Encryption-based Policy Enforcement for Cloud Storage

S. De Capitani di Vimercati*, S. Foresti*, S. Jajodia†, S. Paraboschi‡, G. Pelosi‡, P. Samarati*
*DTI - Università degli Studi di Milano, 26013 Crema - Italy
†CSIS - George Mason University, Fairfax, VA 22030-4444 - USA
‡DIIMM - Università degli Studi di Bergamo, 24044 Dalmine - Italy

***Abstract*—Nowadays, users are more and more exploiting external storage and connectivity for sharing and disseminating user-generated content. To this aim, they can benefit of the services offered by Internet companies, which however assume that the service provider is entitled to access the resources. To overcome this limitation, we present an approach that does not require complete trust in the external service w.r.t. both resource content and authorization management, while at the same time allowing users to delegate to the provider the enforcement of the access control policy on their resources. Our solution relies on the translation of the access control policy into an equivalent encryption policy on resources and on a hierarchical key structure that limits both the number of keys to be maintained and the amount of encryption to be enforced.**

I. Introduction

"Cloud computing" is a relatively recent term, characterizing the collection of technologies and tools supporting the use of large scale Internet services for the remote construction of applications. The realization of cloud computing is consistent with clear technological and economic trends. The correct administration and configuration of computing systems is expensive and presents large economies of scale, supporting the centralization of resources. This is particularly significant when considering reliability and availability requirements, which are difficult to satisfy by final users and small/medium organizations. This evolution is also consistent with the vision offered in the past by most research in network and distributed systems, which assumed a continuous increase in the quality and quantity of tasks assigned to distributed components.

An important application of cloud computing is represented by cloud storage, where an Internet service allows a large open community of users to store and exchange resources (i.e., files containing images, videos, applications, and so on). While these services at the beginning were mostly used to openly publish resources, today users are more and more demanding solutions for regulating the publication and disclosure of their own content. The importance of this requirement is also testified by the recent introduction of resource sharing features in cloud applications (e.g., Google Docs - http://docs.google.com). Existing Web services offer to users a form of control on their resources, as well as existing access control solutions for social networks scenarios Typically these solutions assume that the service provider is completely trusted and always entitled to access the resources. This assumption may not always be applicable, as users may want to restrict access to the server itself, which should be able to guarantee the service without having cleartext access to the resources.

We address the need of enforcing selective access to the resources by proposing an approach that supports the user in the specification of access restrictions to resources the user wishes to share, via an external storage service, with a desired group of other users. Our proposal guarantees that only users in the specified group will be able to access the resources, which remain confidential to all the other parties, including the service itself. In synthesis, our approach assumes that resources are encrypted with a symmetric encryption algorithm. The key used to protect a resource can be derived from a secret held by each user, exploiting a Diffie-Hellman key agreement method and public tokens. The service offered in this way can be realized by any community of users desiring to exchange confidential resources. Compared to existing applications, our approach offers stronger guarantees in terms of protection of resource confidentiality, in a way which is fully compatible with the design of cloud storage applications. Our approach leverages on solutions proposed for the data outsourcing scenarios, extending them to the consideration of the presence of many users exchanging resources, each having both the role of data owner and data consumer.

The remainder of the paper is organized as follows. Section II presents some basic concepts. Section III describes the key derivation technique and the representation of an authorization policy via proper encryption. Section IV describes the algorithms for encrypting resources and accessing them. Section V analyzes the security issues of our model. Section VI presents the performance results obtained by a prototype implementing the proposed algorithms. Section VII describes related work. Finally, Section VIII draws our conclusions.

II. Scenario and basic concepts

We consider a scenario where there is a set $\mathcal{U}$ of users who wish to selectively share their resources among themselves. Each user may then play two different roles: *data owner* and *data consumer*. A user plays the role of data owner when she makes her resources available to other users in the system. A user plays the role of data consumer when she requires access to resources owned by others. In the following, we use the term user when it is not needed to make any distinction between the two roles above. This sharing process is clearly selective, that is, each resource might be accessible only to a subset of users in $\mathcal{U}$, as defined by the data owner. We assume that a resource can be modified only by its owner. Whenever a data owner wishes to share a resource with other users in the system, the management of the resource is delegated to an *external service*. While the service is trusted with respect to the management of the resources, it is not trusted to manage authorization policies and should not be allowed to access the resource content (*honest-but-curious* service). Also, the service is supposed not to prevent access to the resources that each user is authorized to view. In the following, given a user $u \in \mathcal{U}$, R_u denotes the set of resources for which u is the owner. The set of resources managed by the service is denoted by $\mathcal{R}$. Notation $owner(r)$, with $r \in \mathcal{R}$, represents the owner $u \in \mathcal{U}$ of r.

