.

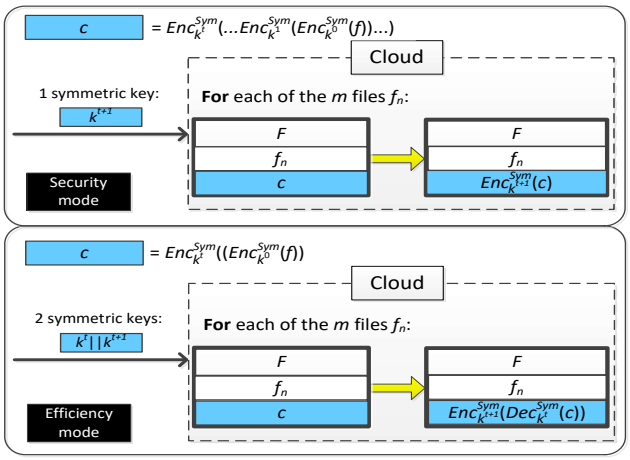


Fig. 5. Design principle of adjustable onion encryption.

Assume that there are m files to which r has permissions. For simplicity, we assume that there are n roles with permissions to each of the m files. For each of the m files f_n, the administrator generates a new revocation key $k^{t+1} \leftarrow B\text{-}Dri(k_t, rsk_{f_n})$. For each of the n roles with permissions to f_n, the administrator uses the role key of it to encrypt $k^0||k^{t+1}||rpk_{f_n}||t+1$ and sends the encryption to the cloud. Upon receiving the encryptions, the cloud provider updates the key lists in the FK tuples of the n roles as:

$$\langle FK,\ r',\ (f_n,\ op),\ c\rangle$$

$$\text{If } r' = r\text{: } c = Enc^{Pub}_{ek^*_r}(k^0||k^{t+1}||rpk_{f_n}||t+1)$$

$$\text{If } r' \neq r\text{: } c = Enc^{Pub}_{ek_{r'}}(k^0||k^{t+1}||rpk_{f_n}||t+1)$$

After the updating, the new FK tuples compactly encrypt the new symmetric key list $\{k^0,\ k^1,...,\ k^{t+1}\}$.

2) Adjustable onion encryption: Adjustable onion encryption enables the administrator to delegate the cloud provider to update F tuples. The administrator only needs to send a new revocation key for the cloud provider to encrypt the files with a new layer of encryption. To constrain the size of the encryption layers, adjustable onion encryption provides two modes: security mode and efficiency mode. Such a design enables the administrator to define a tolerable bound for a file. Initially, the strategy works in the security mode and the encryption layers increase as revocations happen. Once the size of the encryption layers reaches the bound, it turns to the efficiency mode to constrain the encryption layers by putting more trust on the cloud. As a result, the administrator can flexibly adjust a tolerable bound for each file according to file type, access pattern, etc., to achieve a balance between efficiency and security. The design principle of the strategy is shown in Figure 5.

Security mode: In this mode, a file f_n is encrypted in a F tuple by a symmetric key list $\{k^0,\ k^1,...,\ k^t\}$ as follows:

$$\langle F,\ f_n,\ c,\ sign_{r,v_r}\rangle$$

$$c = Enc^{Sym}_{k^t}(...Enc^{Sym}_{k^1}(Enc^{Sym}_{k^0}(f)))$$

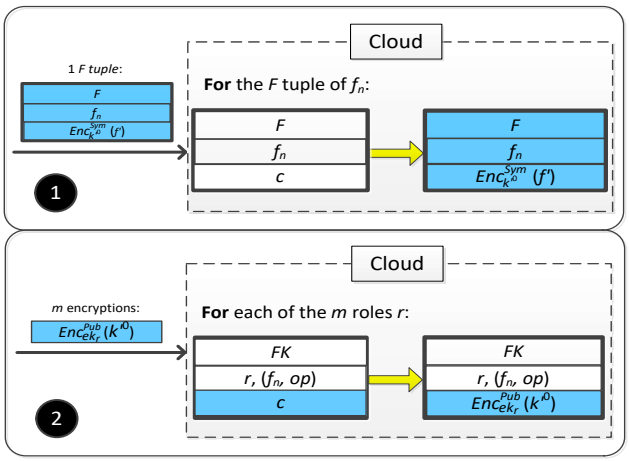


Fig. 6. Design principle of delayed de-onion encryption.

When a user u accesses f_n, u decrypts $k^0||k^t||rpk_{f_n}||t$ from one of these *FK* tuples, recovers the revocation key sequence: $k^{i-1} \leftarrow F\text{-}Dri(k^i, rpk_{f_n})$ $(2 \leq i \leq t)$, and uses $\{k^0, k^1, ..., k^t\}$ to decrypt the F tuple.

To complete the revocation of the user u^* from the role r, for each of the m files f_n to which r has permissions, the administrator derives a new revocation key $k^{t+1} \leftarrow B\text{-}Dri(k^t, rsk_{f_n})$ and sends k^{t+1} to the cloud provider. Upon receiving it, the cloud provider updates the F tuples of the m files as:

$$\langle F, f_n, Enc_{k^{t+1}}^{Sym}(c) \rangle$$

Efficiency mode: In this mode, a file f_n in a F tuple is encrypted by a symmetric key list $\{k^0, k^1, ..., k^t\}$ as follows:

$$\langle F, f_n, c, sign_{r,v_r} \rangle$$

$$c = Enc_{k^t}^{Sym}(Enc_{k^0}^{Sym}(f))$$

When a user u accesses f_n, u decrypts $k^0||k^t||rpk_{f_n}||t$ from one of these *FK* tuples and uses $\{k^0, k^t\}$ to decrypt the F tuple.

To complete the revocation of the user u^* from the role r, for each of the m files f_n to which r has permissions, the administrator derives a new revocation key $k^{t+1} \leftarrow B\text{-}Dri(k^t, rsk_{f_n})$ and sends $k^t||k^{t+1}$ to the cloud provider. Upon receiving it, the cloud provider updates the F tuples of the m files as:

$$\langle F, f_n, Enc_{k^{t+1}}^{Sym}(Dec_{k^t}^{Sym}(c)) \rangle$$

Comparing with the security mode, the efficiency mode encrypts an F tuple with two layers instead of t layers, enabling more efficient file access. The security tradeoff, however, is that the efficiency mode puts more trust on the cloud as it requires the cloud to correctly execute more operations (first decrypt and then encrypt) to update F tuples in each revocation.

