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

Saiyu Qi
State Key Laboratory of Integrated Service Networks (ISN)
Xidian University, China
The State Key Laboratory of Cryptology, PO Box 5159, Beijing 100878, China
syqi@xidian.edu.cn

Yuanqing Zheng
Department of Computing
The Hong Kong Polytechnic University, China
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 challenging. 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 uploads 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 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. Cloud service 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 [10], have raised concerns regarding the privacy of cloud-managed data. Actually, a cloud service provider is usually not secure due to design drawbacks of software and system vulnerability [2], [3]. 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], [4]–[9] 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) [5] is a cryptographic counterpart of attribute-based access control (ABAC) model [11]. However, previous works mainly consider static scenarios in which access control policies rarely change. The previous works incur high overhead when access control policies need to be changed 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 keep a local copy of the keys before the revocation. To prevent such a problem, files have to be re-encrypted with new keys. This requires the file owner to download the file, re-encrypt the file, and upload it back for the cloud to update the previous encrypted file, incurring prohibitive communication overhead at the file owner side.

Currently, only a few works investigated the problem of dynamic data access control. Garrison et al. [12] proposed two revocation schemes. The first scheme requires an administrator to re-encrypt file with new keys as discussed above. This scheme incurs a considerable communication overhead. Instead, the second scheme 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, in which the symmetric homomorphic encryption scheme [24] is used to encrypt the file. Such a design enables the cloud to directly re-encrypt file without decryption. However, this scheme 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 files in permission revocations. In Crypt-DAC, a file is encrypted by a symmetric key list which records a file key and a sequence of revocation keys. In a revocation, the administrator uploads a new revocation key to the cloud, which encrypts the file with a new layer of encryption and updates the encrypted key list accordingly. Same as previous works [12], [23], we assume a honest-but-curious cloud, i.e., the cloud is honest to perform the required commends (such as re-encryption of files and properly update previous encrypted files) but is curious to passively gathering sensitive information. 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, which incurs additional decryption overhead for users to access files. 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 and decrypt a large key list in each file access. To overcome this problem, we adopt the key rotation technique [15] to compactly encrypt the key list in the policy data. 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 delegating encryption operations to 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. For revocation efficiency, Crypt-DAC incurs lightweight communication overhead at the administrator side as it does not need to download and re-upload file data. 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. We have implemented Crypt-DAC as well as several recent works [12], [23] on Alicloud. Real experiments suggest that Crypt-DAC is three orders of magnitude more efficient in communication in access revocation compared with the first scheme in [12], and is nearly two orders of magnitude more efficient in computation in file access compared with the scheme in [23]. Finally, Crypt-DAC is able to immediately revoke access permissions compared with the second scheme in [12].

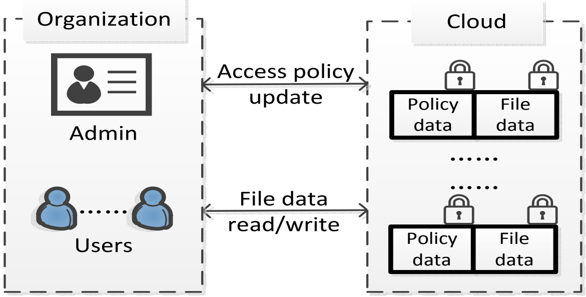


Fig. 1. Cloud enabled data access control.

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. 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. We do not consider data deduplication issue. IF needed, secure data deduplication technique [14] can be used here. We also assume that 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 try to access the file data out of their access permissions by compromising the cloud provider. Same as previous works [12], [23], we assume a honest-but-curious cloud provider. Honest means that the cloud provider honestly follows the commands of the administrator/users such as re-encryption of policy/file data and properly update previous policy/file data. Since any errors due to the provider's misbehavior will harm its reputation, we believe that the cloud provider has incentive to follow the commands required by its customers. However, the cloud provider may be curious to passively gathering sensitive information to acquire commercial benefits as it is hard to be detected.

We notice that the honest-but-curious assumption is critical to resist collusion between the cloud provider and revoked users. Generally, state of the art works fall in two ways to revoke access privileges. The first is for a data owner/administrator to download, re-encrypt, and upload the policy/file data to the cloud provider (e.g. [12]). The second is to delegate the cloud provider to directly re-encrypt the policy/file data (e.g. [23]). In both cases, if a malicious cloud provider does not properly update the previous policy/file data and retains a copy of it before the re-encryption, then the revoked users can continuously access the files. As policy/file data is fully managed by the cloud provider, how to resist such collusion attack without the honest-but-curious assumption is still an open problem.

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 ($RBAC_0$) [13], 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/Asymmetric cryptography: Our construction makes use of symmetric-key encryption scheme (Gen^{Sym}, Enc^{Sym}, Dec^{Sym}), public-key encryption scheme (Gen^{Pub}, Enc^{Pub}, Dec^{Pub}) and digital signature scheme (Gen^{Sig}, Sign, Ver).

Key rotation scheme: Key rotation [15] 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:

$\mathsf{Gen}^{Ro}(1^n) \rightarrow (rsk,\ rpk)$: 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.

A. Basic construction

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

Access enforcement: For each user u, the administrator distributes a certificate for the user write key of u through a user (U) tuple:

$$\langle U,\ (u,\ vk_u),\ \delta_{SU} \rangle$$

This tuple contains a user name u, a verification key vk_u, and a signature of the administrator δ_{SU} signed over (u, vk_u). 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:

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

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:

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

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:

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

This tuple contains a file name f_n of f and a ciphertext of f.

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 F tuple encrypting the new file content f' as well as a signature δ_u over the F tuple signed by the user write key of u. On the other hand, the cloud provider checks (1) if there is an RK tuple assigning u as a member of r and an FK tuple assigning r with permission RW to f; and (2) if there is a U tuple containing vk_u that verifies δ_u as a valid signature over the F tuple. If both checks passed, 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. An illustrative example is shown in Figure 2.

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 access all of these files to cache their file keys when he join the system. Later, u^* is revoked from one of its three roles and cannot access some of 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.

