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

Saiyu Qi*, Yichen Li†, Yuanqing Zheng‡ and Yong Qi† *School of Cyber Engineering, Xidian University, China †Department of Computer Science and Technology, Xi'an Jiaotong University, China ‡Department of Computing, The Hong Kong Polytechnic University, Hong Kong
email: syqi@connect.ust.hk, 245703424@qq.com, csyqzheng@comp.polyu.edu.hk, qiy@xjtu.edu.cn

Abstract

Enabling access controls for data hosted on untrusted cloud is attractive for many users and organizations. Recently, many works have been proposed to use advanced cryptographic primitives such as identity-based encryption, attribute-based encryption, and predicate encryption to enforce data access control on the potentially untrusted cloud. 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 uses *delegation-aware encryption* and *symmetric onion encryption*, which enable access revocation to be executed at the cloud side in a secure manner. Crypt-DAC further uses *lazy de-onion encryption* to facilitate file access without incurring obvious overhead. 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 customer side, and security, as it immediately revoke access permissions, while operating under a similar threat model of previous comparable systems. We use formalization framework and system implementation to demonstrate the security and efficiency of our construction.

I. Introduction

With the considerable advancements in cloud computing, users and organizations are finding increasingly appealing to rely on cloud services for storing file data and sharing them with others. In such contexts, a critical issue is how to enforce data access control on the potentially untrusted cloud. Indeed, commercial cloud providers such as Google, Microsoft, Apple, and Amazon provide abundant cloud based services, ranging from small-scale, personal services to large-scale, industrial services. However, the near-constant media coverage of data breaches, such as releases of private photos [8], has raised concerns regarding the privacy and access control of cloud-hosted data. Therefore, despite its economic and ease-of-use benefits, outsourcing file data to the cloud raises new issues regarding the enforcement of data access control and confidentiality

In response to these security issues, numerous works [1]–[7] have been proposed to support access control on the untrusted cloud by leveraging cryptographic primitives. Advanced cryptographic primitives are well-suited for enforcing many access control paradigms. For example, attribute-based encryption (ABE) [9] is a natural fit for enforcing attribute-based access control (ABAC) model [13]. However, much of the literature concerns static scenarios in which access control policies are rarely changed. Such scenarios are not representative of real-world systems, and previous works oversimplify issues incurred by revocation that can carry substantial practical overheads. Although the development of key revocation [10] and delegated re-encryption [11], [12] support a level of dynamism for current advanced cryptographic primitives, these techniques are not compatible with hybrid encryption, which is necessary from an efficiency perspective.

In IEEE S&P 2016, Garrison et al. [14] explored the practical implications of using these types of cryptographic primitives to enforce a type of prevalent access control model: role based access control model ($RBAC_0$) [15]. They concluded that the cryptographic enforcement of dynamic access control on untrusted cloud incurs prohibitive costs in practice. In specific, dynamic change of access policy includes two aspects: assignment of access permission and revocation of access permission. Assigning an access permission is easy as it is sufficient to give the newly authorized user access to the key. On the other hand, revoking an access permission is a completely different problem [14]. there are two challenging issues in the revocation of access permission.

The first challenge is the prohibitive overhead in policy data updating. Policy data updating entails, for an access control administrator, generating new keys, creating new policy data to encrypt the keys, and uploading the policy data to re-distribute the keys to the users who still hold permissions. Due to multi-to-one property of access permission relations among users, roles and files, the administrator needs to create large amounts of policy data to revoke the permission of even a single user, incurring prohibitive overhead in terms of public encryption and signing operations.

The second challenge is the prohibitive overhead in file data updating. File data updating entails, also for the administrator, downloading the file data, decrypting it, re-encrypting it with the new keys, and re-uploading the file data. This immediate re-encryption strategy incurs prohibitive overhead due to decrypting/re-encrypting and downloading/ re-uploading file data, which becomes even more expensive in emerging big data contexts. To reduce the overhead, Garrison et al. [14] suggest to use a deferred re-encryption strategy, in which file data is re-encrypted by the next users to write to it. This solution, however, comes with a security penalty as the success of revocation is delayed to the next writing. As a result, there is a tension between efficiency and security in access revocation.

To address these two challenges, we present Crypt-DAC, a cryptographically enforced dynamic access control system on untrusted cloud. The central goal of Crypt-DAC is to balance the tension by avoiding expensive updating of policy and file data at the administrator side, while enabling immediate access permission revocations. Crypt-DAC addresses the two challenges using three key techniques:

First, we delegate the cloud to update the policy data in a privacy and verifiability-preserving manner. To this end, Crypt-DAC uses a *delegation-aware encryption strategy*, which is built on a combination of proxy re-encryption scheme [16] and sanitizable signature scheme [20] in a novel way. By adapting delegation-aware encryption strategy, Crypt-DAC encrypts and signs policy data in a way that allows the cloud to update policy data without decryption.

Second, to avoid executing expensive re-encryptions of file data at the administrator side, Crypt-DAC proposes symmetric onion encryption strategy, which is a novel way to encrypt a file through a symmetric key list in a onion manner. By adapting symmetric onion encryption, the administrator only needs to request the cloud to encrypt the file data with a new layer of symmetric encryption and update the encrypted symmetric key list accordingly.

With the symmetric onion encryption deployed, a problem is that as the revocations continuously happen, the encryption layers of the involved files and the length of the corresponding symmetric key list increases, incurring additional overhead in file reading. To remove this overhead without incurring obvious overhead, Crypt-DAC uses a lazy de-onion encryption strategy, in which the next user to write to the file encrypts the writing content by a new symmetric key list containing a single key, and updates the symmetric key list accordingly. We show that lazy de-onion encryption facilitates file reading operations with moderate overhead.

We compare the security and efficiency of Crypt-DAC with two versions of the construction in [14]: IMre, which adopts immediate re-encryption strategy, and DEre, which adopts deferred re-encryption strategy. Crypt-DAC operates under the similar threat model of IMre. Regarding security, we use an access control expressiveness framework [43] to prove that IMre and Crypt-DAC achieve the same security properties, while DEre cannot achieve safety. Regarding efficiency, we implement IMre, DEre and Crypt-DAC on Alicloud and show that the performance of Crypt-DAC is comparable with DEre.

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

II. Related work

Cryptography provides a natural way to enforce access controls on untrusted cloud storage. Encryption schemes, ranging from basic schemes including symmetric and public encryptions, to more advanced schemes including identity-based encryption [23], attribute-based encryption [7], and predicate encryption [26], [27], support a number of access control functionalities. At a high level, these encryption schemes bound data to a policy through encryption, and only users who have secret keys satisfying the policy can decrypt. There has been significant work on using cryptography to enforce access control.

hierarchy access control: Gudes et al. [34] investigate how to enforce hierarchy access control structure using cryptography, but does not address issue of policy updates. Akl et al. [35] propose a key assignment scheme to simply key management in hierarchical access control policy. Again, this work does not consider policy update issues. Later work in key hierarchies by Atallah et al. [36] proposes 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, and the cost of such an operation is not discussed.

Role based access control: Ibraimi et al. [39] 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. [40] enforce role based access control structure using public-key cryptography, but requires a series of active security mediators. Ferrara et al. [41]

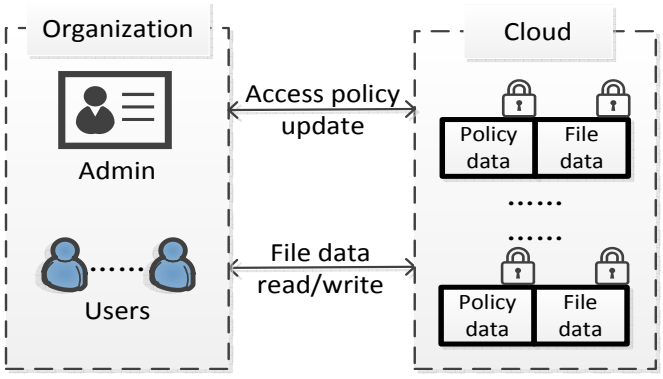


Fig. 1: Cloud enabled data access control.

define a secure model to formally prove the security of a cryptographically-enforced RBAC system. They next show that an ABE-based construction is secure under such model. However, their work is a theoretical implementation without evaluation.

Attribute based access control: Pirretti et al. [42] propose an optimized ABE-based access control for distributed file systems and social networks, but their construction does not consider dynamic revocation. Sieve [28] 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 human-understandable access policies and homomorphic symmetric encryption [31] 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. A problem of homomorphic symmetric encryption, however, is that its efficiency is comparable with public key encryption schemes. Using homomorphic symmetric encryption to encrypt large files incurs prohibitive computation overhead.

Access matrix: GORAM [32] allows a data owner to enforce an access matrix for a list of users to regulate who can access its data stored on the cloud. GORAM provides strong data privacy in two folds. First, user access patterns are hidden from the cloud by using ORAM techniques [33]. Second, policy attributes are hidden from the cloud by using attribute-hiding predicate encryption [26], [27]. The usage of these advanced cryptographic techniques, however, incurs additional performance cost in data communication, encryption and decryption. Also GORAM does not support dynamic policy update. A data owner has to determine a priori the maximum number of users in the list; and if this list changes, the owner must re-build its data.

III. Background and assumptions

In this section, we first define the system and threat models in which we consider this problem. We then specify the $RBAC_0$ model that we propose to enforce. Finally we define the classes of cryptographic primitives that will be used in our design.

A. System model

The system model that we consider is depicted in Figure 1. The system consists of three types of entities: an access control administrator, a large number of users, and a cloud provider. Our system model describes a scenario in which a single cloud provider is contracted by an organization. This is analogous to companies contracting with providers like Microsoft to outsource enterprise storage. The cloud provider is contracted to manage the data storage requirements of an organization. The data includes file data of users in the organization, as well as policy data protecting these files. The access control administrator is responsible for managing access policies of the file data. In specific, the administrator controls the assignment of access permissions, which entails the creation, distribution, and revocation of cryptographic keys used to encrypt files in a role-based manner. Policy data is used to distribute keys and is stored in encryptions on the cloud provider. Users may download any policy/file data from the cloud, but may decrypt and read only the files for which they have appropriate permissions (role-based keys). File data is encrypted prior to being uploaded to the cloud provider. Finally, we assume that all parties can communicate via pairwise authenticated and private channels (e.g., SSL/TLS tunnels).

B. Threat model

In our threat model, the cloud provider is not trusted to protect the privacy and enforce read access permissions of the file data. Instead, read access permissions should be enforced by the access policies defined by the administrator. This is the reason why we need to use cryptography to enforce file access control.

On the other hand, we want to put some reasonable trust assumptions on the cloud provider as several recent constructions [14], [28] to simplify our system design. We explain these assumptions as follows.

First, the cloud provider behaves honestly in access policy updating procedure. This assumption is critical to design efficient yet secure access control system on the cloud. From the efficiency aspect, this assumption enables us to leverage the computing ability of the cloud provider, which is one of the most attractive advantages of cloud computing paradigm. From the security aspect, as policy/file data is stored by the cloud provider, we have to rely on it to honestly replace existing policy/file data with their updated versions to complete access policy updating. For instance, in IMre, although the administrator re-encrypts file data by itself in a revocation, the administrator still needs to rely on the cloud provider to honestly replace existing file data with its re-encrypted version. Otherwise, the cloud provider can just retain a copy of the existing file data for revoked users to continuously access.

Second, the cloud provider is trusted to enforce write access permissions that validates write privileges of users prior to file updates. As file data is stored at the cloud, we have to rely on the cloud provider to execute writing operations submitted by users over the data. To regulate the behaviours of the cloud provider in this procedure, several constructions [29], [30], [37], [38] have been proposed, which can be integrated into our system. As a result, we use this assumption to simplify our system design.

Third, the cloud provider is trusted to ensure availability of policy/file data. Providing data availability is a basis for cloud providers to attract customers to use their services. Current commercial cloud providers promise data availability for their customers through their Service Level Agreements (SLAs). For instance, S3's SLA [21] and Azure's SLA [22] guarantee availability: if availability falls below 99.9%, customers are reimbursed a contractual sum of money. As a result, we use this assumption to simplify our system design.

We believe that our threat model is a fundamental start point to investigate practical cryptographically enforced access control system. Currently, how to design new secure access control system with reduced trust assumptions on the cloud provider while preserving practicality is still an open question.

C. Role based access control

We describe a role-based access control model named ($RBAC_0$) [15], given the prevalence of role based access control systems 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

The access permissions in $RBAC_0$ model renders a multi-to-one property among users, roles, and objects: multiple users are member of a role and multiple roles have permissions to an object.

D. Cryptographic tools

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

Proxy re-encryption scheme: We use public-key encryption scheme (Gen^{Pub}, Enc^{Pub}, Dec^{Pub}, $\oplus$) supporting proxy re-encryption operation.