Adjustable bound: The combination of the above two modes enables the administrator to define a bound T of tolerable encryption layers for a file. T means that when a file is encrypted by less than T encryption layers, the file access efficiency is tolerable. In this case, the administrator prefers to minimize the trust on the cloud and uses the security mode in revocations. Once the file is encrypted by T encryption layers, the administrator prefers file access efficiency and turns to use the efficiency mode in revocations by putting more trust on the cloud.

Considering a sequence of revocations happened in the life cycle of an F tuple. In the ith revocation, if $i \leq T$, the administrator uses the security mode to update the F tuple. To access the F tuple in this case, a user recovers a symmetric key list $\{k^0, k^1, ..., k^i\}$ from an *FK* tuple to decrypt the F tuple. Otherwise, the administrator turns to use the efficiency mode to update the F tuple. To access the F tuple in this case, a user recovers a symmetric key list $\{k^0, k^1, ..., k^{T-1}, k^i\}$ from an *FK* tuple to decrypt the F tuple.

3) Delayed de-onion encryption: During the life cycle of a file, its encryption layers continuously increase until a pre-defined bound is reached. To further improve file access efficiency, Crypt-DAC periodically refreshes the symmetric key list of the file and remove the bounded encryption layers. A direct solution is for the administrator to periodically re-encrypts the F tuple with a new key list which only contains a new file key. This solution however, incurs large communication and computation overhead at the administrator side. Instead, Crypt-DAC proposes delayed de-onion encryption strategy to do so through writing operations. In specific, a user writing to an F tuple encrypts the writing content by a new symmetric key list, and updates the symmetric key list accordingly. In this way, Crypt-DAC amortizes the updating burden to a large number of writing users. The design principle of the strategy is shown in Figure 6.

Suppose a user u wants to write to a file f_n. To do so, u generates a new file key $k'^0 \leftarrow Gen^{Sym}(1^n)$ and creates a F tuple encrypting the writing content f' by a new symmetric key list $\{k'^0\}$:

$$\langle F,\ f_n,\ Enc_{k'^0}^{Sym}(f')\rangle$$

u then requests the cloud provider to replace the existing *F* tuple with this *F* tuple. To alleviate the overhead of *u*, we propose to delegate the administrator to update the key list. In specific, *u* sends the new key list $\{k'^0\}$ to the administrator and delegates it to update the old key list stored in *FK* tuples. Assume that there are *m* roles having permissions to f_n. For each of the *m* roles *r*, the administrator encrypts k'^0 by the encryption key ek_r of *r*, and uploads the encryption for the cloud provider to update the *FK* tuple of *r* as:

$$\langle FK,\ r,\ (f_n,\ op),\ Enc_{ek_r}^{Pub}(k'^0)\rangle$$

After the writing operation, the symmetric key list is refreshed and the multiple encryption layers over the *F* tuple is removed.

IV. Design details

In Crypt-DAC, users add files to the cloud provider by creating *F* tuples. The administrator assigns file permissions to roles by distributing file keys using *FK* tuples, and assigns users to roles by distributing role keys to users using *RK* tuples. We next describe the various operations in Crypt-DAC. There are three types of operations in Crypt-DAC: permission revocation, permission assignment, and file operation. Due to the space constraint, we omit the algorithm details of these operations.

Permission revocation: Permission revocation includes revoking the permission of a user $\mathsf{revokeUser}(u)$ and revoking the permission of a role $\mathsf{revokeRole}(r)$.

The administrator uses $\mathsf{revokeUser}(u)$ to revoke the permission of a user *u* from all of its assigned roles. The algorithm implements the delegation-aware encryption strategy to update the involved *RK* and *FK* tuples respectively. The algorithm also implements the adjustable onion encryption strategy to update the involved *F* tuples. Moreover, $\mathsf{revokeUser}(u)$ provides a sub algorithm REVOKEU(*u*, *r*) for the administrator to revoke the membership of *u* from a certain role *r*.

The administrator uses $\mathsf{revokeRole}(r)$ to revoke the permission of a role *r* from all of its assigned files. Similarly, the algorithm also implements the delegation-aware encryption strategy to update the involved *FK* tuples and the adjustable onion encryption strategy to update the involved *F* tuples. $\mathsf{revokeRole}(r)$ also provides two sub algorithms REVOKEP(*r*, $\langle f_n$, *RW*$\rangle$) and REVOKEP(*r*, $\langle f_n$, *Read*$\rangle$) for the administrator to revoke the permission of *r* from a single file f_n.

Permission assignment: Permission assignment includes adding a new user $\mathsf{addUser}(u)$, adding a new role $\mathsf{addRole}(r)$, assigning a user to a role $\mathsf{assignU}(u, r)$, and assigning a role to a file with *Read*/*RW* permission $\mathsf{assignP}(r, \langle f_n, Read/RW\rangle)$. A user *u* uses $\mathsf{addUser}(u)$ to join the system. The administrator uses $\mathsf{addRole}(r)$ to add a new role *r* into the system, uses $\mathsf{assignU}(u, r)$ to assign a user *u* with a role *r*, and uses $\mathsf{assignP}(r, \langle f_n, Read/RW\rangle)$ to assign a role to a file with *Read*/*RW* permission.

File operation: File operation includes file creation $\mathsf{addFile}(f_n, f, u)$, file read $\mathsf{read}(f_n, u)$, and file write $\mathsf{write}(f_n, u)$. A user *u* uses $\mathsf{addFile}(f_n, f, u)$ to create a new file f_n in the system and uses $\mathsf{read}(f_n, u)$ to read a file f_n from the cloud provider. Also, a user *u* uses $\mathsf{write}(f_n, u)$ to write to a file f_n stored in the cloud provider. This algorithm implements the delayed de-onion encryption strategy to refresh the symmetric key list of f_n.