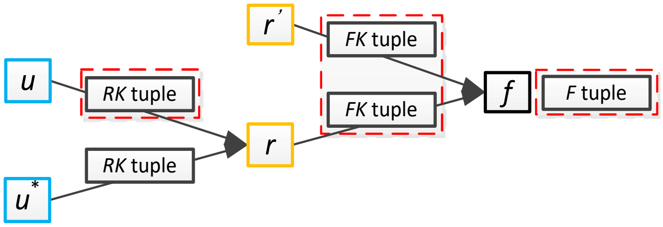


Fig. 2. An illustrative example of removing a user u^* from a role r. The black arrows present the access relationships among users, roles and files before the revocation, and the red rectangles include the RK, FK, and F tuples that need to be re-generated in the revocation. In the first step, a new role key of r is generated. This key is then encrypted into a new RK tuple for u, which is a remaining member of r. In the second step, a new file key is generated for f, which is accessible by r. This key is then encrypted into two new FK tuples for r' and r respectively that both have access to f. In the third step, f is re-encrypted into a new F tuple by the new file key.

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 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. [12] proposed two revocation schemes. The first scheme requires the administrator to re-encrypt file data by itself in a revocation. This scheme completes the revocation immediately with a potentially high communication overhead. Differently, the second scheme 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 which they have accessed previously and cached the file keys.

To alleviate the overhead of file data re-encryption, Wang et al. [23] proposed another revocation scheme, in which the symmetric homomorphic encryption scheme [24] is used to encrypt the file data. Instead by re-encrypting the file data by itself, the scheme enables the administrator to delegate the cloud to update F tuples from old file keys to new file keys without decryption. The problem, 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.

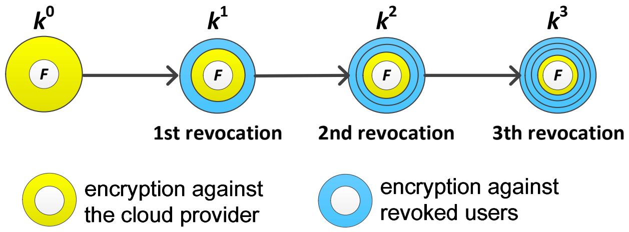

k^0
k^1
k^2
k^3
F
F
F
F
1st revocation
2nd revocation
3th revocation
encryption against the cloud provider
encryption against revoked users

Fig. 3. An illustrative example of encryption evolution of a F tuple. The F tuple is encrypted by a symmetric key list (k^0, k^1, k^2, k^3) which records a file key k^0 and a sequence of revocation keys k^1, k^2, k^3. In the ith revocation ($i \in (1, 2, 3)$), the administrator uploads a new revocation key k^i to the cloud, which encrypts the file with a new layer of encryption.

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 ith revocation, the administrator uploads a new revocation key k^i 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. An illustrative example is shown in Figure 3.

Compared with previous designs, Crypt-DAC achieves efficient revocation, immediate revocation, and efficient file access simultaneously. For revocation efficiency, Crypt-DAC incurs lightweight communication overhead at the administrator side as it does not need to download and re-upload file data but only needs to upload 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 avoid users to decrypt multiple encryption layers in file access operations, Crypt-DAC proposes three key techniques to constrain the size of key list and encryption layers for files 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 [15] to compactly encrypt each key list in one encryption in a *FK* tuple. As a result, the administrator only needs to upload one encryption for the cloud to update a key list. This design also improves file access efficiency as users only need to download and decrypt compact *FK* tuples to access files.

To revoke a user u^* from a role *r*, Crypt-DAC uses the delegation-aware encryption strategy to update the involved *RK* and *FK* tuples as shown in Figure 4. 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 $\mathsf{Enc}^{Pub}_{ek_u}(dk_r^*)$. With this encryption, the cloud updates the *RK* tuple as:

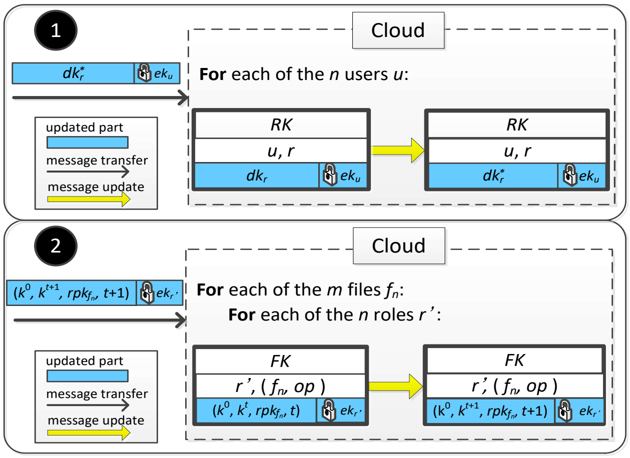


Fig. 4. To revoke a user u^* from a role *r*, the delegation-aware encryption strategy updates the involved *RK* and *FK* tuples in two steps. The administrator first delegates the cloud provider to update *RK* tuples. Suppose there are *n* users remaining in *r*. For each of the *n* users *u*, the administrator uploads an encryption of a new role key to the cloud provider for it to update the *RK* tuple of *u*. The administrator next delegates the cloud provider to update *FK* tuples. Assume that there are *m* files to which *r* has permissions, and *n* roles with permissions to each of the *m* files. For each of the *m*×*n* roles *r'*, the administrator uploads an encryption of a new revocation key to the cloud provider for it to update the *FK* tuple of *r'*.

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

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 $k^{t+1} \leftarrow \mathsf{B\text{-}Dri}(k^t, rsk_{f_n})$ and the whole revocation key sequence k^1,..., k^t can be recovered by $k^{i-1} \leftarrow \mathsf{F\text{-}Dri}(k^i, rpk_{f_n})$ ($2 \leq i \leq t$). As a result, the complete format of an *FK* tuple is:

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

$$c = \mathsf{Enc}^{Pub}_{ek_r}(k^0,\ k^t,\ rpk_{f_n},\ t)$$

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 $k^{t+1} \leftarrow \mathsf{B\text{-}Dri}(k_t, rsk_{f_n})$. For each of the *n* roles *r'* 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 uploads 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:

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

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

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

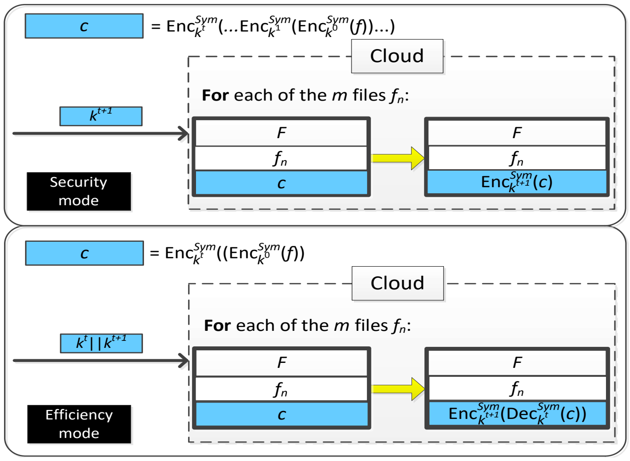

c
= $\mathsf{Enc}^{Sym}_{k^t}$(...$\mathsf{Enc}^{Sym}_{k^1}$($\mathsf{Enc}^{Sym}_{k^0}$(f))...)
Cloud
For each of the m files f_n:
k^{t+1}
F
f_n
c
F
f_n
$\mathsf{Enc}^{Sym}_{k^{t+1}}(c)$
Security mode
c
= $\mathsf{Enc}^{Sym}_{k^t}$(($\mathsf{Enc}^{Sym}_{k^0}$(f))
Cloud
For each of the m files f_n:
$k^t||k^{t+1}$
F
f_n
c
F
f_n
$\mathsf{Enc}^{Sym}_{k^{t+1}}(\mathsf{Dec}^{Sym}_{k^t}(c))$
Efficiency mode

Fig. 5. To complete the revocation of the user u^* from the role r, the adjustable onion encryption strategy provides a security mode and an efficiency mode to update the involved F tuples. In the security mode, the encryption layers of an F tuple increase as the revocations increase. For each of the m files f_n to which r has permissions, the administrator uploads a revocation key to the cloud provider for it to directly encrypt the F tuple of f_n. In the efficiency mode, the encryption layers of an F tuple are constant regardless of the revocations. For each of the m files f_n to which r has permissions, the administrator uploads two revocation keys to the cloud provider for it to first decrypt and then encrypt the F tuple of f_n.

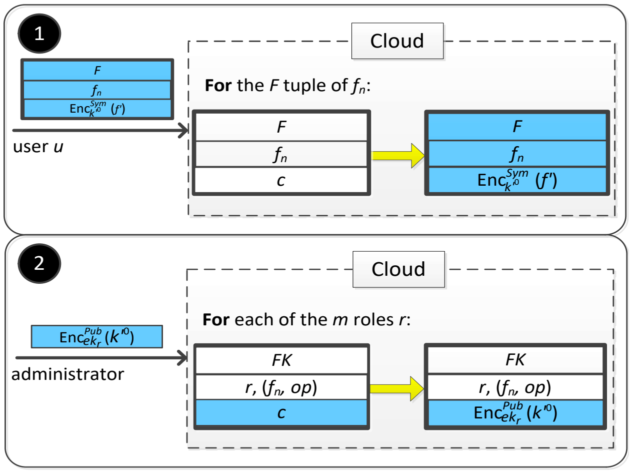

1
F
f_n
$\mathsf{Enc}^{Sym}_{k'^0}(f')$
user u
Cloud
For the F tuple of f_n:
F
f_n
c
F
f_n
$\mathsf{Enc}^{Sym}_{k'^0}(f')$
2
$\mathsf{Enc}^{Pub}_{ek_r}(k'^0)$
administrator
Cloud
For each of the m roles r:
FK
r, (f_n, op)
c
FK
r, (f_n, op)
$\mathsf{Enc}^{Pub}_{ek_r}(k'^0)$

Fig. 6. The delayed de-onion encryption strategy periodically refreshes the symmetric key list of files and removes the encryption layers of F tuples through file writing operations. With the strategy, a user u writes to a file f_n in two steps. u first updates the F tuple of f_n. To do so, u generates a F tuple to encrypt f_n with a new key list containing a single key, and uploads the F tuple to the cloud provider for it to replace the current F tuple of f_n. u next delegates the administrator to update the key list of f_n. Assuming that there are m roles having permissions to f_n. For each of the m roles r, the administrator uploads an encryption of the new key list for the cloud provider to update the FK tuple of r.

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 upload a new revocation key to the cloud provider. Upon receiving the key, the cloud provider uses it to encrypt the files with a new layer of encryption and then deletes it. 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.

To complete the revocation of the user u^* from the role r, Crypt-DAC uses the adjustable onion encryption strategy to update the involved F tuples as shown in Figure 5. The strategy provides two modes: security mode and efficiency mode to do so as follows.

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:

$$\langle F,\ f_n,\ c\rangle$$

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

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}←F-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, 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-Dri(k^t, rsk_{f_n}) and uploads k^{t+1} to the cloud provider. Upon receiving it, the cloud provider updates the F tuples of the m files as:

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

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:

$$\langle F,\ f_n,\ c\rangle$$

$$c = \mathsf{Enc}^{Sym}_{k^t}(\mathsf{Enc}^{Sym}_{k^0}(f))$$

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, 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-Dri(k^t, rsk_{f_n}) and uploads (k^t, k^{t+1}) to the cloud provider. Upon receiving it, the cloud provider updates the F tuples of the m files as:

$$\langle F,\ f_n,\ \mathsf{Enc}^{Sym}_{k^{t+1}}(\mathsf{Dec}^{Sym}_{k^t}(c))\rangle$$

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 predefined bound is reached. To further improve file access efficiency, Crypt-DAC periodically refreshes the symmetric key list of the file and removes 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 work flow of the delayed de-onion encryption strategy is shown in Figure 6. Suppose a user u wants to write to a file f_n. To do so, u first updates the F tuple of f_n. u generates a new file key $k'^0 \leftarrow \mathsf{Gen}^{Sym}(1^n)$ and creates a F tuple encrypting the writing content *f'* by a new symmetric key list (k'^0):

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

u uploads the F tuple to the cloud provider for it to replace the existing F tuple.

u next updates the key list of f_n. To alleviate the overhead, u delegates the administrator to do so. Assume that there are m roles having permissions to f_n. u uploads the new key list (k'^0) to the administrator and delegates it to update the key list in the *FK* tuples of the m roles. 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:

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

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. Our design uses the following notation: u is a user, r is a role, p is a permission, f_n is a file name of a file f, c is a ciphertext (either symmetric or public encryption), and v is a version number. *SU* is the superuser identity owned by the administrator. We use – to represent a wildcard.