$Gen^{Pub}(1^n) \rightarrow (ek, dk)$: on input 1^n, this algorithm outputs an encryption key pair (ek, dk).

$Enc^{Pub}(ek, m) \rightarrow c$: on input an encryption key ek and a message m, this algorithm outputs a ciphertext c.

$Dec^{Pub}(dk, c) \rightarrow m$: on input a decryption key dk and a ciphertext c, this algorithm outputs a message m.

$\oplus$ (*Proxy re-encryption*): Given two encryption key pairs: (ek_1, dk_1) and (ek_2, dk_2), this algorithm outputs a re-key rek. With rek, a proxy can transform a ciphertext by ek_1 to a ciphertext by ek_2 without decryption.

Proxy re-encryption scheme can support different types of proxy functions. We describe an instantiation of proxy re-encryption scheme [16] supporting unidirectional and multi-use proxy re-encryption, which is suitable for our construction.

The scheme is based on El Gamal encryption scheme [19]. let $\mathcal{G}$ be a polynomial-time algorithm that, on input 1^n, outputs a description of a cyclic group G, its order q (with $|q| = n$), and a generator g. On input 1^n, the key generation algorithm runs $\mathcal{G}(1^n)$ to obtain (G, q, g). The algorithm then chooses a random $x \leftarrow Z_q$ and computes h: = g^x. The encryption key *ek* is h and the decryption key *dk* is x. On input an encryption key *ek* = h and a message $m \in G$, the encryption algorithm chooses a random $y \leftarrow Z_q$ and outputs the ciphertext c:

$$c = (g^y, h^y m)$$

On input a decryption key *dk* = x and a ciphertext $c = (g^y, h^y m)$, the decryption algorithm outputs:

$$m = h^y m / g^{yx} = h^y m / h^y$$

Given two encryption key pairs: ($ek_1 = h_1$, $dk_1 = x_1$) and ($ek_2 = h_2$, $dk_2 = x_2$), the proxy algorithm computes the re-key *rek* = $x_2 - x_1$, the proxy re-encryption operation is then:

$$\begin{aligned} & Enc^{Pub}_{ek_1}(m) \oplus rek \\ & = (g^y, h_1^y m) \oplus (x_2 - x_1) = (g^y, h_1^y m g^{y(x_2 - x_1)}) \\ & = (g^y, g^{yx_2} m) = (g^y, h_2^y m) \\ & = Enc^{Pub}_{ek_2}(m) \end{aligned}$$

Sanitizable signature scheme: Sanitizable signature [20] is a special signature scheme (*Gen*^{Sign}, *Sign*, *Verify*, *Sanitize*) that a signer can delegate a proxy to modify a certain portion of a signed message. We next describe the algorithms of this scheme:

Gen$^{Sign}(1^n) \rightarrow (sk^{sign}, vk^{sign})$, (sk^{san}, vk^{san}): On input a security parameter 1^n, this algorithm outputs two types of key pairs: a signing key pair (sk^{sign}, vk^{sign}) for a signer and a sanitization key pair (sk^{san}, vk^{san}) for a proxy.

Sign(*m*, sk^{sign}, vk^{san}) $\rightarrow$ *sign*: On input a message *m*, a signing key sk^{sign}, a sanitization verification key vk^{san}, this algorithm produces a signature *sign*. During the signing procedure, the signer can choose which part of *m* can be changed by the proxy owning the sanitization signing key sk_{san}.

Verify(*m*, *sign*, vk^{sign}, vk^{san}) $\rightarrow$ *Accept/Reject*: On input a message *m*, a possibly valid signature *sign* on *m*, a verification key vk^{sign} and a sanitization verification key vk^{san}, outputs *Accept* or *Reject*.

Sanitize(*m*, *sign*, vk^{sign}, sk^{san}, *m'*) $\rightarrow$ *sign'*: On input a message *m*, a signature *sign* on *m* under signing key sk^{sign}, a verification key vk^{sign}, a sanitization signing key sk^{san}, and a new message *m'*, produces a new signature *sign'* on *m'*.

IV. Design principle

In this section, we begin with the construction of [14] for 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. Previous design

We now overview the construction of [14], which allow us to highlight a variety of issues for revocation. We simplify the construction with the main design principles preserved for clarity and ease of understanding.

Registration: When a user *u* enters the system, it carries out an initial registration process with the administrator. In the process, *u* generates two key pairs: enk_u=(ek_u, dk_u) $\leftarrow$ *Gen*$^{Pub}(1^n)$ and sik_u=(sk_u, vk_u) $\leftarrow$ *Gen*$^{Sign}(1^n)$, and submits (ek_u, vk_u) to the administrator for storage.

Role Administration: For each role *r*, the administrator generates two key pairs: enk_{r,v_r} = (ek_{r,v_r}, dk_{r,v_r}) $\leftarrow$ *Gen*$^{Pub}(1^n)$ and sik_{r,v_r} =(sk_{r,v_r}, vk_{r,v_r}) $\leftarrow$ *Gen*$^{Sign}(1^n)$ for *r*. v_r is a version number representing the current version of the key pairs. initially, v_r=1. For each user u that is a member of r (i.e., for each $(u, r) \in UR$ in the $RBAC_0$ state), the administrator creates and uploads a role key (*RK*) tuple:

$$\langle RK, u, (r, v_r), Enc^{Pub}_{ek_u}(dk_{r,v_r}, sk_{r,v_r}), sign_{SU} \rangle$$

This tuple provides *u* with cryptographically-enforced access to the decryption key dk_{r,v_r} and signing key sk_{r,v_r} of r. *RK* is a label indicating that this is a role key tuple; and $sign_{SU}$ is a signature over the whole *RK* tuple signed by the administrator (as a super user).

File Administration: When a user *u* wants to upload a file *f*, *u* creates and uploads a file (*F*) tuple:

$$\langle F,\ f_n,\ v_{f_n},\ Enc_k^{Sym}(f),\ sign_u \rangle$$

This tuple encrypts *f* by a symmetric file key $k \leftarrow KeyGen^{Sym}(1^n)$. *F* is a label indicating that this is a file tuple; f_n is the file name of *f*; v_{f_n} is a version number representing the current version of *k*; and $sign_u$ is a signature over the whole *F* tuple signed by the signing key sk_u of *u*. *u* also uploads a file key (*FK*) tuple:

$$\langle FK,\ SU,\ (f_n,\ v_{f_n},\ RW),\ Enc_{ek_{SU}}^{Pub}(k),\ sign_u \rangle$$

This tuple provides the administrator with cryptographically-enforced access to the file key *k* for *f*. *FK* is a label indicating that this is a file key tuple; and *RW* means that the administrator can read/write the *FK* tuple.

Next, for each role *r* with permission to *f* (i.e., for each $(r, (f, op)) \in PA$ in the $RBAC_0$ state), the administrator creates and uploads a file key (*FK*) tuple:

$$\langle FK,\ (r,\ v_r),\ (f_n,\ v_{f_n},\ op),\ Enc_{ek_{r,v_r}}^{Pub}(k),\ sign_{SU} \rangle$$

This tuple provides *r* with cryptographically-enforced access to the file key *k* for *f*. *FK* is a label indicating that this is a file key tuple; *op* is the permitted operation: either *Read* or *RW*; and $sign_{SU}$ is a signature over the whole *FK* tuple signed by the administrator.

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* first downloads an *RK* tuple for the role *r* to decrypt the decryption key dk_{r,v_r} by dk_u. *u* then downloads an *FK* tuple for the file *f* to decrypt the file key *k* by dk_{r,v_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: $\langle F,\ f_n,\ v_{f_n},\ Enc_k^{Sym}(f'),\ sign_{r,v_r} \rangle$ to replace the existing *F* tuple for *f*. 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) the uploaded *F* tuple contains a valid signature signed by *r*. 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.

Revoking a permission of a role entails two tasks: (1) creating new *FK* tuples; and (2) re-keying and re-encrypting *F* tuples. Suppose the revoked role has permission to *m* files. In the first task, for each of the *m* files, the administrator creates a new file key $k' \leftarrow KeyGen^{Sym}(1^n)$ and uploads new *FK* tuples encrypting *k'* for each role *r* whose access to the file has not been revoked:

$$\langle FK,\ (r,\ v_r),\ (f_n,\ v_{f_n}+1,\ op),\ Enc_{ek_{r,v_r}}^{Pub}(k'),\ sign_{SU} \rangle$$

The administrator adopts a deferred re-encryption strategy to complete the second task for efficiency consideration. That is, the next users writing to the *F* tuples of the *m* files re-encrypt the *F* tuples with the new file keys. When a user *u* writes to one of the *F* tuples, *u* decrypts the new file key *k'* from the new *FK* tuple, encrypts the writing content by *k'*, and uploads a new *F* tuple to the cloud provider:

$$\langle F,\ f_n,\ v_{f_n}+1,\ Enc_{k'}^{Sym}(f),\ sign_{r,v_r} \rangle$$

Removing a user *u* from a role *r* entails (1) creating new *RK* tuples and *FK* tuples; and (2) re-keying and re-encrypting *F* tuples. The procedure of the first task involves three sub steps:

- The administrator informs the cloud to delete *u*'s *RK* tuple for *r*. The administrator then generates new key pairs enk_{r,v_r+1} and sik_{r,v_r+1} for *r* and uploads new *RK* tuples encrypting dk_{r,v_r+1} and sk_{r,v_r+1} for all the users *u'* remaining in *r* to replace their existing *RK* tuples:
  $$\langle RK,\ u',\ (r,\ v_r+1),\ Enc_{ek_{u'}}^{Pub}(dk_{r,v_r+1},\ sk_{r,v_r+1}),\ sign_{SU} \rangle$$
  After this step, all the users remaining in *r* can access the new keys dk_{r,v_r+1} and sk_{r,v_r+1} through the uploaded *RK* tuples.
- Suppose *r* has permission to *m* files, i.e., there exists *m* *FK* tuples assigning *r* with permissions to the *m* files. For each of the *FK* tuples, the administrator downloads it:

$$\langle FK,\ (r,\ v_r),\ (f_n,\ v_{f_n},\ op),\ Enc^{Pub}_{ek_{r,v_r}}(k),\ sign_{SU}\rangle$$

decrypts the file key k from the tuple, re-encrypts k by the new encryption key ek_{r,v_r+1} of role r, and uploads a new *FK* tuple to the cloud to replace the existing *FK* tuple:

$$\langle FK,\ (r,\ v_r+1),\ (f_n,\ v_{f_n},\ op),\ Enc^{Pub}_{ek_{r,v_r+1}}(k),\ sign_{SU}\rangle$$

After this step, all the users remaining in r can access the file keys of the m files through the uploaded *FK* tuples.

- For each of the m files, the administrator creates a new file key $k' \leftarrow Gen^{Sym}()$ for the file and uploads new *FK* tuples encrypting k' for each role r' (including r) with permission to the file.

  For role r': $\langle FK,\ (r',\ v_{r'}),\ (f_n,\ v_{f_n}+1,\ op),\ Enc^{Pub}_{ek_{r',v_{r'}}}(k'),\ sign_{SU}\rangle$

  For role r: $\langle FK,\ (r,\ v_r+1),\ (f_n,\ v_{f_n}+1,\ op),\ Enc^{Pub}_{ek_{r,v_r+1}}(k'),\ sign_{SU}\rangle$

  After this step, the next users writing to the m files can access the new file keys from the new *FK* tuples and use them to re-encrypt the m files.

Similarly, the administrator also adopts the deferred re-encryption strategy to complete the second task. Later, when a user writes to one of the m files, it re-encrypts the *F* tuple by k' and uploads the new *F* tuple to the cloud:

$$\langle F,\ f_n,\ v_{f_n}{+}1,\ Enc^{Sym}_{k'}(f),\ sign_{r,v_r}\rangle$$

B. Design issues for revocation

Given the above construction, the authors [14] pointed out that the revocation in this construction is not suitable for realistic dynamic access control scenario due to its prohibitive overhead. The overhead is stemmed from two aspects.

Policy data updating: First, creating new *RK* and *FK* tuples in revocation operations incurs prohibitive overhead at the administrator side. Due to the multi-to-one property of access permission relations among users, roles, and objects in $RBAC_0$ model, the administrator has to create and upload new *RK* tuples and *FK* tuples for large numbers of unrevoked users and roles even revoking the membership of a single user from a role. The incurred overhead includes large numbers of public encryption and signing operations, and bandwidth consumptions. For example, suppose the administrator needs to revoke the membership of a user u from a role r with 100 users remaining in r. The administrator needs to create 100 new *RK* tuples encrypting new keys of r for these users. Next, suppose that r has permission to 100 files, and 10 roles have permissions to each of these files. For each of these files, the administrator needs to create 10 new *FK* tuples encrypting a new symmetric key for the file. In summary, the administrator needs to create 100 *RK* tuples and 1000 *FK* tuples to revoke the membership of a single user from a role.