V. Security analysis

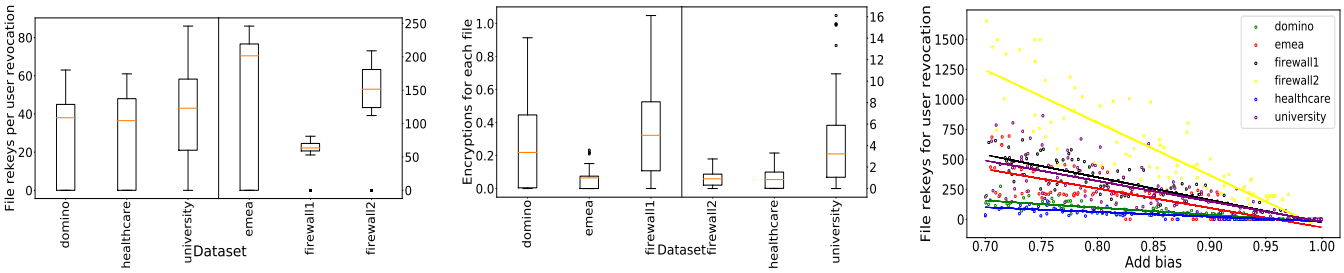
We analyze the security of Crypt-DAC using the access control expressiveness framework known as application-sensitive access control evaluation (ACE) [40]. ACE is a formalized mathematical framework to evaluate how well a candidate access control scheme implement an idealized access control scheme. We show that Crypt-DAC is correct and secure under ACE. In specific, we prove that Crypt-DAC satisfies three properties defined in ACE: *correctness*, *safety* and *AC-preservation*.

At a high level, *correctness* and *safety* ensures that an execution environment cannot determine whether it is interacting with the idealized $RBAC_0$ scheme or with Crypt-DAC through inputs, outputs and intermediate states. The two properties show that Crypt-DAC is correct. *AC-preservation* ensures that a permission in the idealized $RBAC_0$ scheme is authorized if and only if its mapping in Crypt-DAC is authorized. This property shows that Crypt-DAC is secure.

In our proof, we first formalize Crypt-DAC under the ACE framework. We then provide a formal mapping from Crypt-DAC to $RBAC_0$. We show that this mapping achieves *correctness*, *AC-preservation*, and *safety*. We provide the specification and proof online.

VI. System evaluation

We implement IMre, DEre, HOre, and Crypt-DAC on AliCloud. We compare the performance of the four systems in access revocation and file reading/writing. We implement the cryptographic schemes based on Crypto++ [41]. We select the AES scheme and the El Gamal scheme to instantiate the symmetric and public key encryption schemes respectively. Both the schemes are set with a security parameter of 80 bits. To evaluate the performance of the four systems in a realistic access control scenario, we use an access control simulation framework [42] to generate traces of access control actions. We extract several critical access control parameters from these traces to guide our experiments.



(a) File re-encryptions for a user revocation

(b) The average number of re-encryptions for each file

(c) File re-encryptions for user revocations

Fig. 7. Summary of simulation results.

A. Access trace simulation

The simulation framework proposed in [42] describes a range of realistic access control scenarios and allows us to investigate various access control actions and cryptographic operations incurred by these actions. We initialize the simulation from start states extracted from real-world datasets enforced by role based access controls. We then generate traces of access control actions using actor-specific continuous-time Markov chains. From these traces, we can record the number of instances of each cryptographic operation executed, including counts or averages (e.g., the average number of file re-encryptions needed to revoke a role from a user). We simulate one-month periods in which the administrator of the system behaves.

The real-world *RBAC* data sets used in the simulation are summarized in Table I. All of these datasets, aside from university, were originally provided by HP [43]. The domino dataset is from a Lotus Domino server, emea is from a set of Cisco firewalls, firewall1 and firewall2 are generated from network reachability analysis, and healthcare is a list of healthcare permissions from the US Veteran's Administration. The university dataset describes a university's access control system [44], [45].

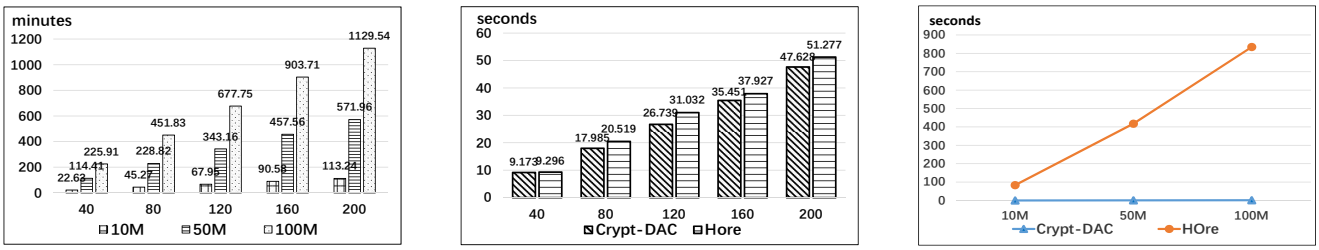
We show our simulation results in Figure 7. In Fig 7(a), we show the number of files that need to be re-encrypted for a single user revocation. In many scenarios, a user revocation triggers the re-encryptions of tens or hundreds of files, such as in emea or firewall2. From this result, we extract the following access control parameter: to revoke a user, the total number of involved F tuples that need to be re-encrypted in average falls in [0, 200].

Fig 7(b) shows the upper bound a file needs to be re-encrypted within a month for the purpose of user revocation. In Crypt-DAC, this bounds the increased number of encryption layers of a file within a month. From this result, we extract the following access control parameter: for file data, the encryption layers of a file in average increases less than 3 within a month.