A. File management

We use three types of files to store metadata for file management. We introduce them as follows.

USERS: We use a file named USERS to store records for all the users. A record (u, ek_u) contains a user identity u and the encryption key ek_u of u.

ROLES: We use a file named ROLES to store records for all the roles, which is publicly viewable and can only be changed by the administrator. A record (r, ek_{v_r}) contains a role identity r and the encryption key ek_{v_r} of r.

FILES: We use a file named FILES to store records for all the files, which is publicly viewable and can only be changed by the cloud provider. A record (f_n) contains the file name f_n of a file.

B. Key management

In our system, the administrator, roles and users are associated with cryptographic keys. We introduce them as follows.

Administrator keys: The administrator plays a role of super user in the system. It has an encryption key pair (ek_{SU}, dk_{SU}) of a public key encryption scheme and a signature key pair (sk_{SU}, vk_{SU}) of a digital signature scheme. The encryption key pair is used by the administrator to create a special *RK* tuple when adding a new role into the system. The *RK* tuple means that the administrator is a super user who can access the keys of the role. With this *RK* tuple, the administrator can assign a user to the role by distributing the role keys using another *RK* tuple. The encryption key pair is also used by a user to create a special *FK* tuple when adding a new file into the system. The *FK* tuple means that the administrator is a super user who can access the symmetric key list of the file. With this *FK* tuple, the administrator can assign a permission to a role by distributing the symmetric key list using another *FK* tuple. On the other hand, the signature key pair is used to assign U tuples to users.

User keys: A user read key of u is an encryption key pair (ek_u, dk_u) of a public key encryption scheme. This key is used to encrypt/decrypt *RK* tuples for u. A user write key of u

Algorithm 1 revokeUser(*u*)

```
For each role r that u is assigned to:
  REVOKEU(u, r);
Req C.P. to delete ⟨U, (u, $vk_u$), $\delta_{SU}$⟩;

procedure REVOKEU(u, r)
  DAE-RK(r);
  DAE-FK(r);
  For each $f_n$ with ⟨FK, r, ($f_n$, op), c⟩:
    ONION-ENCRYPTION($f_n$);

procedure DAE-RK(r)
  Generate a new role key ($ek_r$, $dk_r$) for r;
  For each ⟨RK, u', r, c⟩ with u' $\neq$ u:
    Admin:
      Send $\mathsf{Enc}^{Pub}_{ek_{u'}}(dk_r)$ to C.P.;
    C.P.:
      Update ⟨RK, u', r, c⟩ as ⟨RK, u', r, $\mathsf{Enc}^{Pub}_{ek_{u'}}(dk_r)$⟩;

procedure DAE-FK(r)
  For each $f_n$ with ⟨FK, r, ($f_n$, op), c⟩:
    Admin:
        Generate a new revocation key $k^{t+1} \leftarrow \mathsf{D\text{-}Dri}(k_t, rsk_{f_n})$;
        For each role r' with permission to $f_n$:
            Compute $c' = \mathsf{Enc}^{Pub}_{ek_{r'}}(k^0, k^{t+1}, rpk_{f_n}, t+1)$;
            Send $c'$ to C.P.;
    C.P.:
        For each role r' with permission to $f_n$:
            Update ⟨FK, r', ($f_n$, op), c⟩ as ⟨FK, r', ($f_n$, op), c'⟩;

procedure ONION-ENC($f_n$)
  Admin:
    Compute $\hat{k}^{t+1} \leftarrow \mathsf{hash}(k^{t+1}, t+1)$;
    Send ($\hat{k}^{t+1}$/$\hat{k}^{t+1}$, $\hat{k}^t$) to C.P.;
  C.P.: ⟨F, $f_n$, c⟩:
    Compute $c \leftarrow \mathsf{Enc}^{Sym}_{\hat{k}^{t+1}}(c)$/$c \leftarrow \mathsf{Enc}^{Sym}_{\hat{k}^{t+1}}(\mathsf{Dec}^{Sym}_{\hat{k}^t}(c))$;
```

Algorithm 2 revokeRole(*r*)

```
Remove (r, $ek_r$) from ROLES;
For each permission p = ⟨$f_n$, op⟩ that r has access to:
  REVOKEP(r, ⟨$f_n$, Read⟩);

procedure REVOKEP(r, ⟨$f_n$, RW⟩)
  Admin:
      Send ⟨FK, r, ($f_n$, Read), c⟩ to C.P.;
  C.P.:
      Update ⟨FK, r, ($f_n$, RW), c⟩ as ⟨FK, r, ($f_n$, Read), c⟩;

procedure REVOKEP(r, ⟨$f_n$, Read⟩)
  Req C.P. to delete all ⟨RK, –, r, – ⟩;
  Req C.P. to delete ⟨FK, r, ($f_n$, –), –⟩;
  VDAE-FK($f_n$);
  ONION-ENCRYPTION($f_n$);

procedure VDAE-FK($f_n$)
  Admin:
    Generate a new revocation key $k^{t+1} \leftarrow \mathsf{D\text{-}Dri}(k_t, rsk_{f_n})$;
    For each role r' $\neq$ r with permission to $f_n$:
        Compute $c' = \mathsf{Enc}^{Pub}_{ek'_r}(k^0, k^{t+1}, rpk_{f_n}, t+1)$;
        Send $c'$ to C.P.;
  C.P.:
    For each role r' $\neq$ r with permission to $f_n$:
      Update ⟨FK, r', ($f_n$, op), c⟩ as ⟨FK, r', ($f_n$, op), c'⟩;
```

is a digital signature key pair (ek_u, dk_u) of a digital signature scheme. This key is used to sign/verify *F* tuples written by *u*.

Role keys: A role key of *r* is an encryption key pair (ek_r, dk_r) of a public key encryption scheme. This key pair is used to encrypt/decrypt *FK* tuples for *r*.

File keys: A file key of f_n is a symmetric key list (k^0, k^1,... k^t) of a symmetric key encryption scheme and a rotation key pair (rsk_{f_n}, rpk_{f_n}) of a key rotation scheme. (k^0, k^1,... k^t) is used by users to encrypt the *F* tuple of f_n, and (rsk_{f_n}, rpk_{f_n}) is used by the administrator to compactly store (k^0, k^1,... k^t) in the *FK* tuple of f_n.