Such a prohibitive overhead is also validated through simulation of realistic dynamic workloads over real data sets. As shown in [14], removing a single user from a role in a moderately-sized organization can require hundreds or thousands of public encryption operations.

File data updating: Second, re-keying and re-encrypting *F* tuples incurs prohibitive overhead at the administrator side. After updating the policy data, the administrator still needs to re-key and re-encrypt *F* tuples, as it ensures that users who have been revoked permission to access the *F* tuples cannot later read its contents. Here, we distinguish two versions of the construction in [14]: IMre, which adopts immediate re-encryption strategy, and DEre, which adopts deferred re-encryption strategy. In IMre, the administrator immediately re-encrypt the *F* tuples to complete the revocation. This version, however, comes with a potentially high overhead at the administrator side in the sense that many *F* tuples may be re-keyed due to changes to some role, and the administrator needs to download, decrypt, re-encrypt, and upload these *F* tuples. On the other hand, DEre relies on next users writing to the *F* tuples to re-encrypt the *F* tuples, removing the overhead of re-encrypting *F* tuples at the administrator side. This version, 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. As a result, a challenging issue is how to design a file data updating mechanism with the security of IMre and the efficiency of DEre.

Besides, the authors [14] also noticed that proxy re-encryptions [16]–[18] are not suitable for use in this scenario. At a first glance, proxy re-encryption schemes allow us to remove the reliance on deferred re-encryption by delegating the cloud provider to update encryptions in revocation operations. This would be done by generating new keys of a role, using proxy re-encryption to transfer the *RK* and *FK* tuples from the old role keys to the new role keys. The critical issue, here, is that such schemes are not compatible with hybrid encryption, which is essential for efficiency consideration. In specific, hybrid encryption relieves the administrator from using expensive public key encryption to encrypt large files by using *FK* tuples to publicly encrypt small symmetric keys, and using *F* tuples

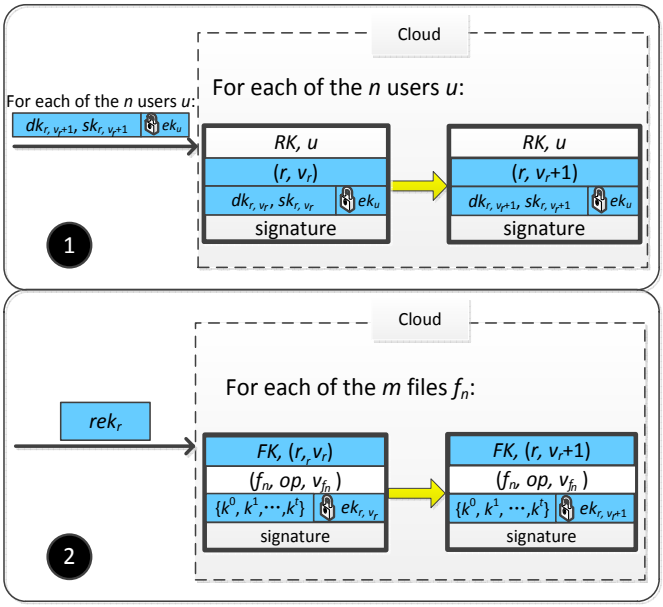


Fig. 2: Delegation-aware encryption overview.

to symmetrically encrypt large files by the symmetric keys. As a result, using proxy re-encryption on the *RK* and *FK* tuples and not on the *F* tuples would allow users to cache the small symmetric keys during file access, and continuously access large files of which their access permissions have been revoked.

C. Our design

To alleviate the overhead of access revocation, a potential solution is to replace the symmetric encryption scheme in the hybrid encryption with the homomorphic encryption scheme [31]. This scheme enables the administrator to delegate the cloud to transfer *F* tuples from old file keys to new file keys without decryption, removing the overhead of re-encrypting *F* tuples. This idea has been used in [32] to develop an attribute based access control system with efficient revocation. The problem of homomorphic symmetric encryption, however, is that its cost is comparable with public key encryption schemes. Using homomorphic symmetric encryption to encrypt/decrypt large files incurs prohibitive computation overhead.

Instead, Crypt-DAC develops new techniques using lightweight symmetric encryption scheme. In specific, Crypt-DAC uses delegation-aware encryption and symmetric onion encryption as a whole to support efficient and immediate revocation, achieving the best of both efficiency and security. Also, Crypt-DAC uses lazy de-onion encryption strategy to remove the overhead incurred by the symmetric onion encryption in file access without incurring obvious overhead. In the following, we use an example of revoking the membership of a user *u'* from a role *r* to show how the three techniques work.

Delegation-aware encryption: To revoke the membership of a user *u'* from a role *r*, The administrator first needs to update the *RK* and *FK* tuples involving *r*. Delegation-aware encryption enables the administrator to delegate the cloud provider to update *RK* and *FK* tuples in a privacy and verifiability-preserving manner. The administrator only needs to send short messages for the cloud to do so instead of creating and uploading new *RK* and *FK* tuples by itself. The observation is that proxy re-encryption scheme [16] is compatible with our symmetric onion encryption strategy, which enables us to use proxy re-encryption to alleviate the overhead at the administrator side. We further use sanitizable signature scheme [20] to preserve the verifiability of the *RK* and *FK* tuples while enabling the cloud to update them.

To revoke *u'* from *r*, the administrator updates the encryption and signing key pairs of *r* as: $(enk_{r,v_r}, sik_{r,v_r}) \rightarrow (enk_{r,v_r+1}, sik_{r,v_r+1})$. 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 $Enc^{Pub}_{ek_u}(dk_{r,v_r+1}, sk_{r,v_r+1})$ by the encryption key ek_u of *u*. With this encryption, the cloud updates the *RK* tuple as:

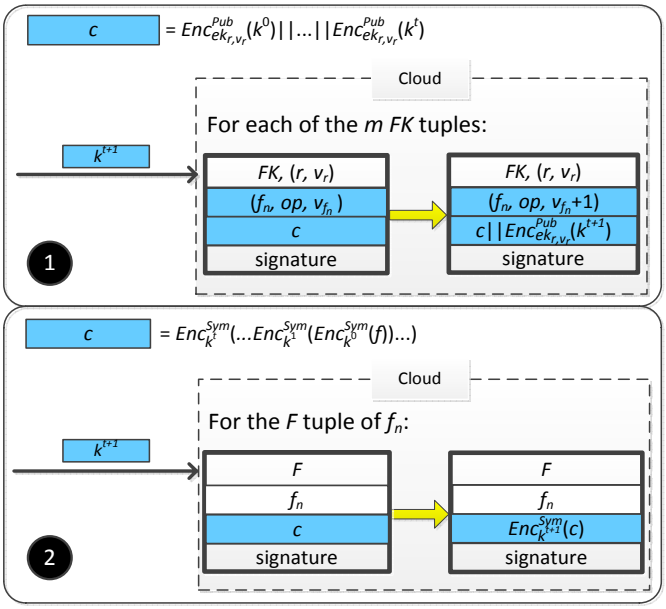


Fig. 3: Symmetric onion encryption overview.

$$\langle RK,\ u,\ (r,\ v_r{+}1),\ Enc^{Pub}_{ek_u}(dk_{r,v_r+1},\ sk_{r,v_r+1}),\ sign_{SU}\rangle$$

The updated *RK* tuple encrypts the new decryption and signing keys (dk_{r,v_r+1}, sk_{r,v_r+1}). The procedure is shown in Figure 2.(1).

Assume that there are *m* files to which *r* has permissions. For each of these files f_n, the administrator delegates the cloud provider to update the *FK* tuple of *r* for f_n. To do so, the administrator generates a re-key $rek_r = dk_{r,v_r+1}{-}dk_{r,v_r}$ of the proxy re-encryption [16] between the new role decryption key dk_{r,v_r+1} and the old role decryption key dk_{r,v_r}. With this key, the cloud can update the *FK* tuple through proxy re-encryption as:

$$\langle FK,\ (r,\ v_r+1),\ (f_n,\ v_{f_n},\ op),\ c,\ sign_{SU}\rangle$$

$$c = Enc^{Pub}_{ek_{r,v_r+1}}(k^0)||Enc^{Pub}_{ek_{r,v_r+1}}(k^1)||...||Enc^{Pub}_{ek_{r,v_r+1}}(k^t)$$

$$\text{for } 0 \le i \le t\text{: } Enc^{Pub}_{ek_{r,v_r+1}}(k^i) = Enc^{Pub}_{ek_{r,v_r}}(k^i) \oplus rek_r$$

Notice that with symmetric onion encryption strategy, an *FK* tuple encrypts a symmetric key list $\{k^0, k^1, ..., k^t\}$ instead of a single file key *k*. After the updating, the new *FK* tuple encrypts the symmetric key list by the new encryption key ek_{r,v_r+1} of *r*. The procedure is shown in Figure 2.(2).

Delegating the cloud provider to update the contents of *RK* and *FK* tuples invalidate the signatures in these tuples. This is one of the reasons why the construction in [14] requires the administrator to update *RK* and *FK* tuples by itself. To solve this problem, our solution is to rely on sanitizable signature scheme [20], in which a signer can delegate a proxy to modify a certain portion of a signed message, while disallowing the proxy to generate fake signed messages. With this scheme, the administrator signs each *RK*/*FK* tuple keeping the version number part and the ciphertext part sanitizable for the cloud provider.

Symmetric onion encryption: To complete the revocation, the administrator still needs to re-key and re-encrypt all the *F* tuples to which *r* have permissions. With symmetric onion encryption, the administrator just needs to request the cloud to encrypt the file data with a new layer of symmetric encryption and update the encrypted symmetric key list accordingly. Symmetric onion encryption encrypts file data in a onion manner; and Crypt-DAC uses the innermost encryption layer to protect the file data against the cloud provider and the outermost encryption layer to protect the file data against the revoked users.

With symmetric onion encryption, a *F* tuple encrypts a file f_n by a symmetric key list $\{k^0, k^1,..., k^t\}$ in a onion manner:

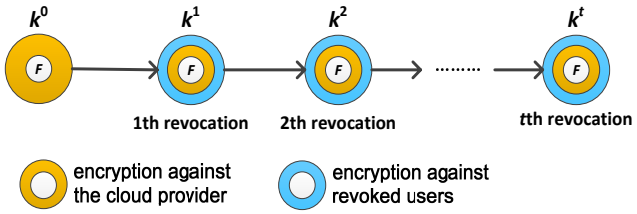


Fig. 4: Evolution of encryptions of a F tuple.

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

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

Accordingly, all the FK tuples with permissions to f_n encrypt the symmetric key list $\{k^0, k^1,..., k^t\}$:

$$\langle FK,\ (r,\ v_r),\ (f_n,\ v_{f_n},\ op),\ c,\ sign_{SU}\rangle$$

$$c = Enc^{Pub}_{ek_{r,v_r}}(k^0)||Enc^{Pub}ek_{r,v_r}(k^1)||...||Enc^{Pub}ek_{r,v_r}(k^t)$$

When a user u accesses f_n, u downloads one of these FK tuples, decrypts the symmetric key list, and uses the list to decrypt the F tuple in a de-onion manner. When the F tuple is created for the first time, the symmetric key list contains only one key $\{k^0\}$. Each time the F tuple is re-keyed and re-encrypted in a revocation operation (either user or role), a new key is inserted into the symmetric key list.

Returning to the revocation of u' from r, for each of the files f_n to which r has permissions, suppose that the symmetric key list of the F tuple is $\{k^0, k^1,..., k^t\}$, the administrator generates a new key $k^{t+1} \leftarrow KeyGen^{Sym}(1^n)$ and sends k^{t+1} to the cloud provider. Suppose there are m roles with permissions to f_n. The cloud provider updates the encrypted symmetric key lists in the FK tuples of the m roles:

$$\text{For role } r\text{: } \langle FK,\ (r,\ v_r+1),\ (f_n,\ op),\ v_{f_n}\text{+1},\ c||c',\ sign_{SU}\rangle$$

$$c' = Enc^{Pub}_{ek_{r,v_r+1}}(k^{t+1})$$

$$\text{For other roles } r'\text{: } \langle FK,\ (r',\ v_{r'}),\ (f_n,\ op),\ v_{f_n}\text{+1},\ c||c',\ sign_{SU}\rangle$$

$$c' = Enc^{Pub}_{ek_{r',v_{r'}}}(k^{t+1})$$

The procedure is shown in Figure 3.(1). The administrator also requests the cloud to update the F tuple as:

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

The procedure is shown in Figure 3.(2).