Fig 7(c) shows the total number of file re-encryptions that take place over a month for the purpose of user revocation. *add bias* is a simulation parameter that describes the relative proportion of assignment vs. revocation operations. In many scenarios,

TABLE I
OVERVIEW OF THE DATASETS

set	**users**	**|P|/|R|**	**|UR|/|PA|**
domino	79	231/20	75/629
emea	35	3046/34	35/7211
firewall1	365	709/60	1130/3455
firewall2	325	590/10	325/1136
healthcare	46	46/13	55/359
university	493	56/16	495/202
roles-user max/min	**users-role max/min**	**perm-role max/min**	**roles-perm max/min**
3/0	30/1	209/1	10/1
1/1	2/1	554/9	31/1
14/0	174/1	617/1	25/1
1/1	222/1	590/6	8/1
5/1	17/1	45/7	12/1
2/1	288/1	40/2	12/1



(a) Performance of IMre at administrator side

(b) Performance of Crypt-DAC and HOre at administrator side

(c) Performance of Crypt-DAC and HOre at cloud side

Fig. 8. Performance in revocation.

we see several times as many re-encryptions as total files. This means that, on average, each file is re-encrypted multiple times per month for the purposes of user revocation. This result highlights the importance of revocation efficiency when designing an access control scheme.

B. End to end experiments

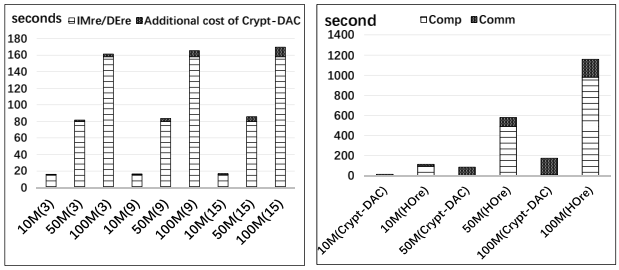
Revocation: We evaluate the performance of IMre, HOre, and Crypt-DAC in revoking a user u from a role r. We do not consider DEre as it incurs no overhead at the administrator side. This experiment is affected by two parameters: file size and number of F tuples that need to be re-encrypted. We consider three sizes of a file: 10 M, 50 M and 100 M. We use the access control parameter that the number of F tuples that need to be re-encrypted falls in [0, 200] and show the experiment results in Figure 8.

In Figure 8(a-b), we observe that the time cost of IMre at the administrator side is prohibitive and increases fast as the file size and the number of F tuples increases. When processing 200 F tuples with 100 M file size, IMre takes about 1129 minutes. On the other hand, HOre and Crypt-DAC costs about 50 seconds under the same parameters, achieving 1356 times improvement. Also, we observe that the time cost of HOre and Crypt-DAC is not affected by the file size. The reason is that the administrator only needs to generate and send cryptographic keys for F tuples regardless of their file size. Additionally, in Figure 8(c), we observe that the time cost of HOre at the cloud side is prohibitive and increases with the file size. The reason is that the cloud needs to homomorphically re-encrypt each F tuple, while in Crypt-DAC, the cloud only needs to AES-encrypt each F tuple. The time cost of HOre is 520 times higher than Crypt-DAC for a cloud to process a file regardless of the file size.

File reading/writing: We evaluate the performance of IMre, DEre, HOre, and Crypt-DAC in file reading/writing. The total execution time (including both computation and communication time) is affected by file size. We vary the file size from 10 M, 50 M to 100 M. For Crypt-DAC, it is also affected by the number of encryption layers of files. We use the access control parameter that the encryption layers of a file increases less than 3 within a month. Recall that once a file is written, the number of its encryption layers will be reset to 1 due to the delayed re-onion encryption strategy. We consider that a large fraction of files will be written at least once within every k months. For the remainder files, the administrator requests some users to write to them at the end of every k months. As a result, the maximum number of encryption layers of a file is $3k$ all the time. We evaluate the performance of Crypt-DAC in the worst case by encrypting files with the maximum number of encryption layers. We varies k (from 1 to 5) to investigate the performance trend of Crypt-DAC and predicate its performance when k is large enough (e.g., k = 12).

Figure 9 compares the execution time of reading operations. In Figure 9(a), we see that Crypt-DAC takes slightly more time compared with IMre and DEre (7.2% when k = 5). The reason is that Crypt-DAC uses AES scheme multiple times to encrypt/decrypt files. Fortunately, this additional cost increases very slow with the number of encryption layers because the AES operations are lightweight. According to our results, it can be predicted that the additional cost is about 15% even when k = 12 (a year). Moreover, we notice that a file will not be encrypted by the maximum number of encryption layers all the time. The actual number only increases when revocation happens and is reset to 1 once the file is written. For a file to be frequently revoked, we can further use the adjustable onion encryption strategy to bound the encryption layers of the file. Considering the performance advantage of Crypt-DAC in revocation, we believe that this extra cost is acceptable. In Figure 9(b), we see that HOre takes 80 computation times higher than Crypt-DAC (k = 5) and 6.7 total (computation and communication) times higher than Crypt-DAC (k = 5). This is because HOre uses homomorphic symmetric encryption to encrypt files, the overhead of which is comparable with public key encryption schemes. Instead, Crypt-DAC uses symmetric key encryption scheme to do so.

Table 1 compares the execution time of writing operations. We see that IMre, DEre and Crypt-DAC takes same time. The reason is that all the three schemes use symmetric key encryption scheme to encrypt files and the delayed de-onion encryption adopted in Crypt-DAC incurs no additional cost at user side. Besides, HOre takes 82 computation times higher than Crypt-DAC and 6.3 total (computation and communication) times higher than Crypt-DAC in average. This is because HOre uses



(a) Performance of IMre/DEre and Crypt-DAC at user side

(b) Performance of Crypt-DAC and HOre at user side

Fig. 9. Performance in file reading

homomorphic symmetric encryption to encrypt files, the overhead of which is comparable with public key encryption schemes. Instead, Crypt-DAC uses symmetric key encryption scheme to do so.

VII. Related work

Current cryptographically enforced access control schemes can be classified as follows.

Hierarchy access control: Gudes et al. [29] explore cryptography to enforce hierarchy access control without considering dynamic policy scenarios. Akl et al. [30] propose a key assignment scheme to simplify key management in hierarchical access control policy. Also, this work does not consider policy update issues. Later, Atallah et al. [31] propose a method that allows policy updates, but in the case of revocation, all descendants of the affected node in the access hierarchy must be updated, which involves high computation and communication overhead.