C. Operations

Permission revocation: Permission revocation includes revoking the permission of a user $\mathsf{revokeUser}(u)$ (as described in **Algorithm** 1) and revoking the permission of a role $\mathsf{revokeRole}(r)$ (as described in **Algorithm** 2).

The administrator uses $\mathsf{revokeUser}(u)$ to revoke the permission of a user *u* from all of its assigned roles. The algorithm invokes DAE-RK(*r*) and DAE-FK(*r*), which implements the delegation-aware encryption strategy, to update the involved *RK* and *FK* tuples respectively. The algorithm also invokes ONION-ENC(f_n), which implements the adjustable onion encryption strategy, to update the involved *F* tuples. Also, the administrator can directly use REVOKEU(*u*, *r*) 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. The algorithm invokes VDAE-FK(*r*), which partly implements the delegation-aware encryption strategy, to update the involved *FK* tuples. The algorithm also invokes ONION-ENC(f_n) to update the involved *F* tuples. $\mathsf{revokeRole}(r)$ can be slightly modified to revoke the permission of *r* from a single file f_n. There are two cases. First, the administrator can directly use REVOKEP(*r*, ⟨f_n, *RW*⟩) to revoke a permission of *r* from ⟨f_n, *RW*⟩ to ⟨f_n, *Read*⟩. Second, the administrator can directly use REVOKEP(*r*, ⟨f_n, *Read*⟩) to totally revoke a permission ⟨f_n, *op*⟩ of *r*.

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 $\mathsf{assignP}(r, \langle f_n, op\rangle)$. Due to the space constraint, we omit the algorithm details of these operations. 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 $\mathsf{assignP}(r, \langle f_n, op\rangle)$ 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 $\mathsf{read}(f_n, u)$ (as described in **Algorithm** 3), and file write $\mathsf{write}(f_n, u)$ (as described in **Algorithm** 4). 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 invokes DELAYED DE-ONION(⟨*RK*, *u*, *r*, *c*⟩), which implements the delayed de-onion encryption strategy, to refresh the symmetric key list of f_n. We noticed that in some cases, *u* does not need to execute the delayed de-onion encryption strategy as no revocation happens subject to f_n.

D. Data integrity

Many works [16]–[19] have been done to ensure data integrity in untrusted cloud. Lighten by these works, we show how to extend Crypt-DAC to ensure data integrity when a

Algorithm 3 read(f_n, *u*)

```
User u:
  Send (u, r, $f_n$) to C.P.;
C.P.:
  If there exists an RK tuple $\langle$RK, u, r, c$\rangle$ $\wedge$
    a FK tuple $\langle$FK, r, ($f_n$, op), c'$\rangle$ $\wedge$
    a F tuple $\langle$F, $f_n$, c"$\rangle$:
  Then
    Return the RK tuple, FK tuple, and F tuple to u;
  Else
    Return $\perp$ to u;
User u:
    Compute $dk_r \leftarrow \mathsf{Dec}^{Pub}_{dk_u}(c)$;
    Compute $(k_0, k_t, rpk_{f_n}, t) \leftarrow \mathsf{Dec}^{Pub}_{dk_r}(c')$;
    For i = t & i > 1 & i−−:
      Compute $k_{i-1} \leftarrow \mathsf{F\text{-}Dri}_{rpk_{f_n}}(k_i)$;
    Compute $\hat{k_t}$ = $\mathsf{hash}(k_t, t)$;
    Compute $c" \leftarrow \mathsf{Dec}^{Sym}_{\hat{k_t}}(c")$;
    For i = T & i ≥ 0 & i−−:
      Compute $\hat{k_i}$ = $\mathsf{hash}(k_i, i)$;
      Compute $c" \leftarrow \mathsf{Dec}^{Sym}_{\hat{k_i}}(c")$;
    Output c";
```

Algorithm 4 write(f_n, *u*)

```
User u:
  Send ($f_n$, write request) to C.P.;
C.P.:
  Include all the FK tuples containing $f_n$ into FK-set;
  Return $\langle$RK, u, r, c$\rangle$ to u;
User u:
    DELAYED DE-ONION($\langle$RK, u, r, c$\rangle$);

procedure DELAYED DE-ONION($\langle$RK, u, r, c$\rangle$)
  User u:
    Compute $k'^0 \leftarrow \mathsf{Gen}^{Sym}()$;
    Compute $c' \leftarrow \mathsf{Enc}^{Sym}_{k'^0}(f')$;
    Compute $\delta_u \leftarrow \mathsf{Sign}_{sk_u}(F, f_n, c')$;
    Send $\langle F, f_n, c', \delta_u \rangle$ to C.P.;
    Send $(k'^0, f_n)$ to Admin;
  Admin:
    For each role r' with permission to $f_n$:
      Compute $c" \leftarrow \mathsf{Enc}^{Pub}_{pk_r}(k'^0, rpk_{f_n}, 0)$;
      Insert c" into C-set;
    Send C-set to C.P.;
  C.P.:
    If there exists a U tuple $\langle U, (u, vk_u), \delta_{SU} \rangle$|valid $\leftarrow \mathsf{Ver}_{vk_u}(F, f_n, c') \wedge$
    an RK tuple $\langle$RK, u, r, c$\rangle$ $\wedge$ a FK tuple $\langle$FK, r, ($f_n$, RW), c'$\rangle$:
    Then Write $\langle F, f_n, c' \rangle$;
          For each $\langle$FK, r', ($f_n$, RW), c'$\rangle$ $\in$ FK-set and a proper c" $\in$ C-set:
          Update $\langle$FK, r', ($f_n$, RW), c'$\rangle$ to $\langle$FK, r', ($f_n$, RW), c"$\rangle$;
    Else
      Return $\perp$ to u;
```

stronger threat model is needed. The two schemes in [12] can achieve data integrity by using signature scheme to sign *RK*, *FK*, and *F* tuples. As these tuples are updated at the administrator/user side, the administrator/user can generate new signatures over the updated tuples before uploading to the cloud provider. On the other hand, Crypt-DAC and the scheme in [23] cannot to do so by directly using signature scheme. As the administrator delegates the cloud provider to update *RK*, *FK*, and *F* tuples, the cloud provider cannot generate new signatures over the updated tuples since it does not have the signing key.