For a F tuple encrypted by a symmetric key list $\{k^0, k^1,..., k^t\}$, Crypt-DAC uses the innermost encryption layer to protect the file data against the cloud provider, as k^0 is not revealed to the cloud provider during the life cycle of the F tuple. Also, Crypt-DAC uses the outermost encryption layer to protect the file data against revoked users. After ith user revocation, the file data is encrypted by a new key k^{i+1}, which cannot be accessed by the revoked user. The evolution of the encryptions of a F tuple is shown in Figure 4.

We notice that in symmetric onion encryption, the cloud provider just encrypts the ciphertext part of F tuple with a new layer of symmetric encryption, and not changes the encrypted file content. Therefore, this operation does not invalidate the signature of the F tuple, and we do not need to use sanitizable signature scheme to sign F tuples.

Lazy de-onion encryption: With the symmetric onion encryption, a problem is that as re-keying and re-encrypting F tuples continuously happen, the size of the corresponding symmetric key lists incrementally increase, incurring additional overhead in file reading. For a F tuple with re-keying and re-encryption happening t+1 times, the symmetric key list becomes $\{k^0, k^1,..., k^{t+1}\}$, and a user accessing the F tuple has to download an FK tuple with the encrypted symmetric key list, and symmetrically decrypt the F tuple t+2 times to recover the file.

Crypt-DAC adopts a lazy de-onion encryption strategy to remove this overhead without incurring obvious overhead. In specific, the next user writing to a F tuple creates a new F tuple to encrypt the writing content

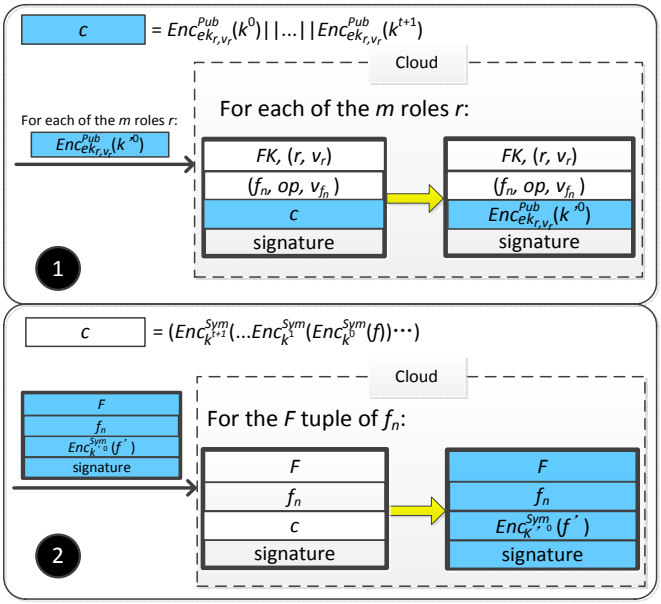


Fig. 5: Lazy de-onion encryption overview.

by a new symmetric key list containing only one key, and requests the cloud to replace the existing symmetric key lists in all the *FK* tuples with permissions to the *F* tuple with the new symmetric key list.

Suppose a user u wants to write to a file f_n. To do so, u generates a new symmetric key $k'^0 \leftarrow Gen^{Sym}(1^n)$. Assume that there are m roles having permissions to f_n. For each of the m roles r, u encrypts k'^0 by the encryption key ek_{r,v_r} of r, and uploads the encryption for the cloud provider to update the *FK* tuple of r as:

$$\langle FK, (r, v_r), (f_n, v_{f_n}, op), Enc^{Pub}_{ek_{r,v_r}}(k'^0), sign_{SU} \rangle$$

The procedure is shown in Figure 5.(1). u then creates a F tuple encrypting the writing content f' by a new symmetric key list $\{k'^0\}$:

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

and requests the cloud provider to replace the existing F tuple with this F tuple. The procedure is shown in Figure 5.(2). After the writing operation, users accessing the F tuple do not suffer the additional overhead incurred by the symmetric onion encryption.

Lazy de-onion encryption strategy requires the next user writing to a F tuple to additionally update a number of *FK* tuples on the access path to the F tuple. Generally, this number is small as in realistic data organizations, a large number of users often access files through a small number of roles. This observation is validated in the simulation of realistic workloads over real data sets [14], in which users access files through less than 8 *FK* tuples (roles) in average.

V. Design Details

In this section, we describe the various operations in Crypt-DAC. Our system partly integrates the design of *RK*, *FK* and *F* tuples used in [14]. There are three main classes of operations we consider: permission revocation, permission assignment, and file operation. The first class includes revoking the permission of a role and revoking the permission of a user. Both of the operations are conducted by the administrator and integrates our delegation-aware encryption and symmetric onion encryption strategies. The second class includes adding a new role/assigning a role to a file with *Read*/*RW* permission; and adding a new user/assigning a user to a role. All the operations are conducted by the administrator. The third class includes file creation, file reading and file writing. A user uploads his file through the file creation operation and accesses a file through file reading/writing operation. The design of file writing operation integrates our lazy de-onion encryption strategy to remove the file access overhead incurred by the symmetric onion encryption strategy.

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. The subscript after an operation name identifies who performs the operation if it is not performed by an administrator.

File management: We use three special 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, vk_u) contains a user identity *u*, the encryption key ek_u of *u*, and the verification key vk_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*, v_r, ek_{v_r}, vk_{v_r}) contains a role identity *r*, the current version of the encryption and signing key pairs of *r*, the encryption key ek_{v_r} of *r*, and the verification key vk_{v_r} of *r*.
- *FILES:* Also, 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, v_{f_n}) contains the file name f_n of a file and the current version v_{f_n} of the symmetric key list of f_n.

Key management: In our system, the administrator, cloud provider, roles and users have their own keys to process *RK*, *FK*, and *F* tuples. We introduce them as follows.

- *Administrator:* The administrator plays a role of super user and has an encryption key pair enk_{SU} of a general public key encryption scheme and a signing key pair sik_{SU} of a general signature scheme. enk_{SU} and sik_{SU} are used by the administrator to create a special *RK* tuple when adding a new role into the system. This *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. Also, ek_{SU} is 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. Besides, the administrator plays the role of signer in the sanitizable signature scheme and has a signing key pair (sk_{SU}^{sign}, vk_{SU}^{sign}) with vk_{SU}^{sign} publicly viewable. This key pair is used by the administrator to sign general *RK* and *FK* tuples.
- *Cloud provider:* The cloud provider plays the role of proxy in the sanitizable signature scheme and has a sanitization key pair ($sk_{C.P.}^{san}$, $vk_{C.P.}^{san}$) with $vk_{C.P.}^{san}$ publicly viewable. This key pair is used by the cloud provider to update general *RK* and *FK* tuples without invalidating their signatures.
- *Role:* The administrator generates an encryption key pair enk_{r,v_r} of proxy re-encryption scheme and a signing key pair sik_{r,v_r} of a general signature scheme for *r*. enk_{r,v_r} is used by the administrator to encrypt *FK* tuples, and sik_{r,v_r} is used by users writing to *F* tuples to sign their new *F* tuples.
- *User:* A user *u* has an encryption key pair enk_u of a general public key encryption scheme and a signing key pair sik_u of a general signature scheme. enk_u is used by the administrator to encrypt *RK* tuples for *u*, and sik_u is used by *u* to sign its *F* tuples when adding files into the system.

A. Permission revocation

The class of permission revocation includes revoking the permission of a role and revoking the permission of a user, as we described below.

revokeUser(*u*)(Algorithm 1): To revoke the permission of a user *u*, the administrator revokes the membership of *u* from all of its assigned roles. To do so, the administrator invokes REVOKEU(*u*, *r*) for each of the assigned roles *r*. In REVOKEU(*u*, *r*), the administrator first invokes DAE-RK(*r*) and DAE-FK(*r*), which jointly implements our delegation-aware encryption strategy. After the execution of DAE-RK(*r*) and DAE-FK(*r*), all the involved *RK* and *FK* tuples are updated. Next, for each of the assigned roles *r* and for each of the files f_n to which *r* has permission, the administrator invokes ONION-ENCRYPTION(f_n), which implements our symmetric onion encryption strategy. After the execution of ONION-ENCRYPTION(f_n), all the involved *F* tuples are re-encrypted.

revokeUser(*u*) revokes the membership of *u* from all of its assigned roles. After the revocation, *u* cannot access any file in the system. Sometimes, however, the administrator may only need to revoke the membership of *u* from a certain role *r*. To do so, the administrator directly invokes REVOKEU(*u*, *r*).

revokeRole(*r*)(Algorithm 2): To revoke the permission of a role *r*, the administrator revokes the permission of *r* from all of its assigned files. To do so, the administrator deletes the record (*r*, v_r, ek_r, vk_r) from ROLES. For each permission $p = \langle f_n, op \rangle$ that *r* has access to, the administrator invokes REVOKEP(*r*, $\langle f_n$, *Read*$\rangle$) to revoke *p*.

In REVOKEP(*r*, $\langle f_n$, *Read*$\rangle$), the administrator first requests the cloud provider to delete all the *RK* tuples about *r*. After that, users assigned to *r* cannot access the secret key sk_{r,v_r} from the cloud anymore. The administrator

Algorithm 1 revokeUser(u)

For each role r that u is assigned to:
  REVOKEU(u, r);

procedure REVOKEU(u, r)
DAE-RK(r);
DAE-FK(r);
Increment v_r in ROLES;
For each f_n with $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c, sign_{SU}\rangle$:
  ONION-ENCRYPTION(f_n);

procedure DAE-RK(r)
Generate new encryption pair enk_{r,v_r+1} and signing key pair sik_{r,v_r+1} for r;
For each $\langle RK, u', (r, v_r), c, sign_{SU}\rangle$ with $u' \neq u$:
  Admin:
    Send $Enc^{Pub}_{ek_{u'}}(dk_{r,v_r+1}, sk_{r,v_r+1})$ to C.P.;
  C.P.:
    Sanitize($\langle RK, u', (r, v_r), c\rangle$, sk^{sign}_{SU}, $vk^{san}_{C.P.}$, $sign_{SU}$
    $\langle RK, u', (r, v_r + 1), Enc^{Pub}_{ek_{u'}}(dk_{r,v_r+1}, sk_{r,v_r+1})\rangle$);

procedure DAE-FK(r)
For each f_n with $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c, sign_{SU}\rangle$:
  Admin:
    Generate re-key $rek_r \leftarrow dk_{r,v_r+1} - dk_{r,v_r}$;
    Send rek_r to C.P.;
  C.P.:
    Parse $c = Enc^{Pub}_{r,v_r}(k^1)||...||Enc^{Pub}_{r,v_r}(k^t)$;
    Compute $c' = Enc^{Pub}_{r,v_r}(k^1)\oplus rek_r||...\oplus rek_r||$
    $Enc^{Pub}_{r,v_r}(k^t)\oplus rek_r$;
    Sanitize($\langle FK, (r, v_r), (f_n, v_{f_n}, op), c\rangle$, sk^{sign}_{SU},
    $vk^{san}_{C.P.}$, $sign_{SU}$, $\langle FK, (r, v_r{+}1), (f_n, v_{f_n}, op), c'\rangle$);

procedure ONION-ENCRYPTION(f_n)
  Admin:
    Generate new symmetric key $k^{t+1} \leftarrow Gen^{Sym}()$;
    Send k^{t+1} to C.P.;
  C.P.: $\langle F, f_n, c, sign_{r,v_r}\rangle$:
    Compute $c \leftarrow Enc^{sym}_{k^{t+1}}(c)$;
  For each $\langle FK, (r', v_{r'}), (f_n, v_{f_n}, op), c, sign_{SU}\rangle$:
    C.P.:
    Sanitize($\langle FK, (r', v_{r'}), (f_n, v_{f_n}, op), c\rangle$, sk^{sign}_{SU},
    $vk^{san}_{C.P.}$, $sign_{SU}$, $\langle FK, (r', v_{r'}), (f_n, v_{f_n}{+}1, op),$
    $c||Enc^{Pub}_{r',v_{r'}}(k^{t+1})\rangle$);

then requests the cloud provider to delete the *FK* tuple of r for f_n. After that, users assigned to r cannot access the symmetric key list encrypting the *F* tuple of f_n from the cloud anymore. However, users assigned to r who have accessed f_n may locally cache the symmetric key list to continuously access the *F* tuple of f_n. To forbidden such illegal accesses, the administrator invokes ONION-ENCRYPTION(f_n), which implements our onion-encryption strategy, as we explained earlier.

revokeRole(r) totally revokes the permission of r from all of its assigned files. After the revocation, members of r cannot access any file in the system through r. Sometimes, however, the administrator may only need to revoke the permission of r from a certain file. We notice that revokeRole(r) can be slightly modified to revoke a single permission of r. There are two cases. To revoke a permission of r from $\langle f_n, RW\rangle$ to $\langle f_n, Read\rangle$, the administrator directly invokes REVOKEP(r, $\langle f_n, RW\rangle$) to update the *FK* tuple. After that, the cloud provider does not allow users assigned to r to write to f_n anymore. To totally revoke a permission $\langle f_n, op\rangle$ of r, the administrator directly invokes REVOKEP(r, $\langle f_n, Read\rangle$).