Role based access control: Ibraimi et al. [34] cryptographically support role based access control structure using mediated public encryption. However, their revocation operation relies on additional trusted infrastructure and an active entity to re-encrypt all affected files under the new policy. Similarly, Nali et al. [35] enforce role based access control structure using public-key cryptography, but requires a series of active security mediators. Ferrara et al. [36] define a secure model to formally prove the security of a cryptographically-enforced RBAC system. They further show that an ABE-based construction is secure under such model. However, their work focuses on theoretical analysis.

Attribute based access control: Pirretti et al. [37] propose an optimized ABE-based access control for distributed file systems and social networks, but their construction does not explicitly address the dynamic revocation. Sieve [23] is a attribute based access control system that allows users to selectively expose their private data to third web services. Sieve uses ABE to enforce attribute based access policies and homomorphic symmetric encryption [26] to encrypt data. With homomorphic symmetric encryption, a data owner can delegate revocation tasks to the cloud assured that the privacy of the data is preserved. This work however incurs prohibitive computation overhead since it adopts the homomorphic symmetric encryption to encrypt files.

Access matrix: GORAM [27] allows a data owner to enforce an access matrix for a list of authorized users and provides strong data privacy in two folds. First, user access patterns are hidden from the cloud by using ORAM techniques [28]. Second, policy attributes are hidden from the cloud by using attribute-hiding predicate encryption [21], [22]. The cryptographic algorithms, however, incur additional performance overhead in data communication, encryption and decryption. Also, GORAM does not support dynamic policy update. Over-encryption [38], [39] is a cryptographical method to enforce an access matrix on outsourced data. Over-encryption uses double encryption to enforce the whole access matrix. As a result, the administrator has to rely on the cloud to run complex algorithms over the matrix to update access policy, assuming a high level of trust on the cloud.

VIII. Conclusions

We presented Crypt-DAC, a system that provides practical cryptographic enforcement of dynamic access control in the potentially untrusted cloud provider. Crypt-DAC meets its goals using three techniques. In particular, we propose to delegate

TABLE II
Performance in file writing

File size	Others	HOre	Cost of HOre
10 M	1.1/16.8 Sec	98.6/17 Sec	88.3/6.4 times
50 M	6.1/84.9 Sec	490.4/89.1 Sec	79.5/6.3 times
100 M	12.3/168 Sec	981.7/175.1 Sec	79.8/6.4 times

the cloud to update the policy data in a privacy-preserving manner using a delegation-aware encryption strategy. We propose to avoid the expensive re-encryptions of file data at the administrator side using a adjustable onion encryption strategy. In addition, we propose a delayed de-onion encryption strategy to avoid the file reading overhead. The theoretical analysis and the performance evaluation show that Crypt-DAC achieves orders of magnitude higher efficiency in access revocations while ensuring the same security properties compared with state-of-the-art schemes.

REFERENCES

[1] J. Bethencourt, A. Sahai, and B. Waters, *Ciphertext-policy attribute based encryption*, in IEEE S&P, 2007.
[2] V. Goyal, A. Jain, O. Pandey, and A. Sahai, *Bounded ciphertext policy attribute based encryption*, in ICALP, 2008.
[3] V. Goyal, O. Pandey, A. Sahai, and B. Waters, *Attribute-based encryption for fine-grained access control of encrypted data*, in ACM CCS, 2006.
[4] J. Katz, A. Sahai, and B. Waters, *Predicate encryption supporting disjunctions, polynomial equations, and inner products*, in EUROCRYPT, 2008.
[5] S. Muller and S. Katzenbeisser, *Hiding the policy in cryptographic access control*, in STM, 2011.
[6] R. Ostrovsky, A. Sahai, and B. Waters, *Attribute-based encryption with non-monotonic access structures*, in ACM CCS, 2007.
[7] A. Sahai, and B. Waters, *Fuzzy identity-based encryption*, in EUROCRYPT, 2005.
[8] T. Ring, *Cloud computing hit by celebgate*, http://www.scmagazineuk. com/cloud-computing-hit-by-celebgate/article/370815/, 2015.
[9] V. Goyal, O. Pandey, A. Sahai, and, B. Waters, *Attribute-based encryption for fine-grained access control of encrypted data*, in ACM CCS, 2006.
[10] X. Jin, R. Krishnan, and R. S. Sandhu, *A unified attribute-based access control model covering DAC, MAC and RBAC*, in DDBSec, 2012.
[11] W. C. Garrison III, A. Shull, S. Myers, and, A. J. Lee, *On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud*, in IEEE S&P, 2016.
[12] R. S. Sandhu, *Rationale for the RBAC96 family of access control models*, in proceedings of ACM Workshop on RBAC, 1995.
[13] A. Ivan, and Y. Dodis, *Proxy Cryptography Revisited*, in proceedings of NDSS, 2003.
[14] M. Blaze, G. Bleumer, and M. Strauss, *Divertible protocols and atomic proxy cryptography*, in proceedings of Eurocrypt, 1998.
[15] G. Ateniese, K. Fuy, M. Green, and S. Hohenbergery, *Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage*, in proceedings of NDSS, 2003.
[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, *Plutus: Scalable Secure File Sharing on Untrusted Storage*, in proceedings of USENIX FAST, 2003.
[17] G. Ateniese, D. H. Chou, B. Medeiros, and G. Tsudik, *Sanitizable Signatures*, in proceedings of ESORICS, 2005.
[18] D. Boneh and M. Franklin, *Identity-based encryption from the Weil pairing*, SIAM Journal on Computing, vol. 32, no. 3, 2003.
[19] M. Green, S. Hohenberger, and B. Waters, *Outsourcing the decryption of abe ciphertexts*, in USENIX Security, 2011.
[20] B. Libert and D. Vergnaud, *Adaptive-id secure revocable identity-based encryption*, in CT-RSA, 2009.
[21] J. Katz, A. Sahai, and B. Waters, *Predicate encryption supporting disjunctions, polynomial equations, and inner products*, in EUROCRYPT, 2008.
[22] E. Shen, E. Shi, and B. Waters, *Predicate privacy in encryption systems*, in TCC, 2009.
[23] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, *Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds*, in NSDI, 2016.
[24] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, *Enabling security in cloud storage SLAs with CloudProof*, in USENIX ATC, 2011.
[25] B. H. Kim, and D. Liey, *Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices*, in IEEE S&P, 2015.
[26] D. Boneh, K. Lewi, H. Montgomery, and A. Raghu Raghunathan, K*ey homomorphic PRFs and their applications*, in CRYPTO, 2013.
[27] M. Maffei, G. Malavolta, M. Reinert, and D. Schr?der, *Privacy and access control for outsourced personal records*, in IEEE S&P, 2015.
[28] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman, *Shroud: ensuring private access to large-scale data in the data center*, in FAST, 2013.
[29] E. Gudes, *The Design of a Cryptography Based Secure File System*, IEEE Transactions on Software Engineering, vol. 6, no. 5, 1980.
[30] S. G. Akl and P. D. Taylor, *Cryptographic solution to a problem of access control in a hierarchy*, TOCS, vol. 1, no. 3, 1983.
[31] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken, *Dynamic and efficient key management for access hierarchies*, TISSEC, vol. 12, no. 3, 2009.
[32] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati, *Enforcing dynamic write privileges in data outsourcing*, Computers & Security, vol. 39, 2013.
[33] A. L. Ferrara, G. Fuchsbauer, B. Liu, and B. Warinschi, *Policy privacy in cryptographic access control*, in CSF, 2015.
[34] L. Ibraimi, *Cryptographically enforced distributed data access control*, Ph.D. dissertation, University of Twente, 2011.
[35] D. Nali, C. M. Adams, and A. Miri, *Using mediated identity-based cryptography to support role-based access control*, in ISC 2004, 2004.
[36] A. L. Ferrara, G. Fuchsbauer, and B. Warinschi, *Cryptographically enforced RBAC*, in CSF, 2013.
[37] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters, *Secure attributebased systems*, in ACM CCS, 2006.
[38] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, *Over-encryption: Management of access control evolution on outsourced data*, in VLDB, 2007.
[39] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, *Encryption policies for regulating access to outsourced data*, TODS, vol. 35, no. 2, 2010.
[40] T. L. Hinrichs, D. Martinoia, W. C. Garrison III, A. J. Lee, A. Panebianco, and L. Zuck, *Application-sensitive access control evaluation using parameterized expressiveness*, in CSF, 2013.
[41] https://www.cryptopp.com/
[42] W. C. Garrison III, A. J. Lee, and T. L. Hinrichs, *An actor-based, application-aware access control evaluation framework*, in SACMAT, 2014.
[43] A. Ene, W. Horne, N. Milosavljevic, P. Rao, R. Schreiber, and R. E. Tarjan, *Fast exact and heuristic methods for role minimization problems*, in SACMAT, 2008.
[44] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo, *Mining roles with semantic meanings*, in SACMAT, 2008.
[45] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, *Ef?cient policy analysis for administrative role based access control*, in CCS, 2007.