Instead, we propose to use sanitizable signature scheme [39] to ensure data integrity. Sanitizable signature scheme is a special signature scheme in which a signer can delegate a proxy to modify a certain portion of a signed message without invalidating the attached signature. For *RK* and *FK* tuples, the administrator uses sanitizable signature scheme to sign them and delegates the cloud provider to modify the encryption portion of them. In this way, the cloud provider can directly update the *RK* and *FK* tuples without invalidating the attached signatures. Differently for F tuples, the administrator uses general signature scheme to sign them. The reason is that in a revocation, the administrator only delegates the cloud provider to update the encryption portion of F tuples by adding a new encryption layer over it. This operation does not change the encrypted file contents, and thus does not invalidate the attached signatures. As a result, we do not need to use sanitizable signature scheme to sign F tuples.

With our extension, a malicious cloud provider cannot arbitrarily modify *RK*, *FK*, and *F* tuples or generate fake tuples. Also, maliciously modifying the encryption portion of *RK* and *FK* tuples can be easily detected by users.

V. Security analysis

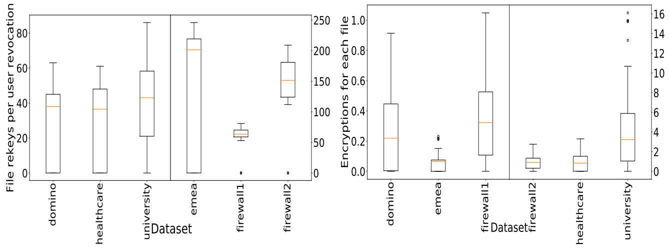
We analyze the security of Crypt-DAC using the access control expressiveness framework known as application-sensitive access control evaluation (ACE) [36]. ACE is a formalized mathematical framework to evaluate how well a candidate access control scheme implement an idealized access control scheme. We 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 ensure 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 ensures that Crypt-DAC is secure. We summarize our result in theorem 1.

Theorem 1: Crypt-DAC implements *RBAC*$_0$ with correctness, AC-preservation, and safety.

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*. The full proof of Theorem 1 can be found in Appendix.

As Crypt-DAC inherits the design of [12], our proof is also similar with the proof of [12]. The difference is our formalization of the query *auth*(*u*, *p*), which asks whether a user *u* has a permission *p*=(f_n, *op*). We formalize *auth*(*u*, (f_n, *Read*)) in a candidate access control system as whether *u* can decrypt the F tuple $\langle F, f_n, c \rangle$. This formalization includes the fact that in a revocation, if the involved F tuples are not timely re-keyed and re-encrypted, the revoked users can still access the files encrypted in the F tuples. With this formalization, the second scheme in [12] has to use **revokeUser** and **write** to implement a revocation operation in *RBAC*$_0$ to achieve correctness. This implementation, however, breaks safety. In specific, to execute a revocation operation in *RBAC*$_0$, the scheme sequentially executes **revokeUser** and **write**. The



(a) File re-encryptions for a user revocation

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

Fig. 7. Summary of simulation results.

execution of **revokeUser** generates an intermediate state, in which a query *auth*(u, (f_n, *Read*)) is *TRUE* for a revoked user u and a file f_n involved in the revocation. This query, however, is *FALSE* in the end state of *RBAC*$_0$ generated by the execution of revocation.

VI. System evaluation

To simplify the presentation, we term the two revocation schemes proposed in [12] as immediate re-encryption (IMre) and deferred re-encryption (DEre), respectively. We also term the revocation scheme proposed in [23] as homomorphic re-encryption (HOre). We implement IMre, DEre, HOre, and Crypt-DAC. In our implementation, we deploy the administrator on a PC which is equipped with a 4-core Intel Xeon 2.0 GHz processor and 16 GB RAM in our lab, and the cloud provider 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++ [37]. 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 derive several critical access control parameters through a simulation of data access control. We use the same simulation framework [38] over the same real-world *RBAC* data sets as in [12] to generate traces of access control actions, and extract the parameters from these traces.

A. Access trace simulation

The simulation framework proposed in [38] 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 data sets 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. We simulate one-month periods in which the administrator of the system behaves.

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 each file in average increases less than 3 within a month.

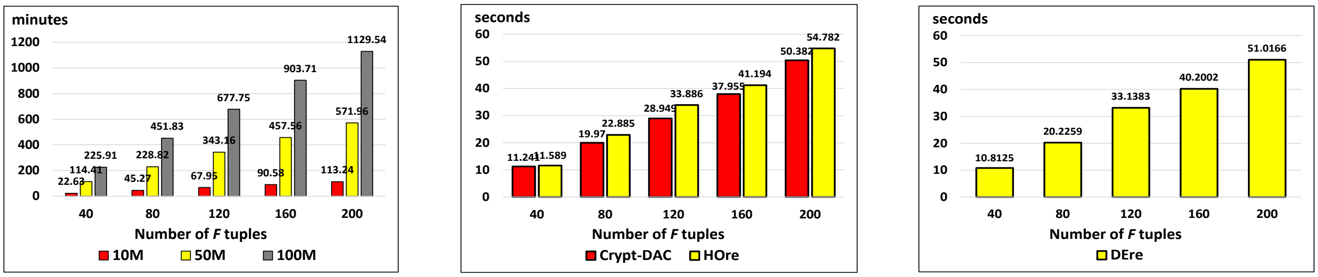
B. End to end experiments

Crypt-DAC only uses lightweight symmetric encryptions to encrypt file data and develops three new strategies to constrain the size of key list and encryption layers for files.

For access revocation, Crypt-DAC uses delegation-aware encryption strategy to delegate the cloud provider to update *RK*/*FK* tuples. As key lists are compactly stored in *FK* tuples, the administrator costs the same overhead to update *RK*/*FK* tuples as previous works. Crypt-DAC also uses adjustable onion encryption strategy to delegate the cloud to update F tuples. As the administrator only sends symmetric keys for the cloud provider to encrypt files, it costs far less overhead to update F tuples than previous works.

For file read/write, Crypt-DAC constrains encryption layers over files to improve the efficiency of file read/write operations. In specific, Crypt-DAC uses the adjustable onion encryption strategy to constrain the encryption layers in revocation operations and delayed re-onion encryption strategy to remove them periodically. The combination of the two strategies ensure that the encryption layer of each file is under an upper bound all the time. More interestingly, the administrator can adjust this upper bound to suit specific application requirements by combining the two strategies in a flexible way.