B. Permission assignment

The class of permission assignment includes adding a new user, assigning a user to a role, adding a new role, and assigning a role to a file with *Read*/*RW* permission, as we described below.

addUser(u)(Algorithm 3): When a user u joins the system, it generates a encryption pair enk_u and a signing key pair sik_u for itself and sends a message (u, ek_u, vk_u) to the administrator for registration. Upon receiving the message, the administrator inserts a record (u, ek_u, vk_u) for u into USERS.

addRole(r)(Algorithm 4): When the administrator needs to add a new role r, the administrator generates an encryption key pair $enk_{r,1}$ and a signing key pair $sik_{r,1}$ for r, and inserts (r, 1, $ek_{r,1}$, $vk_{r,1}$) into ROLES, where 1 means that r is in its first version. The administrator then encrypts the decryption key $dk_{r,1}$ and signing key $sk_{r,1}$

Algorithm 2 revokeRole(r)

Remove (r, v_r, ek_r, vk_r) from ROLES;
For each permission $p = \langle f_n, op \rangle$ that r has access to:
  REVOKEP(r, $\langle f_n, Read \rangle$);

procedure REVOKEP(r, $\langle f_n, RW \rangle$)
  Send $\langle FK, (r, v_r), (f_n, v_{f_n}, Read), c, sign_{SU} \rangle$ to C.P.;
  Req C.P. to delete $\langle FK, (r, v_r), (f_n, v_{f_n}, RW), c, sign_{SU} \rangle$;

procedure REVOKEP(r, $\langle f_n, Read \rangle$)
  Req C.P. to delete all $\langle RK, –, (r, v_r), –, – \rangle$;
  Req C.P. to delete $\langle FK, (r, v_r), (f_n, –, –), –, – \rangle$;
  ONION-ENCRYPTION(f_n);
  Increment v_{f_n} in FILES;

procedure ONION-ENCRYPTION(f_n)
  Admin:
    Generate new symmetric key $k^{t+1} \leftarrow Gen^{Sym}()$;
    Send k^{t+1} to C.P.;
  C.P.: $\langle F, f_n, c, sign_{r,v_r} \rangle$:
    Compute $c \leftarrow Enc^{Sym}_{k^{t+1}}(c)$;
  For each $\langle FK, (r', v_{r'}), (f_n, v_{f_n}, op), c, sign_{SU} \rangle$:
    C.P.:
      Sanitize($\langle FK, (r', v_{r'}), (f_n, v_{f_n}, op), c \rangle$, sk^{sign}_{SU},
      $vk^{san}_{C.P.}$, $sign_{SU}$, $\langle FK, (r', v_{r'}), (f_n, v_{f_n}+1, op),$
      $c||Enc^{Pub}_{r',v_{r'}}(k^{t+1}) \rangle$);

Algorithm 3 addUser(u)

user u:
    Compute $(ek_u, dk_u) \leftarrow Gen^{Pub}()$;
    Compute $(sk_u, vk_u) \leftarrow Gen^{Sign}()$;
    Send (u, ek_u, sk_u) to admin;
Admin:
    Add (u, ek_u, sk_u) to USERS;
    Return success to u;

by its encryption key ek_{SU}, and uploads an *RK* tuple containing the encryption to the cloud provider. The *RK* tuple means that the administrator is a super user who can access ($dk_{r,1}$, $sk_{r,1}$). With this *RK* tuple, the administrator can assign a user u to r by distributing ($dk_{r,1}$, $sk_{r,1}$) using another *RK* tuple(as described in assignU (u, r)).

assignU(u, r)(Algorithm 5): To assign a user u with a membership of role r, the administrator first downloads the *RK* tuple $\langle RK, SU, (r, v_r), c, sign_{SU} \rangle$, which was created by the administrator when adding r into the system. From the *RK* tuple, the administrator decrypts the decryption key dk_{r,v_r} and signing key sk_{r,v_r} of r, encrypts (dk_{r,v_r}, sk_{r,v_r}) with the public key pk_u of u, and uploads a new *RK* tuple containing the encryption to the cloud provider. This *RK* tuple enables u to access dk_{r,v_r} and sk_{r,v_r}, from which u can further access all the F tuples assigned to r.

assignP(r, $\langle f_n, op \rangle$)(Algorithm 6): There are two cases to assign a role r with a permission $\langle f_n, op \rangle$. In the first case, r has *Read* access to f_n (there exists an *FK* tuple $\langle FK, (r, v_r), (f_n, v_{f_n}, Read), c, sign_{SU} \rangle$) while *op* is *RW*. The administrator downloads the *FK* tuple, changes *Read* to *RW*, and uploads the new *FK* tuple to the cloud provider to replace the existing *FK* tuple. In the second case, r has no permission to f_n (there does not exist $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c, sign_{SU} \rangle$). The administrator first downloads the *FK* tuple $\langle FK, SU, (f_n, RW), c, sign_{SU} \rangle$, which was created by the administrator when creating f_n. From the *FK* tuple, the administrator decrypts the symmetric key list $\{k^1, ..., k^t\}$ of f_n, encrypts $\{k^1, ..., k^t\}$ with the encryption key ek_{r,v_r} of r, and uploads a new *FK* tuple containing the encryption to the cloud provider. This *FK* tuple enables r to access f_n with operation *op*.

C. File operation

The class of file operation includes file creation, file read and file write, as we described below.

addFile(f_n, f, u)(Algorithm 7): When a user u wants to upload a file f, it generates a symmetric key list with a single key $\{k\}$. u then encrypts f with k to create an F tuple and encrypts k with the encryption key ek_u of u to create an *FK* tuple. Both the tuples are signed by the signing key sk_u of u. The *FK* tuple means that the administrator is a super user who can access $\{k\}$. With this *FK* tuple, the administrator can assign a permission $\langle f_n, op \rangle$ to a role r by distributing $\{k\}$ using another *FK* tuple(as described in assignP(r, $\langle f_n, op \rangle$)). Finally, u uploads both the F tuple and *FK* tuple to the cloud provider and requests the cloud provider to insert (f_n, 1) into FILES.

Algorithm 4 addRole(*r*)

1: Compute $(ek_{r,1}, dk_{r,1}) \leftarrow Gen^{Pub}()$;
2: Compute $(sk_{r,1}, vk_{r,1}) \leftarrow Gen^{Sign}()$;
3: Add (*r*, 1, $ek_{r,1}$, $vk_{r,1}$) to ROLES;
4: Compute $c \leftarrow Enc^{Pub}_{ek_{SU}}(dk_{r,1}, sk_{r,1})$;
5: Compute $sign_{SU} \leftarrow Sign_{sk_{SU}}$(*RK*, *SU*, (*r*, 1), *c*);
6: Send $\langle$*RK*, *SU*, (*r*, 1), *c*, $sign_{SU}\rangle$ to C.P.;

Algorithm 5 assignU(*u*, *r*)

1: Download $\langle$*RK*, *SU*, (*r*, v_r), *c*, $sign_{SU}\rangle$ from C.P.;
2: Compute $(dk_{r,v_r}, sk_{r,v_r}) \leftarrow Dec^{Pub}_{dk_{SU}}(c)$;
3: Compute $c' \leftarrow Enc^{Pub}_{ek_u}(dk_{r,v_r}, sk_{r,v_r})$;
4: Compute $sign_{SU} \leftarrow Sign_{sk_{SU}}$(*RK*, *u*, (*r*, v_r), *c'*);
5: Send $\langle$*RK*, *u*, (*r*, v_r), *c'*, $sign_{SU}\rangle$ to C.P.;

read(f_n, *u*)(Algorithm 8): When a user *u* wants to read a file f_n, *u* sends a message (*u*, *r*, f_n) to the cloud provider. Upon receiving the message, the cloud provider searches: (1) an *RK* tuple that contains *u* and *r*; (2) a *FK* tuple that contains *r* and $\langle f_n$, *Read*$\rangle$; and (3) a *F* tuple that contains f_n. If all of the three tuples can be searched, the administrator directly returns the three tuples to *u*. Upon receiving them, *u* first verifies if the three tuples contain valid signatures. If yes, *u* then decrypts the decryption key dk_{r,v_r} of *r* from the RK tuple by its decryption key dk_u. Next, *u* decrypts the symmetric key list from the *FK* tuple by dk_{r,v_r}. Finally, *u* decrypts the file f from the F tuple by the symmetric key list.

write(f_n, *u*)(Algorithm 9): When a user *u* wants to write to a file f_n, *u* sends a message (f_n, write request) to the cloud provider. Upon receiving the message, the cloud provider searches: (1) an *RK* tuple that assigns *u* with a role *r*; (2) a *FK* tuple that assigns *r* with a permission (f_n, *RW*). If both the tuples can be searched, the cloud provider searches all the roles with permission to f_n (through their *FK* tuples), and includes their records in ROLES into a set *role-set*. After that, the cloud provider returns *role-set* as well as the *RK* tuple $\langle$*RK*, *u*, (*r*, v_r), *c*, $sign_{SU}\rangle$ to *u* for whom to execute lazy de-onion encryption strategy on them. Upon receiving *role-set* and $\langle$*RK*, *u*, (*r*, v_r), *c*, $sign_{SU}\rangle$, *u* invokes LAZY DE-ONION(*role-set*, $\langle$*RK*, *u*, (*r*, v_r), *c*, $sign_{SU}\rangle$), which implements our lazy de-onion encryption strategy as discussed earlier.

We noticed that in some cases, *u* does not need to execute lazy de-onion encryption strategy as no revocation happens subject to f_n (i.e., no role with permission to f_n is revoked and no user assigned to a role with permission to f_n is revoked). In this case, the cloud provider returns the searched *RK* tuple and *FK* tuple. Upon receiving them, *u* first decrypts the decryption key dk_{r,v_r} and signing key sk_{r,v_r} of *r* from the RK tuple by its decryption key dk_u. *u* next decrypts the symmetric key list $\{k\}$ from the *FK* tuple by dk_{r,v_r}. After that, *u* encrypts its writing content by $\{k\}$ to create a new *F* tuple, and signs the *F* tuple by sk_{r,v_r}. Finally, *u* uploads the *F* tuple and requests the cloud provider to use it to replace the existing *F* tuple.

VI. Performance analysis

We compare the performance of Crypt-DAC with IMre and DEre through theoretical analysis. We only concern four operations: revocation of a user from a role, revocation of a role from a file, file reading, and file writing, as Crypt-DAC performs well as IMre and DEre in the remainder operations. We summarize the analysis results in Tables 1, 2, and 3 in Appendix A. We use the following notations:

- (Gen^{Sym}, Gen^{Pub}, Gen^{Sign}): computation overhead of generating encryption key pair, signing key pair and symmetric key
- (Enc^{Pub}, Dec^{Pub}): computation overhead of encrypt/decrypt a message by a public encryption scheme
- $|Enc^{Pub}|$: communication overhead of a public encryption
- (*Sign*, *Verify*): computation overhead of signing/verification by a signature scheme
- |*Sign*|: communication overhead of a signature
- (Enc^{Sym}, Dec^{Sym}): computation overhead of encrypt/decrypt a message by a symmetric encryption scheme
- $|k|$: communication overhead of a symmetric key
- $|f|$: communication overhead of a file
- *mems*(*r*): number of users who are members of role *r*
- *files*(*r*): number of files to which *r* has permissions
- |*files*(*r*)|: communication overhead of files to which *r* has permissions

Algorithm 6 assignP(r, $\langle f_n, op \rangle$)

If there exists $\langle FK, (r, v_r), (f_n, v_{f_n}, Read), c, sign_{SU} \rangle \wedge op = RW$:
  Download $\langle FK, (r, v_r), (f_n, v_{f_n}, Read), c,$
  $sign_{SU} \rangle$ from C.P.;
  Compute $sign_{SU} \leftarrow Sign_{sk_{SU}^{sign}, vk_{C.P.}^{san}}(FK, (r, v_r),$
  $(f_n, v_{f_n}, RW), c)$;
  Send $\langle FK, (r, v_r), (f_n, v_{f_n}, RW), c, sign_{SU} \rangle$
  to C.P.;
  Req C.P. to delete $\langle FK, (r, v_r), (f_n, v_{f_n}, Read),$
  $c, sign_{SU} \rangle$;
If there does not exist $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c, sign_{SU} \rangle$:
  Download $\langle FK, SU, (f_n, RW), c, sign_{SU} \rangle$ from C.P.;
  Parse $c = c^1||...||c^t$;
  Compute $\{k^1 \leftarrow Dec_{dk_{SU}}^{Pub}(c^1), ..., k^t \leftarrow Dec_{dk_{SU}}^{Pub}(c^t)\}$;
  Compute $c' \leftarrow Enc_{ek_{r,v_r}}^{Pub}(k^1)||...||Enc_{ek_{r,v_r}}^{Pub}(k^t)$;
  Compute $sign_{SU} \leftarrow Sign_{sk_{SU}^{sign}, vk_{C.P.}^{san}}(FK, (r, v_r),$
  $(f_n, v_{f_n}, op), c')$;
  Send $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c', sign_{SU} \rangle$ to C.P.;

Algorithm 7 addFile(f_n, f, u)

Compute $k \leftarrow Gen^{Sym}()$;
Compute $c \leftarrow Enc_k^{Sym}(f)$;
Compute $c' \leftarrow Enc_{ek_u}^{Pub}(k)$;
Compute $sign_u \leftarrow Sign_{sk_u}(F, f_n, c)$;
Compute $sign_u \leftarrow Sign_{sk_u}(FK, SU, (f_n, Read), c')$;
Send $\langle F, f_n, c, sign_u \rangle$ and $\langle FK, SU, (f_n, Read), c', sign_u \rangle$ to C.P.;
Req C.P. to insert (f_n, 1) into FILES.