APPENDIX A
FORMALIZATION OF CRYPT-DAC

To prove theorem 1, we need to represent Crypt-DAC as a formal access control system. We use m as the symmetric-key size. For signatures, we assume that hash-and-sign is used, where the message is hashed with a collision-resistant hash function and then digitally signed.

Definition 1 access control system: As defined in [40], a formal access control system includes following components:

1) *S:* a set of states

2) *R:* a set of access control requests
3) *Q:* a set of queries including *auth*(*r*) for every $r \in R$
4) $\vdash$*:* a subset of $S \times Q$ (the entailment relation)
5) *L:* a set of labels
6) *Next: States*$(M) \times L \rightarrow$ *States*(M) (the transition function)

We represent Crypt-DAC as a formal access control system as follows.

1) *S:* (USERS, ROLES, FILES, *FS*)
   - USERS: a list of (u, enk_u, sik_u) records containing user names and their encryption and signing key pairs
   - ROLES: a list of (r, v_r, $enk_{r,\, v_r}$, $sik_{r,\, v_r}$) records containing role names, version numbers, and their encryption and signing key pairs
   - FILES: a list of (f_n, v_{f_n}) records containing file names and version numbers
   - *FS*: the set of *RK*, *FK*, and *F* tuples stored on the cloud provider
2) *R:* (u, p)
   - (u, p) for whether user u has permission p
3) *Q:* (*RK*, *FK*, *Role*, *auth*)
   - *RK*(u, r) returns whether a user is a member of a role:
   $$\exists(c,\ sign)\ (\langle RK,\ u,\ (r,\ v_r),\ c,\ sign\rangle \in FS \wedge Accept = Verify_{vk_{SU}}\ ((\langle RK,\ u,\ (r,\ v_r),\ c\rangle),\ sign)$$
   - *FK*(r, (f_n, *op*)) returns whether a role has a permission for the latest version of a file.
   $$\exists(c,\ sign)\ (\langle FK,\ (r,\ v_r)\ (f_n,\ v_{f_n},\ op),\ c,\ SU,\ sign\rangle \in F \wedge sign = Sign_{sk_{SU}}\ (\langle FK,\ (r,\ v_r),\ (f_n,\ v_{f_n},\ op),\ c,\ SU\rangle)$$
   - *Role*(r) returns whether a role exists in the system:
   $$\exists(v,\ k_1,\ k_2)\ (r,\ v,\ k_1,\ k_2) \in ROLES$$
   - *auth*(u, p) returns whether a user has a permission:
   $$\exists r\ (RK(u,\ r) \wedge FK(r,\ p))$$
4) $\vdash$*:* the entailment relation is implicitly defined in the design of Crypt-DAC
5) *L:* all of the operations (**revokeUser**, **revokeRole**, **addUser**, **addRole**, **assignU**, **assignP**, **addFile**, **read**, **write**) in Crypt-DAC
6) *Next:* the transition function is implicitly defined in the design of Crypt-DAC

Definition 2 Correctness: Consider a workload *W*, a candidate access control system *Y*, and an implementation (α, σ, π). Correctness states that the implementation is correct if (1) σ preserves π: for every workload state x we have *Th*(x) = π(*Th*($\sigma(x)$)) and (2) α congruence-preserves σ, which means the following: For all $n \in \mathbb{N}$, states x_0, and labels l_1, ..., l_n, let $y_0 = \sigma(x_0)$, x_i = *Next*(x_{i-1}, l_i) for i = 1, ..., n, and y_i = *terminal*(y_{i-1}, $\alpha(y_{i-1}, l_i)$) for i = 1, ..., n. Then α congruence-preserves σ means that $y_i \cong \sigma(x_i)$ for all i = 1, ..., n.