We show such a way as follows. Suppose each file needs to be re-encrypted at most l times within every fixed-sized period for the purpose of user revocation (3 times within every month in our trace simulation). By combining the two strategies, the administrator can adjust an upper bound $l{\times}k$ of encryption layers over files. Given the delayed re-onion encryption strategy, the administrator sets a parameter α such that α fraction of files will be written at least once within every k periods. For these files, the administrator uses the security mode of the adjustable onion encryption strategy to update them in revocation operations. For the remainder 1-α files, the administrator uses the security mode to update them and turn to the efficiency mode once their encryption layers reach $l \times k$ in revocation operations. In this way, the encryption layer of each file is bounded by $l{\times}k$ all the time. As α and k has a positive relationship, the administrator can adjust the upper bound $l{\times}k$ by adjusting the parameter α.



(a) Performance of IMre at administrator side

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

(c) Performance of DEre at cloud side

Fig. 8. Performance in revocation.

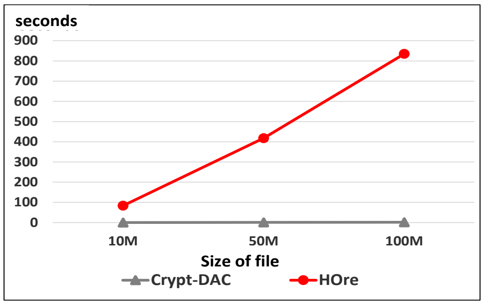
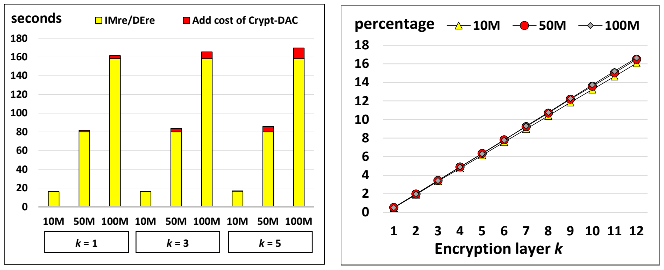


Fig. 9. Computation overhead of Crypt-DAC and HOre at cloud side in revocation.



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

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

Fig. 10. Performance in file reading

Revocation: We evaluate the performance of IMre, HOre, Crypt-DAC and DEre in revoking a user u from a role r. 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-c), 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. Finally, the performance of DEre is similar with HOre and Crypt-DAC and is not affected by the file size. The reason is that DEre delayed the re-encryption of the F tuples to the next users writing to them.

Additionally, in Figure 9, 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 set $l = 3$ and vary k (from 1 to 12) to evaluate the performance trend of Crypt-DAC in a long time (one year). We evaluate Crypt-DAC in the worst case by encrypting files with the upper bound $l \times k$ of encryption layers.

Figure 10 compares the execution time of IMre/DEre and Crypt-DAC in reading operations. Figure 10(a) shows the additional cost of Crypt-DAC comparing with IMre/DEre. We can see that the additional cost is moderate (7.2% when $k = 5$). The reason is that Crypt-DAC uses AES scheme multiple times to encrypt/decrypt files. Figure 10(b) shows the relation of the additional cost with the parameter k. We can see that the additional cost increases slowly as k increases regardless of file size (1.4% each time k increases 1). The reason is that AES is a lightweight cryptographic primitive. Moreover, the administrator can adjust the upper bound $l \times k$ to control this additional cost. Considering the performance advantage of Crypt-DAC in revocation, we believe that this extra cost is acceptable. Table II compares the execution time of HOre and Crypt-DAC in reading operations. 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-

TABLE I
PERFORMANCE OF CRYPT-DAC/HORE IN FILE READING

File size	Crypt-DAC	HOre	Cost of HOre
	Comp/Comm	Comp/Comm	Comp/Total
10 M	1.1/15.8 Sec	98.6/17.7 Sec	82.8/6.8 times
50 M	6.1/81.8 Sec	491.1/90.4 Sec	80.2/6.6 times
100 M	12.3/164.1 Sec	982.3/176.8 Sec	79.8/6.5 times

TABLE II
PERFORMANCE IN FILE WRITING

File size	Others	HOre	Cost of HOre
	Comp/Comm	Comp/Comm	Comp/Total
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

DAC (k = 5). The reason is that HOre uses homomorphic symmetric encryption to encrypt files, and the overhead of which is comparable with public key encryption schemes. Instead, Crypt-DAC uses symmetric key encryption scheme to do so.

Table III 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. The reason is that HOre uses homomorphic symmetric encryption to encrypt files, and 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. [27] explore cryptography to enforce hierarchy access control without considering dynamic policy scenarios. Akl et al. [28] 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. [29] 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. [30] 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. [31] enforce role based access control structure using public-key cryptography, but requires a series of active security mediators. Ferrara et al. [32] 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. [33] 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 [24] 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 [25] 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 [26]. 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 [34], [35] 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. CONCLUSION

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 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 under the honest-but-curious threat model compared with previous schemes.

IX. ACKNOWLEDGEMENT

We acknowledge the support from National Natural Science Foundation of China (No 61602363, No. 61702437, No 61572382), China Postdoctoral Science Foundation (No 2016M590927), Hong Kong ECS under Grant PolyU

252053/15E, Key Project of Natural Science Basic Research Plan in Shaanxi Province of China (No. 2016JZ021), The State Key Laboratory of Cryptology, PO Box 5159, Beijing 100878, China, Doctoral Fund of Ministry of Education of China (No. 20130203110004), China 111 Project (No. B16037), and the Fundamental Research Funds for the Central Universities (No XJS16001, No JB161509).

References

[1] J. Bethencourt, A. Sahai, and B. Waters, *Ciphertext-policy attribute based encryption*, in IEEE S&P, 2007.

[2] X. Wang, Y. Qi, and Z. Wang, *Design and Implementation of SecPod: A Framework for Virtualization-based Security Systems*, IEEE Transactions on Dependable and Secure Computing, vol. 16, no. 1, 2019.

[3] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, *AppSec: A Safe Execution Environment for Security Sensitive Applications*, in ACM VEE, 2015.

[4] V. Goyal, A. Jain, O. Pandey, and A. Sahai, *Bounded ciphertext policy attribute based encryption*, in ICALP, 2008.