- *roles*(f_n): number of roles which are assigned permissions to file f_n
- *versions*(f_n): number of revocations involving f_n

A. Revocation of a user from a role

Removing a user *u* from a role *r* entails two tasks: (1) creating new *RK* tuples and *FK* tuples; and (2) re-keying and re-encrypting *F* tuples. We evaluate the performance of the two tasks separately.

Creating new *RK* tuples and *FK* tuples: The computation and communication overhead of creating new *RK* tuples and *FK* tuples is shown in Table 1.

Comparing with IMre, Crypt-DAC saves overhead as:

Computation:

$$\begin{aligned} &mems(r)\times Sign + files(r)\times(Enc^{Pub} + Sign) \\ &+(\textstyle\sum_{i=1}^{files(r)} roles(f_{n_i}))\times(Enc^{Pub} + Sign) \end{aligned}$$

Communication:

$$\begin{aligned} &mems(r)\times|Sign| + files(r)\times(|Enc^{Pub}| + |Sign|) \\ &+(\textstyle\sum_{i=1}^{files(r)} roles(f_{n_i}))\times(|Enc^{Pub}| + |Sign|) \end{aligned}$$

Comparing with DEre, Crypt-DAC saves overhead as:

Computation:

$$\begin{aligned} &mems(r)\times Sign + files(r)\times(Dec^{Pub} + Enc^{Pub} + Sign) \\ &+(\textstyle\sum_{i=1}^{files(r)} roles(f_{n_i}))\times(Enc^{Pub} + Sign) \end{aligned}$$

Communication:

$$\begin{aligned} &mems(r)\times|Sign| + 2files(r)\times(|Enc^{Pub}| + |Sign|) \\ &+(\textstyle\sum_{i=1}^{files(r)} roles(f_{n_i}))\times(|Enc^{Pub}| + |Sign|) \end{aligned}$$

Comparing with the previous work, Crypt-DAC saves computation and communication overhead to process public encryptions and signatures. This efficiency improvement stems from delegation-aware encryption strategy.

Algorithm 8 read(f_n, u)

User u:
  Send (u, r, f_n) to C.P.;

C.P.:
  If there exists an *RK* tuple $\langle RK, u, (r, v_r), c, sign_{SU}\rangle \wedge$
    a *FK* tuple $\langle FK, (r, v_r), (f_n, v_{f_n}, op), c', sign_{SU}\rangle \wedge$
    a *F* tuple $\langle F, f_n, c'', sign\rangle$:
  Then
    Return the *RK* tuple, *FK* tuple, and *F* tuple to u;
  Else
    Return $\perp$ to u;

User u:
Compute $Verify_{vk_{SU}^{sign}, vk_{C.P}^{san}}((RK, u, (r, v_r), c), sign_{SU})$;
Compute $Verify_{vk_{SU}^{sign}, vk_{C.P}^{san}}((FK, (r, v_r), (f_n, v_{f_n}, op), c'), sign_{SU}^{sign})$;
Compute $Verify_{vk}((F, f_n, c''), sign)$;
  If all the signatures are valid:
    Compute $(dk_{r,v_r}, sk_{r,v_r}) \leftarrow Dec_{dk_u}^{Pub}(c)$;
    Parse c' as $c_1||c_2||...c_n$;
    For $i = n$ & $i > 0$ & i++:
      Compute $k_i \leftarrow Dec_{dk_{r,v_r}}^{Pub}(c_i)$;
      Compute $c'' \leftarrow Dec_{k_i}^{Sym}(c'')$;

Algorithm 9 write(f_n, u)

User u:
  Send (f_n, write request) to C.P.;

C.P.:
  If there exists an *RK* tuple $\langle RK, u, (r, v_r), c, sign_{SU}\rangle \wedge$
    a *FK* tuple $\langle FK, (r, v_r), (f_n, v_{f_n}, RW), c', sign_{SU}\rangle$:
  Then
    Include all the *FK* tuples $\langle FK, (r', v_{r'}), (f_n, v_{f_n}, RW)$
    $, c'', sign_{SU}\rangle$ into *FK-set*;
    Include all the records $(r', v_{r'}, ek_{r',v_{r'}}, vk_{r,v_{r'}})$
    in ROLES into *role-set*;
    Return *role-set* and $\langle RK, u, (r, v_r), c, sign_{SU}\rangle$ to u;
  Else
    Return $\perp$ to u;

User u:
  LAZY DE-ONION(*role-set*, $\langle RK, u, (r, v_r), c, sign_{SU}\rangle$);

procedure LAZY DE-ONION(*role-set*, $\langle RK, u, (r, v_r), c, sign_{SU}\rangle$)
  Compute $sk_{r,v_r} \leftarrow Dec_{dk_u}^{Pub}(c)$;
  Compute $k'^1 \leftarrow Gen^{Sym}()$;
  Compute $c' \leftarrow Enc_{k'^1}^{Sym}(f')$;
  Compute $sign_{r,v_r} \leftarrow Sign_{sk_{r,v_r}}(F, f_n, c')$;
  For each record $(r', v_{r'}, ek_{r',v_{r'}}, vk_{r',v_{r'}}) \in$ *role-set*:
    Compute $c'' \leftarrow Enc_{pk_{r,v_r}}^{Pub}(k'^1)$;
    Inserts c'' into *C-set*;
  Send *C-set* and $\langle F, f_n, c', sign_{r,v_r}\rangle$ to C.P.;
  C.P.:
    If *Accept* $\leftarrow Verify_{vk_{r,v_r}}(F, f_n, c', sign_{r,v_r})$:
      For each $\langle FK, (r', v_{r'}), (f_n, v_{f_n}, RW), c''', sign_{SU}\rangle$
        $\in$ *FK-set* and a $c'' \in$ *C-set*:
        $Sanitize((FK, (r', v_{r'}), (f_n, v_{f_n}, RW), c'''), sign_{SU},$
        $vk_{SU}^{sign}, sk_{C.P.}^{san}, c'')$;
      Use the updated *FK-set* to replace the existing
      FK tuples;
      Use $\langle F, f_n, c', sign_{r,v_r}\rangle$ to replace the existing
      F tuple;

Re-keying and re-encrypting *F* tuples: The computation and communication overhead of re-keying and re-encrypting F tuples is shown in Table 1.

Comparing with DEre, Crypt-DAC saves $files(r)\times(Dec^{Sym}+Enc^{Sym}+Sign)$ computation overhead and $2\times(|files(r)|+files(r)\times|Sign|)$ $files(r)\times|k|$ communication overhead.

Comparing with IMre, Crypt-DAC costs additional $files(r)\times Gen^{Sym}$ computation overhead and $files(r)\times|k|$ communication overhead.

Comparing with the previous work, Crypt-DAC supports immediate revocation the same as IMre with lightweight

computation and communication overhead to process symmetric keys. This overhead is incurred by symmetric onion encryption strategy.

B. Revocation of a role from a file

Revoking a permission of a role entails two tasks: (1) creating new *FK* tuples; and (2) re-keying and re-encrypting *F* tuples. We evaluate the performance of these two tasks separately.

Creating new *FK* tuples: The computation and communication overhead of creating *FK* tuples is summarized in Table 2.

Comparing with IMre and DEre, Crypt-DAC saves *files*(*r*)×(Enc^{Pub}+*Sign*) computation overhead and *files*(*r*)×(|Enc^{Pub}|+|*Sign*|) communication overhead.

Comparing with the previous work, Crypt-DAC saves computation and communication overhead to process public encryptions and signatures. This efficiency improvement stems from delegation-aware encryption strategy.

Re-keying and re-encrypting *F* tuples: The computation and communication overhead of re-keying and re-encrypting *F* tuples is summarized in Table 2.

Comparing with IMre, Crypt-DAC saves Gen^{Sym}+Dec^{Sym}+Enc^{Sym}+*Sign* computation overhead and 2×(|*file*|+|*Sign*|) communication overhead.

Comparing with DEre, Crypt-DAC costs additional *files*(*r*)×Gen^{Sym} computation overhead and *files*(*r*)×|*k*| communication overhead.

Comparing with the previous work, Crypt-DAC supports immediate revocation the same as IMre with lightweight computation and communication overhead to process symmetric keys. This overhead is incurred by symmetric onion encryption strategy.

C. File reading/writing

File reading: The computation and communication overhead of reading a file f_n is summarized in Table 3. In comparison, Crypt-DAC performs well as the previous work when no revocation involving f_n happens. On the other hand, Crypt-DAC costs additional *version*(f_n)×Dec^{Pub}+*version*(f_n)×Dec^{Sym} computation overhead when revocations involving f_n happen. Also, Crypt-DAC performs well as the previous work when no revocation involving f_n happens. On the other hand, Crypt-DAC costs additional *version*(f_n)×|Enc^{Pub}| communication overhead when revocations involving f_n happen.

Comparing with the previous work, Crypt-DAC performs well as the previous work in the non-revocation case, and costs additional computation and communication overhead to process public encryptions and signatures in the revocation case. This overhead is incurred by symmetric onion encryption strategy. We also notice that the performance of Crypt-DAC in the revocation case can be reduced to that in the non-revocation case once the file is written by a user through lazy de-onion encryption strategy.

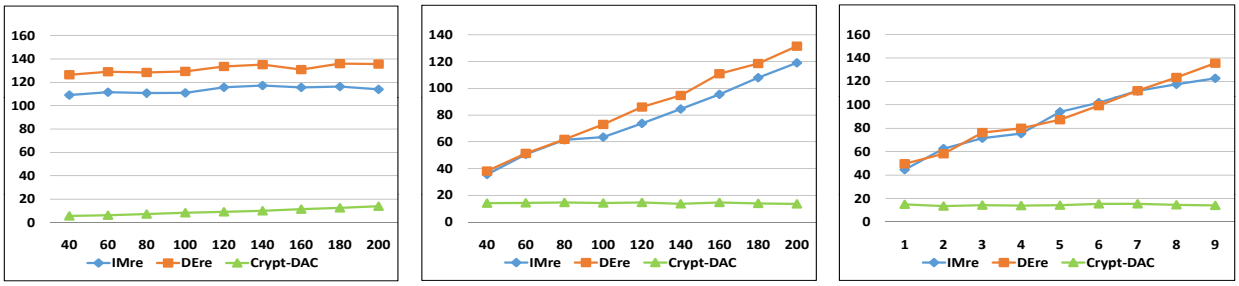
File writing: The computation and communication overhead of writing to a file f_n is summarized in Table 3. In comparison, Crypt-DAC performs well as the previous work when no revocation involving f_n happens. On the other hand, Crypt-DAC costs additional *roles*(f_n)×(Enc^{Pub}+*Verify*) computation overhead when revocations involving f_n happen. Also, Crypt-DAC performs well as the previous work when no revocation involving f_n happens. On the other hand, Crypt-DAC costs additional *roles*(f_n)×(|Enc^{Pub}|+|*Sign*|) communication overhead when revocations involving f_n happen.

Comparing with the previous work, Crypt-DAC performs well as the previous work in the non-revocation case, and costs additional computation and communication overhead to process public encryptions and signatures in the revocation case. This overhead is incurred by lazy de-onion encryption strategy.

VII. Security analysis

We analyze the security of Crypt-DAC using the access control expressiveness framework known as parameterized expressiveness [43]. In particular, we consider three fundamental security properties: correctness, AC-preservation, and safety. We show that IMre and Crypt-DAC achieve all of the three security properties and DEre cannot achieve safety.