Definition 3 AC-preservation: An implementation with query-mapping π is called AC-preserving if for all workload states s and authorization requests r we have $s \vdash$ *auth*(r) iff $\pi_{auth(r)}$(*Th*($\sigma(s)$)) = true.

Definition 4 Safety: An implementation is safe if the following holds for all i whenever the execution of a workload label yields the access control state sequence (s_0, ..., s_n).

Auth(s_i) – *Auth*(s_0) $\subseteq$ *Auth*(s_n) – *Auth*(s_0) (Grant)
Auth(s_0) – *Auth*(s_i) $\subseteq$ *Auth*(s_0) – *Auth*(s_n) (Revoke)

APPENDIX B
IMPLEMENTATION OF $RBAC_0$ BY CRYPT-DAC

We show that Crypt-DAC implements $RBAC_0$ with correctness, AC-preserving, and safety achieved.

Definition 5 implementation: For *RBAC*$_0$ and a candidate access control system *Y*, an implementation has fields (α, σ, π):

- State-mapping σ: *States*(*RBAC*$_0$) $\rightarrow$ *States*(*Y*)
- Label mapping α: *States*(*Y*) $\times$ *Labels*(*RBAC*$_0$) $\rightarrow$ *Labels*(*RBAC*$_0$)*
- Query mapping π: for each $q \in$ *Queries*(*RBAC*$_0$), a function π_q that maps each theory for *Y* to either true or false

We first show a implementation of $RBAC_0$ by Crypt-DAC:

1) *State mapping* α:
   For each $u \in U \cup \{SU\}$
   - Generate $enk_u \leftarrow Gen^{Pub}$ and $sik_u \leftarrow Gen^{Sign}$.
   - Add (u, ek_u, vk_u) to USERS

   Let *FS* = {}
   Let ROLES and FILES be blank. For each *R*(r) $\in$ *M*:
   - Generate $enk_{r,1} \leftarrow Gen^{Pub}$ and $sik_{r,1} \leftarrow Gen^{Sign}$.
   - Add (r, 1, $ek_{r,1}$, $vk_{r,1}$) to ROLES.

- Update $FS = FS \cup \{\langle RK, SU, (r, 1), Enc^{Pub}_{ek_{SU}}(dk_{r,1}, sk_{r,1}), sign_{SU}\rangle\}$.

For each $P(f_n, u) \in M$:

- Add $(f_n, 1)$ to FILES
- Generate $k \leftarrow Gen^{Sym}$
- Update $FS = FS \cup \{\langle F, f_n, Enc^{Sym}_{k}(f), sign_u\rangle\}$
- Update $FS = FS \cup \{\langle FK, (SU, 1), (f_n, 1, RW), Enc^{Pub}_{ek_{SU}}(k), sign_u\ \rangle\}$

For each $UR(u, r) \in M$:

- Find $\langle FK, (SU, 1), (r, 1, RW), c, sign\rangle \in FS$
- Update $FS = FS \cup \{\langle RK, u, (r, 1), Enc^{Pub}_{ek_u}(dk_{r,1}, sk_{r,1}), sign_{SU}\rangle\}$.

For each $PA(r, (f_n, op)) \in M$:

- Find $\langle FK, (SU, 1), (f_n, 1, RW), Enc^{Pub}_{ek_{SU}}(k), sign_{SU}\rangle \in FS$
- Update $FS = FS \cup \{\langle FK, (r, 1), (f_n, 1, RW), Enc^{Pub}_{ek_{r,1}}(k), sign_{SU}\rangle\}$

Output (FS, ROLES, FILES)

2) *Label mapping σ:* The label mapping α simply maps any $RBAC_0$ label, regardless of the state, to the Crypt-DAC label.
3) *Query mapping π*:

$$\pi_{UR(u,r)}(T) = true \text{ if } RK(u, r) \in T$$
$$\pi_{PA(r,p)}(T) = true \text{ if } FK(r, p) \in T$$
$$\pi_{R(r)}(T) = true \text{ if } Role(r) \in T$$
$$\pi_{auth(u,p)}(T) = true \text{ if } auth(u, p) \in T$$

for each theory T of Crypt-DAC.

In crypt-DAC, the encryptions, signatures and version numbers in *RK*/*FK*/*F* tuples are different after a revocation. As a result, assigning and then revoking a user will not result in the same state as if the user was never assigned. However, all the authorization requests will not be changed. These differences, however, make crypt-DAC unable to achieve the original definition of correctness. For example, l_1 assigns a user to a role and then l_2 revokes that user from the role, x_2 will be equal to x_0. Accordingly, $\sigma(x_2)$ should be equal to $\sigma(x_0)$. However, y_2 will have different encryptions, signatures and version numbers from y_0.

We follow the way of [11] and prove a variant of correctness named congruence correctness. We consider states, which are equal except for differences in encryptions, signatures and version numbers, to be congruent, and represent this with the $\cong$ relation. We define that state mappings σ' and σ are congruent if $\sigma'(x) \cong \sigma(x)$ for all states x. Thus we will show that $y_i \cong \sigma(x_i)$ for all i = 1,..., n, which we define as α congruence preserves σ.

We first prove that Crypt-DAC implements $RBAC_0$ with congruence-correctness achieved:

1) *σ preserves π*: To prove this, we show that for each $RBAC_0$ state x and query q, $x \vdash q$ if and only if $\pi_q(Th(\sigma(x)))$ = TRUE. As the proof is similar with that of [11], we just omit the details here.

2) *α congruence-preserves σ:* To prove this, we show that for each $RBAC_0$ state x and label l, and state mappings σ' congruent to σ, we have:

$$\sigma'(next(x, l)) \cong terminal(\sigma'(x), \alpha(\sigma'(x), l))$$

We consider each $RBAC_0$ label l separately. As the proof is similar with that of [11] except **revokeU** and **revokeR**, we just show the two labels here.