[5] V. Goyal, O. Pandey, A. Sahai, and B. Waters, *Attribute-based encryption for fine-grained access control of encrypted data*, in ACM CCS, 2006.

[6] J. Katz, A. Sahai, and B. Waters, *Predicate encryption supporting disjunctions, polynomial equations, and inner products*, in EUROCRYPT, 2008.

[7] S. Muller and S. Katzenbeisser, *Hiding the policy in cryptographic access control*, in STM, 2011.

[8] R. Ostrovsky, A. Sahai, and B. Waters, *Attribute-based encryption with non-monotonic access structures*, in ACM CCS, 2007.

[9] A. Sahai, and B. Waters, *Fuzzy identity-based encryption*, in EUROCRYPT, 2005.

[10] T. Ring, *Cloud computing hit by celebgate*, http://www.scmagazineuk.com/cloud-computing-hit-by-celebgate/article/370815/, 2015.

[11] X. Jin, R. Krishnan, and R. S. Sandhu, *A unified attribute-based access control model covering DAC, MAC and RBAC*, in DDBSec, 2012.

[12] 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.

[13] R. S. Sandhu, *Rationale for the RBAC96 family of access control models*, in ACM Workshop on RBAC, 1995.

[14] T. Jiang, X. Chen, Q. Wu, J. Ma, W. Susilo, and W. Lou, *Secure and Efficient Cloud Data Deduplication With Randomized Tag*, IEEE Trasactions on Information Forensics and Security, vol. 12, no. 3, 2017.

[15] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, *Plutus: Scalable Secure File Sharing on Untrusted Storage*, in USENIX FAST, 2003.

[16] J. Wang, X. Chen, X. Huang, I. You, and Y. Xiang, *Verifiable Auditing for Outsourced Database in Cloud Computing*, IEEE Transactions on Computers, vol. 64, no. 11, 2015.

[17] J. Wang, X. Chen, J. Li, J. Zhao, and J. Shen, *Towards achieving flexible and verifiable search for outsourced database in cloud computing*, Future Generation Computer Systems, vol. 67, 2017.

[18] X. Chen, J. Li, X. Huang, J. Ma, and W. Lou, *New Publicly Verifiable Databases with Efficient Updates*, IEEE Transactions on Dependable and Secure Computing, vol. 12, no. 5, 2015.

[19] T. Jiang, X. Chen, and J. Ma, *Public Integrity Auditing for Shared Dynamic Cloud Data with Group User Revocation*, IEEE Transactions on Computers, vol. 65, no. 8, 2016.

[20] D. Boneh and M. Franklin, *Identity-based encryption from the Weil pairing*, SIAM Journal on Computing, vol. 32, no. 3, 2003.

[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] D. Boneh, K. Lewi, H. Montgomery, and A. Raghu Raghunathan, *Key homomorphic PRFs and their applications*, in CRYPTO, 2013.

[25] M. Maffei, G. Malavolta, M. Reinert, and D. Schroder, *Privacy and access control for outsourced personal records*, in IEEE S&P, 2015.

[26] 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 USENIX FAST, 2013.

[27] E. Gudes, *The Design of a Cryptography Based Secure File System*, IEEE Transactions on Software Engineering, vol. 6, no. 5, 1980.

[28] S. G. Akl and P. D. Taylor, *Cryptographic solution to a problem of access control in a hierarchy*, ACM Transactions on Computer Systems, vol. 1, no. 3, 1983.

[29] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken, *Dynamic and efficient key management for access hierarchies*, ACM Transactions on Information and System Security, vol. 12, no. 3, 2009.

[30] L. Ibraimi, *Cryptographically enforced distributed data access control*, Ph.D. dissertation, University of Twente, 2011.

[31] D. Nali, C. M. Adams, and A. Miri, *Using mediated identity-based cryptography to support role-based access control*, in ISC 2004, 2004.

[32] A. L. Ferrara, G. Fuchsbauer, and B. Warinschi, *Cryptographically enforced RBAC*, in CSF, 2013.

[33] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters, *Secure attribute-based systems*, in ACM CCS, 2006.

[34] 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.

[35] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, *Encryption policies for regulating access to outsourced data*, ACM Transactions on Database Systems, vol. 35, no. 2, 2010.

[36] 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.

[37] https://www.cryptopp.com/

[38] W. C. Garrison III, A. J. Lee, and T. L. Hinrichs, *An actor-based, application-aware access control evaluation framework*, in SACMAT, 2014.

[39] G. Ateniese, D. H. Chou, B. Medeiros, and G. Tsudik, *Sanitizable Signatures*, in ESORICS, 2005.

Saiyu Qi received the B.S. degree in computer science and technology from Xian Jiaotong University, Xian, China, in 2008, and the Ph.D. degree in computer science and engineering from Hong Kong University of Science and Technology, Hong Kong, in 2014. He is currently an Assistant Professor with the School of Cyber Engineering, Xidian University, China. His research interests include applied cryptography, cloud security, distributed systems, and pervasive computing.

Yuanqing Zheng received PhD degree from the School of Computer Engineering in Nanyang Technological University in 2014. Before that he received the B.S. degree in Electrical Engineering and the M.E. degree in Communication and Information System from Beijing Normal University, Beijing, China, in 2007 and 2010 respectively. He is currently an assistant professor with the Department of Computing in the Hong Kong Polytechnic University. His research interest includes human centered computing, mobile and wireless computing, RFID systems, etc. He is a member of IEEE and ACM.

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. We give the formal definitions of the three security properties as follows.

Definition 1 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*(σ(*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 2 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 3 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)

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*(σ(*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:*

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

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:

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

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:

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

- **revokeP:** If l is an instance of *revokeP*(r, p) with $p = \langle f_n, op\rangle$, then $x' = x \backslash PA(r, p)$. 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
  $\sigma'(x') = \sigma'(x \setminus PA(r, p))$
  $=\sigma'(\text{next}(x', revokeP(r, p)))$
  $=\sigma'(x) \setminus \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, RW), c, sig\rangle) \mid (r, c, sig) \in T\}$
  $\cup\{FS(\langle FK, (r, v_r),(f_n, v_{f_n}, Read), c, sig\rangle) \mid (r, c, sig) \in T\}$
  $=\textit{terminal}(\sigma'(x), \alpha(\sigma'(x), l))$.

  – 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*.