At a high level, correctness ensures that an execution environment cannot determine whether it is interacting with the original *RBAC*$_0$ system or with a candidate access control system through basic inputs and outputs. AC-preservation ensures that the authorization requests of the *RBAC*$_0$ system is asked directly in the candidate access control system, rather than being translated to any other queries. Finally, safety ensures that the candidate access control system does not grant or revoke unnecessary permissions during the implementation of a single *RBAC*$_0$



(a) n=200 and k=9, change m from 40 to 200 (b) m=200 and k=9, change n from 40 to 200 (c) m=200 and n=200, change k from 1 to 9

Fig. 6: Time (seconds) of creating new *RK* and *FK* tuples.

command. The formal definitions of the three security properties can be found in [43]. As the security analysis of IMre is similar with that of Crypt-DAC, we only focus on Crypt-DAC and DEre here. We get our result in theorem 1.

Theorem 1: Crypt-DAC system implements $RBAC_0$ with correctness, AC-preservation, and safety achieved.

The full proof of Theorem 1 can be found in Appendix A. We first formalize Crypt-DAC under the parameterized expressiveness framework. We then provide a formal mapping from Crypt-DAC to $RBAC_0$. We show that this mapping achieves correctness, AC-preservation, and safety.

As Crypt-DAC inherits the design of [14], our proof is also similar with the proof of [14]. 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 $\langle F, f_n, c, sign_{r,v_r}\rangle$. 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, DEre 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 $RBAC_0$, DEre sequentially executes **revokeUser** and **write**. The 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.

VIII. System Evaluation

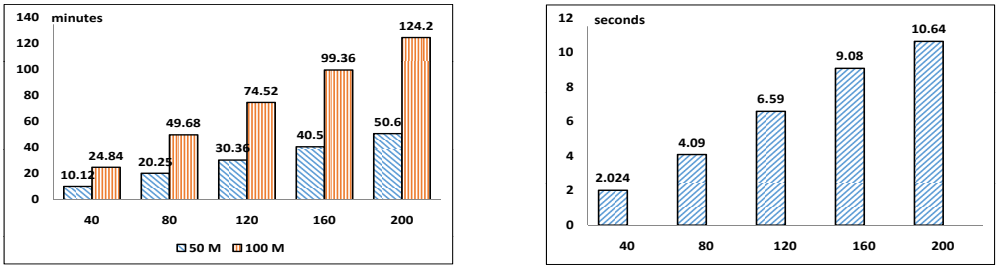
We implement IMre, DEre, and Crypt-DAC on AliCloud. Our goal is to compare the performance of the three systems at the administrator side and the user side. We build the cryptographic schemes used in the three systems based on two libraries: Crypto++ [44] and Charm [45]. We select the AES scheme with 128 bit keys and instantiate the El Gamal encryption scheme, the DSA signature scheme, and the chameleon hash function with a security parameter of 1024 bits.

Garrison et al. [14] use a simulation framework [46] to simulate a realistic workload over 6 real datasets, and use the workload to evaluate the performance of DEre. To reflect the performance of IMre, DEre, and Crypt-DAC in realistic file access scenarios, we extract three invariants from the simulation results in [14] to guide parameter selection in our experiments:

- To revoke a user from a role, the total number of involved *RK* and *FK* tuples that need to be updated falls in [0, 2000]
- To revoke a user from a role, the total number of involved F tuples that need to be re-keyed and re-encrypted falls in [0, 200]
- The number of *FK* tuples assigned to a F tuple falls in [0, 8]

Micro benchmarks: We evaluate five basic operations **read**(f_n, u), **write**(f_n, u), **revokeUser**(u, r), **addFile**(f_n, f, u), **revokeUser**(r, f_n) in Crypt-DAC to give an intuitive understanding of its performance. In specific, we evaluate (1) reading 100 KB file data with no revocation; (2) writing 100 KB file data with no revocation; (3) uploading 100 KB file data; (4) revoking a user u from a role r with *users*(r)=5, *files*(r)=2, and *roles*(f_n) = 5 for each of the two files; and (5) revoking a role r from a file f_n with *roles*(f_n) = 5.

We summarize the results in Table 1. From the table, we can see that the most expensive operation is to revoke a user from a role, which costs 2.2 seconds. All the other operations can be completed within 30 mili seconds. The



(a) Performance of IMre

(b) Performance of Crypt-DAC

Fig. 7: Performance in file data updating

reason is that revoke a user from a role requires generation of new encryption and signing key pairs for the role, which is expensive. We further find that the time to generate encryption and signing key pairs is not fixed, and can be varied from 0.25 seconds to 8.9 seconds in our experiments. We generate encryption and signing key pairs 1000 times and compute the average time as 2.5 seconds. We will use this time in our subsequent experiments.

Revocation: We evaluate the performance of IMre, DEre, and Crypt-DAC in revoking a user u from a role r. We first evaluate the time of the three systems in creating new *RK* and *FK* tuples at the administrator side. This experiment is affected by three parameters: (1) m users remaining in r; (2) n *F* tuples to which r has permissions; and (3) each *F* tuple has k roles (*FK* tuples) having permissions. To measure the effect of the three parameters, we conduct three sub groups of experiments. In the first group, we fix n=200 and k=9, and changes m from 40 to 200. In the second group, we fix m=200 and k=9, and changes n from 40 to 200. In the third group, we fix m=200 and n=200 and changes k from 1 to 9. In all the three groups of experiments, we preserve the invariant that the total number of involved *RK* and *FK* tuples that need to be updated falls in [0, 2000]. We show the experiment results in Figure 6.

From Figure 6, we observe that as the number of *RK* and *FK* tuples increases, the time cost of IMre and DEre is 6 times higher than Crypt-DAC. This validates the advantage of our delegation-aware encryption strategy to reduce the overhead of policy data updating at the administrator side. We further observe that the time cost of IMre and DEre increases fast as n and k increases. The reason is that both IMre and DEre need to update $n \times k$ *FK* tuples for the $n \times k$ roles with permissions to the n files. When the parameter n (k) increases by a constant α, the administrator needs to update $\alpha \times k$ ($n \times \alpha$) additional *FK* tuples. On the other hand, the time cost of Crypt-DAC is not affected by n and k as the administrator does not need to update the $n \times k$ *FK* tuples due to the design of delegation-aware encryption.

We next evaluate the time of IMre and Crypt-DAC in re-keying and re-encryption of *F* tuples at the administrator side. 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 two sizes of a file: 50 M and 100 M. To preserve the invariant that the number of *F* tuples that need to be re-encrypted falls in [0, 200], we change this number from 40 to 200. We show the experiment results in Figure 7.

From Figure 7, we observe that the time cost of IMre is prohibitive and increases fast as the file size and the

TABLE I: Performance of basic operations in Crypt-DAC

Operation	**Time**
Read 100KB file data	20.6 mSec
Write 100KB file data	14 mSec
Upload 100KB file data	12.4 mSec
Revoke a user from a role	2.2 Sec
Revoke a role from a file	2.9 mSec
Generate encryption and signing key pairs	2.5 Sec

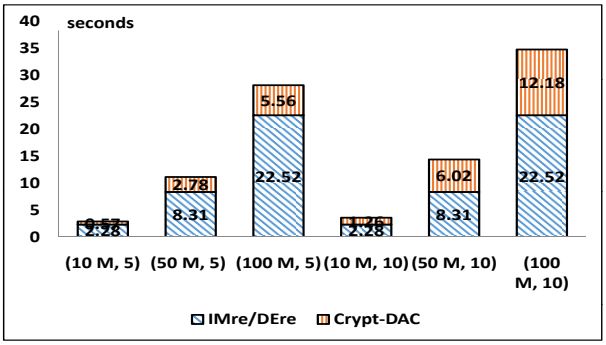


Fig. 8: Performance in file reading

number of F tuples increases. When processing 200 F tuples with 100 M file size, IMre costs more than 2 hours. On the other hand, Crypt-DAC costs about 10 seconds under the same parameters. This validates the advantage of our symmetric onion encryption strategy to reduce the overhead of file data updating at the administrator side. Also, we observe that the time cost of Crypt-DAC is not affected by the file size. The reason is that the administrator only needs to generate and send symmetric keys for F tuples regardless of their file size.

File reading/writing: We evaluate the performance of IMre, DEre, and Crypt-DAC in file reading/writing. We first consider reading case. This experiment is affected by two parameters: file size and number of revocations involving a file. We vary the file size from 10 M, 50 M to 100 M and the number of revocations from 5 to 10. We show the experiment results in Figure 8. From the Figure, we can see that Crypt-DAC costs more time comparing with IMre and DEre. This validates that symmetric onion encryption incurs additional file reading overhead. When the number of revocations is small, the additional time cost of Crypt-DAC is acceptable, i.e., just several seconds. Considering the performance advantage of symmetric onion encryption in revocation, we believe that this additional cost is valuable. We also observe that the additional cost increases as the number of revocations increases. To prevent the cost accumulates over time, we rely on lazy de-onion encryption to periodically remove the overhead. Besides, we can also rely on the administrator to use part of its resource saved in revocation to help to remove the overhead.

We next consider writing case. This experiment is affected by one parameter: number of FK tuples assigned to a F tuple. To preserve the invariant that the number of FK tuples assigned to a F tuple falls in [0, 8], we set this parameter to 4. We show the experiment results in Table 2. From the table, we can see that Crypt-DAC incurs lightweight additional time cost comparing with IMre and DEre. This validates the advantage of our lazy de-onion encryption strategy to remove the reading overhead incurred by symmetric onion encryption without incurring obvious overhead. Also, we observe that the time cost of Crypt-DAC converges to IMre and DEre as the file size increases. In 100 M case, Crypt-DAC only incurs 1.7% additional time cost. The reason is that when the file size becomes large, file communication dominates the whole time cost, outweighing the additional time cost incurred by Crypt-DAC.

IX. 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: delegating the cloud to update the policy data in a privacy and verifiability-preserving manner using a delegation-aware encryption strategy, avoiding expensive re-encryptions of file data at the administrator side using a symmetric onion encryption

TABLE II: Performance in file writing

File size	IMre/DEre	Crypt-DAC	Additional cost
10 M	2.96 Sec	3.21 Sec	7.8%
50 M	8.67 Sec	8.91 Sec	2.7%
100 M	17.58 Sec	17.87 Sec	1.7%

strategy, and a lazy de-onion encryption strategy to remove the file reading overhead incurred by the symmetric onion encryption strategy. Our theoretical and system evaluations show that Crypt-DAC achieves a balance between efficiency and security in access revocation.

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] A. Boldyreva, V. Goyal, and V. Kumar, *Identity-based encryption with efficient revocation*, in ACM CCS, 2008.
[11] M. Green, and G. Ateniese, *Identity-based proxy re-encryption*, in ACNS, 2007.
[12] A. Sahai, H. Seyalioglu, and B. Waters, *Dynamic credentials and ciphertext delegation for attribute-based encryption*, in CRYPTO, 2012.
[13] X. Jin, R. Krishnan, and R. S. Sandhu, *A unified attribute-based access control model covering DAC, MAC and RBAC*, in DDBSec, 2012.
[14] 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.
[15] R. S. Sandhu, *Rationale for the RBAC96 family of access control models*, in proceedings of ACM Workshop on RBAC, 1995.
[16] A. Ivan, and Y. Dodis, *Proxy Cryptography Revisited*, in proceedings of NDSS, 2003.
[17] M. Blaze, G. Bleumer, and M. Strauss, *Divertible protocols and atomic proxy cryptography*, in proceedings of Eurocrypt, 1998.
[18] 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.
[19] T. E. Gamal, *A Public Key Cryptosystem and a Signature Scheme Based on the Discrete Logarithm*, in IEEE Transactions of Information Theory, pages 31(4): 469C472, 1985.
[20] G. Ateniese, D. H. Chou, B. Medeiros, and G. Tsudik, *Sanitizable Signatures*, in proceedings of ESORICS, 2005.
[21] AMAZON. Amazon s3 service level agreement, 2009. http: //aws.amazon.com/s3-sla/.
[22] MICROSOFT CORPORATION. Windows Azure Pricing and Service Agreement, 2009. http://www.microsoft.com/ windowsazure/pricing/.
[23] D. Boneh and M. Franklin, *Identity-based encryption from the Weil pairing*, SIAM Journal on Computing, vol. 32, no. 3, 2003.
[24] M. Green, S. Hohenberger, and B. Waters, *Outsourcing the decryption of abe ciphertexts*, in USENIX Security, 2011.
[25] B. Libert and D. Vergnaud, *Adaptive-id secure revocable identity-based encryption*, in CT-RSA, 2009.
[26] J. Katz, A. Sahai, and B. Waters, *Predicate encryption supporting disjunctions, polynomial equations, and inner products*, in EUROCRYPT, 2008.
[27] E. Shen, E. Shi, and B. Waters, *Predicate privacy in encryption systems*, in TCC, 2009.
[28] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, *Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds*, in NSDI, 2016.
[29] 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.
[30] B. H. Kim, and D. Liey, *Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices*, in IEEE S&P, 2015.
[31] D. Boneh, K. Lewi, H. Montgomery, and A. Raghu Raghunathan, K*ey homomorphic PRFs and their applications*, in CRYPTO, 2013.
[32] M. Maffei, G. Malavolta, M. Reinert, and D. Schr?der, *Privacy and access control for outsourced personal records*, in IEEE S&P, 2015.
[33] 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.
[34] E. Gudes, *The Design of a Cryptography Based Secure File System*, IEEE Transactions on Software Engineering, vol. 6, no. 5, 1980.
[35] S. G. Akl and P. D. Taylor, *Cryptographic solution to a problem of access control in a hierarchy*, TOCS, vol. 1, no. 3, 1983.
[36] 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.
[37] 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.
[38] A. L. Ferrara, G. Fuchsbauer, B. Liu, and B. Warinschi, *Policy privacy in cryptographic access control*, in CSF, 2015.
[39] L. Ibraimi, *Cryptographically enforced distributed data access control*, Ph.D. dissertation, University of Twente, 2011.
[40] D. Nali, C. M. Adams, and A. Miri, *Using mediated identity-based cryptography to support role-based access control*, in ISC 2004, 2004.
[41] A. L. Ferrara, G. Fuchsbauer, and B. Warinschi, *Cryptographically enforced RBAC*, in CSF, 2013.
[42] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters, *Secure attributebased systems*, in ACM CCS, 2006.
[43] 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.
[44] https://www.cryptopp.com/
[45] J. A. Akinyele, M. D. Green, and A. D. Rubin, *Charm: A Framework for Rapidly Prototyping Cryptosystems, in Journal of Cryptographic Engineering*, 2013.
[46] W. C. Garrison III, A. J. Lee, and T. L. Hinrichs, *An actor-based, application-aware access control evaluation framework*, in SACMAT, 2014.