- **revokeU:** If l is an instance of $revokeU(u, r)$, then $x' = x \backslash UR(u, r)$. Let $(ek_{r,v_r+1}, dk_{r,v_r+1}) \leftarrow Gen^{Pub}$ and $(sk_{r,v_r+1}, vk_{r,v_r+1}) \leftarrow Gen^{Sig}$. Let $T_0 = \{(u', sig) \mid \exists\langle RK, u', (r, v_r), c_{u'}, sig\rangle \in FS\}$ and $T_1 = \{f_n \mid \exists\langle FK, (r, v_r), (f_n, v_{f_n}, op), v_{f_n}, c_r, sig\rangle \in FS\}$. For each $f_n \in T_1$, let $T_{f_n} = \{(r, sig) \mid \langle FK, (r, v_r), (f_n, v_{f_n}, op'), c_r, sig\rangle \in FS\}$ and $T'_{f_n} = \{(r', sig) \mid \langle FK, (r', v_{r'}), (f_n, v_{f_n}, op), c_{r'}, sig\rangle \in FS\}$. Then:

$\sigma'(x') = \sigma'(x \setminus UR(u, r))$
$= \sigma'(\text{next}(x, revokeU(u, r))$
$\cong \sigma'(x) \backslash \{FS(\langle RK, u', (r, v_r), c_{u'},$
$sig\rangle) \mid (u', sig) \in T_0\} \cup \{FS(\langle RK, u',$
$(r, v_r + 1), c'_{u'}, sig\rangle) \mid (u', sig) \in T_0 \wedge u' \neq u\}$
$\backslash \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, op'), c_r,$
$sig\rangle) \mid f_n \in T_1 \wedge (r, sig) \in T_{f_n}\}$
$\cup \{FS(\langle FK, (r, v_r + 1), (f_n, v_{f_n}+1, op'), c'_r, sig\rangle) \mid f_n \in T_0 \wedge (r, sig) \in T_{f_n}\}$
$\backslash \{FS(\langle FK, (r', v_{r'}), f_n, v_{f_n}, op'), c_{r'}$
$, sig\rangle) \mid f_n \in T_1 \wedge (r', sig) \in T'_{f_n}\}$

$\cup\{FS(\langle FK, (r', v_{r'}), (f_n, v_{f_n}+1, op'), c'_{r'}, sig\rangle) \mid f_n \in T_1 \wedge (r', sig) \in T'_{f_n}\}$
$\setminus \{FS(\langle F, f_n, c_{f_n}, sig\rangle) \mid (f_n) \in T_1\}$
$\cup\{FS(\langle F, f_n, c'_{f_n}, sig\rangle) \mid (f_n) \in T_1\}$
$\cup\{\text{FILES}(f_n, v_{f_n}+1) \mid f_n \in F\}$
$\setminus \{\text{FILES}(f_n, v_{f_n}) \mid f_n \in F\}$
$\cup$ ROLES(r, v_r+1, $ek_{(r,v_r+1)}$, $vk_{(r,v_r+1)}$)
$\setminus$ ROLES(r, v_r, $ek_{(r,v_r)}$, $vk_{(r,v_r)}$)
$=terminal(\sigma'(x), \alpha'(\sigma'(x), l))$

- **revokeP:** If l is an instance of *revokeP*(r, p) with $p = \langle f_n, op\rangle$, then $x' = x \setminus PA(r, p)$. We consider two cases:

  – If op = *RW*, then let $T = \{(r, c, sig) \mid \exists\langle FK, (r, v_r), (f_n, v_{f_n}, RW), c, sig\rangle\}$. Then
  $\sigma'(x') = \sigma'(x \setminus PA(r, p))$
  $=\sigma'(\text{next}(x', revokeP(r, p)))$
  $=\sigma'(x) \setminus \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, RW),$
  $c, sig\rangle) \mid (r, c, sig) \in T\}$
  $\cup\{FS(\langle FK, (r, v_r),(f_n, v_{f_n}, Read),$
  $c, sig\rangle) \mid (r, c, sig) \in T\}$
  $=terminal(\sigma'(x), \alpha(\sigma'(x), l))$.

  – If op = *Read*, then let $T = \{(r, sig) \mid \langle FK,(r, v_r), (f_n, v_{f_n}, op'), c, sig\rangle \in FS\}$, $T_1 = \{(r', op') \mid r' \neq r \wedge \exists\langle FK,$ $r', (f_n, v_{f_n}, op'), c_{r'}, sig\rangle \in FS)\}$, and $T_2 = \{(f_n) \mid \langle F, f_n, c_{f_n}, SU, sig\rangle \in FS)\}$. Then
  $\sigma'(x') = \sigma'(x \setminus PA(r, p))$
  $=\sigma'(\text{next}(x, revokeP(r, p))$
  $\cong\sigma'(x) \setminus \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, op'),$
  $c, sig\rangle) \mid (r, sig) \in T\}$
  $\setminus \{FS(\langle FK, r', (f_n, v_{f_n}, op'),$
  $c_{r'}, sig\rangle) \mid (r', op') \in T_1\}$
  $\cup\{FS(\langle FK, r', (f_n, v_{f_n}, op'), v_{f_n}+1,$
  $c'_{r'}, sig\rangle) \mid (r', op') \in T_1\}$
  $\setminus \{FS(\langle F, f_n, c_{f_n}, sig\rangle) \mid (f_n) \in T_2\}$
  $\cup\{FS(\langle F, f_n, c'_{f_n}, sig\rangle) \mid (f_n) \in T_2\}$
  $\cup$ FILES(f_n, v_{f_n}+1) $\setminus$ FILES(f_n, v_{f_n})
  $=terminal(\sigma'(x), \alpha(\sigma'(x), l))$.

We then prove that Crypt-DAC implements $RBAC_0$ with AC-preserving achieved. The query mapping π is AC-preserving because it maps *auth*(u, p) to TRUE for theory T if and only if T contains *auth*(u, p).

We finally prove that Crypt-DAC implements $RBAC_0$ with safety achieved. The label mapping α is safe by inspection–for any $RBAC_0$ state x and label l, the Crypt-DAC label $\alpha(\sigma(x), l)$ never revokes or grants authorizations except the images of those that are revoked or granted by l.