APPENDIX A
SUMMARY OF COMPLEXITY ANALYSIS

We summarize the complexity analysis of IMre, DEre and Crypt-DAC in Tables 1, 2, and 3.

TABLE III: Performance of revoking a user from a role

		Crypt-DAC	Previous
RK* and *FK	**Computation**	$(Gen^{Pub}$+$Gen^{Sign})$+*mems*(*r*)×(Enc^{Pub})	**IMre:** $(Gen^{Pub}$+$Gen^{Sign})$+*mems*(*r*)×$(Enc^{Pub}$+*Sign*) +*files*(*r*)×$(Enc^{Pub}$+*Sign*) +$(\sum_{i=1}^{files(r)}$*roles*$(f_{n_i}))$×$(Enc^{Pub}$+*Sign*) **DEre:** $(Gen^{Pub}$+$Gen^{Sign})$+*mems*(*r*)×$(Enc^{Pub}$+*Sign*) +*files*(*r*)×$(Dec^{Pub}$+Enc^{Pub}+*Sign*) +$(\sum_{i=1}^{files(r)}$*roles*$(f_{n_i}))$×$(Enc^{Pub}$+*Sign*)
	Communication	*mems*(*r*)×$\lvert Enc^{Pub}\rvert$	**IMre:** *mems*(*r*)×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$ +*files*(*r*)×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$ +$(\sum_{i=1}^{files(r)}$*roles*$(f_{n_i}))$×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$ **DEre:** *mems*(*r*)×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$ +2*files*(*r*)×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$ +$(\sum_{i=1}^{files(r)}$*roles*$(f_{n_i}))$×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$
F	**Computation**	*files*(*r*)×Gen^{Sym}	**IMre:** *files*(*r*)×$(Gen^{Sym}$+Dec^{Sym}+Enc^{Sym}+*Sign*) **DEre:** —
	Communication	*files*(*r*)×$\lvert k\rvert$	**IMre:** 2×$(\lvert$*files*(*r*)$\rvert$+*files*(*r*)×$\lvert$*Sign*$\rvert)$ **DEre:** —

TABLE IV: Performance of revoking a role from a file

		Crypt-DAC	Previous
***RK* and *FK* tuples**	**Computation**	—	*files*(*r*)×$(Enc^{Pub}$+*Sign*)
	Communication	—	*files*(*r*)×$(\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert)$
***F* tuples**	**Computation**	*files*(*r*)×Gen^{Sym};	**IMre:** Gen^{Sym}+Dec^{Sym}+Enc^{Sym}+*Sign* **DEre:** —
	Communication	*files*(*r*)×$\lvert k\rvert$	**IMre:** 2×$(\lvert$*file*$\rvert$+$\lvert$*Sign*$\rvert)$ **DEre:** —

TABLE V: Performance of file reading/writing

		Crypt-DAC: (1) non-revocation; (2) revocation	Previous
File reading	**Computation**	(1): 2(Dec^{Pub}+*Verify*)+Dec^{Sym}+*Verify* (2): 2(Dec^{Pub}+*Verify*)+*version*(f_n)×Dec^{Pub}+ *version*(f_n)×Dec^{Sym}+*Verify*	2(Dec^{Pub}+*Verify*)+Dec^{Sym}+*Verify*
	Communication	(1): 2($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+$\lvert f\rvert$+$\lvert$*Sign*$\rvert$ (2): 2($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+*version*(f_n)×$\lvert Enc^{Pub}\rvert$+ $\lvert f\rvert$+$\lvert$*Sign*$\rvert$	2($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+$\lvert f\rvert$+$\lvert$*Sign*$\rvert$
File writing	**Computation**	(1): 2(Dec^{Pub}+*Verify*)+Enc^{Sym}+*Sign* (2): *roles*(f_n)×(Enc^{Pub}+*Verify*)+Enc^{Sym}+*Sign*	2(Dec^{Pub}+*Verify*)+Enc^{Sym}+*Sign*
	Communication	(1): 2($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+$\lvert f\rvert$+$\lvert$*Sign*$\rvert$ (2): *roles*(f_n)×($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+$\lvert f\rvert$+$\lvert$*Sign*$\rvert$	2($\lvert Enc^{Pub}\rvert$+$\lvert$*Sign*$\rvert$)+$\lvert f\rvert$+$\lvert$*Sign*$\rvert$

APPENDIX B
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 [43], 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: If *op* = *Write*:
     $$\exists r\ (RK(u, r) \wedge FK(r, p))$$
     If *op* = *Read*:
     $$\exists F\ Dec(F, u)$$
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

APPENDIX C
IMPLEMENTATION OF $RBAC_0$ BY CRYPT-DAC

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

Definition 2 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*:

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

for each theory T of Crypt-DAC.

We next show that this implementation achieves correctness, AC-preserving, and safety. For the definition of congruence, please refer to [14].

Definition 3 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 $Th(x) = \pi(Th(\sigma(x))$ 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.

We prove that Crypt-DAC implements $RBAC_0$ with 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 [14], 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 [14] except **revokeUser** and **revokeRole**, we just show the two labels here.

- **revokeU:** If l is an instance of $revokeU(u,r)$, then $x' = x \setminus UR(u, r)$. 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 $T_0 = \{(u', c_{u'}, sig) \mid \langle RK, u', (r, v_r), c_{u'}, sig\rangle \in FS\}$ and $T_1 = \{f_n \mid \exists(op, c_{f_n}, sig).(\langle FK, (r, v_r), (f_n, v_{f_n}, op), v_{f_n}, c_{f_n}, sig\rangle \in FS\}$. For each $f_n \in T_1$, let $k_{f_n} \leftarrow Gen^{Sym}$, $T_{f_n} = \{(op', c_u, sig) \mid \langle FK, (r, v_r), (f_n, v_{f_n}, op'), c_u, sig\rangle \in FS\}$ and $T'_{f_n} = \{id, op', c_{id}, sig) \mid \langle FK, id, (f_n, v_{f_n}, op'), c_{id}, sig\rangle \in FS\}$. Then:

  $\sigma'(x') = \sigma'(x \setminus UR(u, r))$
  $=\sigma'(\text{next}(x, revokeU(u, r)))$

$\cong \sigma'(x) \backslash \{FS(\langle RK, u', (r, v_r), c_{u'},$
$sig\rangle) \mid (u', c_{u'}, sig) \in T_0\} \cup \{FS(\langle RK, u',$
$(r, v_r + 1), Enc^{Pub}_{ek_{u'}}(dk_{r,v_r+1}, sk_{r,v_r+1}), sig\rangle) \mid (u', c_{u'}, \text{sig}) \in T_0 \wedge u' \neq u\}$
$\backslash \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, op'), c_v,$
$sig\rangle) \mid f_n \in T_1 \wedge (op', c_v, sig) \in T_{f_n}\}$
$\cup \{FS(\langle FK, (r, v_r + 1), (f_n, v_{f_n}, op'),$
$c_v \oplus rek_r || Enc^{Pub}_{ek_{r,v_r+1}}(k_{f_n}), \text{sig}\rangle)$
$\mid f_n \in F \wedge (op', c_v, sig) \in T_{f_n}\}$
$\backslash \{FS(\langle FK, id, f_n, v_{f_n}, op'), c_{id}$
$, sig\rangle) \mid f_n \in T_1 \wedge (id, c_{id}, sig) \in T'_{f_n}\}$
$\cup \{FS(\langle FK, id, (f_n, v_{f_n}+1, op'), c_{id}$
$|| Enc^{Pub}_{ek_{id}}(k_{f_n}), sig\rangle) \mid f_n \in T_1$
$\wedge (id, c_{id}, sig) \in T'_{f_n}\}$
$\backslash \{FS(\langle F, f_n, c_{f_n}, sig\rangle) \mid (f_n) \in T_1\}$
$\cup \{FS(\langle F, f_n, Enc^{Sym}_{k_{fn}}(c_{f_n}), sig\rangle)$
$\mid (f_n) \in T_1\}$
$\cup \{FILES(f_n, v_{f_n}+1) \mid f_n \in F\}$
$\backslash \{FILES(f_n, v_{f_n}) \mid f_n \in F\}$
$\cup$ *ROLES*$(r, v_r+1, ek_{(r,v_r+1)},$
$vk_{(r,v_r+1)})$
$\backslash$ *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 $x' = x \backslash PA(r, p)$. We consider two cases:

  – If op = *RW*, then let $T = \{(r, c, sig) \mid \langle FK, (r, v_r), (f_n, v_{f_n}, RW), c, sig\rangle\}$. 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 $k' \leftarrow Gen^{Sym}$, $T = \{(op', \text{c}, sig) \mid \langle FK, (r, v_r), (f_n, v_{f_n}, op'), c, sig\rangle \in FS\}$, $T_1 = \{(id, op') \mid id \neq r \wedge \exists(c_{id}, sig).(\langle FK, id, (f_n, v_{f_n}, op'), c_{id}, sig\rangle \in FS)\}$, and $T_2 = \{(f_n) \mid \langle F, f_n,$ $, c_{f_n}, SU, sig\rangle \in FS)\}$. Then
  $\sigma'(x') = \sigma'(x \setminus PA(r, p))$
  $=\sigma'(\text{next}(x), assignP(r, p)))$
  $\cong \sigma'(x) \setminus \{FS(\langle FK, (r, v_r), (f_n, v_{f_n}, op'),$
  $c, sig\rangle) \mid (op', c, sig) \in T\}$
  $\setminus \{FS(\langle FK, id, (f_n, v_{f_n}, op'),$
  $c_{id}, sig\rangle) \mid (id, op') \in T_1\}$
  $\cup \{FS(\langle FK, id, (f_n, v_{f_n}, op'), v_{f_n}+1,$
  $c_{id} || Enc^{Pub}_{ek_{id}}(k'), sig\rangle) \mid (id, op') \in T_1\}$
  $\setminus \{FS(\langle F, f_n, c_{f_n}, sig\rangle) \mid (f_n) \in T_2\}$
  $\cup \{FS(\langle F, f_n, Enc^{Sym}_{k'}(c_{f_n}), sig\rangle)$
  $\mid (f_n) \in T_2\}$
  $\cup$ *FILES*$(f_n, v_{f_n}+1) \setminus$ *FILES*(f_n, v_{f_n})
  $=$*terminal*$(\sigma'(x), \alpha(\sigma'(x), l))$.

Definition 4 AC-preserving: An implementation with query-mapping π is called AC-preserving if for all workload states s and authorization requests r we have that $s \vDash auth(r)$ if and only if $\pi_{auth(r)}(Th(\sigma(s)))$ = true.

We 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).

Definition 5 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)

We 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*.