Each user u can define an *authorization policy* that regulates who can access her resources.

Definition II.1 (Authorization policy)*: Given a data owner $u \in \mathcal{U}$, the* authorization policy *defined by u over R_u, denoted P_u, is a set of pairs of the form $\langle u_i, r_j \rangle$, where $u_i \in \mathcal{U}$ and $r_j \in R_u$.*

The semantics of an authorization $\langle u_i, r_j \rangle \in P_u$ is that data owner u has granted to user u_i the permission to access resource r_j. In the following, given a resource $r \in \mathcal{R}$, with $u = owner(r)$, $acl(r)$ denotes the *access control list* of resource r, that is, the set of users that can access r according to the authorization policy P_u. The owner of a resource is always permitted to access the resource content and appears as a member of the corresponding acl.

Example II.1*: Consider a system with five users $\mathcal{U}=\{A,B,C,D,E\}$, $R_A=\{r_1,r_2\}$, $R_B=\{r_3,r_4\}$, $R_C=\{r_5\}$, $R_D=R_E=\emptyset$. The authorization policies defined by A, B, and C are:*

- $P_A = \{\langle A,r_1\rangle,\langle B,r_1\rangle,\langle A,r_2\rangle,\langle B,r_2\rangle,\langle C,r_2\rangle\}$;
- $P_B = \{\langle B,r_3\rangle,\langle D,r_3\rangle,\langle E,r_3\rangle,\langle A,r_4\rangle,\langle B,r_4\rangle,\langle C,r_4\rangle\}$;
- $P_C = \{\langle A,r_5\rangle,\langle B,r_5\rangle,\langle C,r_5\rangle,\langle D,r_5\rangle,\langle E,r_5\rangle\}$.

According to these policies, $acl(r_1)=\{A,B\}$, $acl(r_2)=\{A,B,C\}$, $acl(r_3)=\{B,D,E\}$, $acl(r_4)=\{A,B,C\}$, and $acl(r_5)=\{A,B,C,D,E\}$

Our goal is to realize a mechanism that allows data owners to share their resources in such a way that only authorized users can access the resources and that the service in charge for their management has no access to the resource content. To this purpose, we exploit encryption as a protection mechanism for enforcing an authorization policy. Encryption is applied with two objectives in mind: *i)* the efficiency, to minimize the number of keys that each user in the system has to manage; *ii)* the correct enforcement of the authorization policy defined by the owner, to guarantee that a resource is accessible to all and only the users specified in the corresponding acl. In the following, we illustrate an encryption schema correctly enforcing the authorization policies and such that each user maintains a *single secret* and each resource is encrypted by using a single key only.

III. Encryption schema

In the considered scenario there are different owners responsible for different portions of the resources publicly available. A simple solution for sharing resources in a selective way consists in applying the approaches developed for the outsourced scenario [5], where a single owner, before outsourcing her resources, encrypts them with different keys and each authorized user has a key from which she can derive all the keys of the resources she is authorized to access. While simple, the application of this solution in our context requires each user to manage a potentially large number of keys (in the worst case, one key for each owner in the system). We then propose a novel solution that exploits two cryptographic techniques: a *key agreement method* allows two users to share a secret key for subsequent cryptographic use; a *key derivation method* adopts the secret keys shared between pairs of users for allowing users to derive all keys used for encrypting resources that they are authorized to access. The combination of these two techniques results in an *encryption policy* that correctly enforces the authorization policies defined by the data owners (see Section III-C).

A. Key agreement

Our key agreement method is based on a slight variation of the Diffie-Hellman (DH) key agreement method, where the two involved parties do not directly interact for computing the common secret, but they interact with the external service. Our variation of the DH method works as follows. Let $(\mathbb{G}, \cdot)$ be a public algebraic cyclic group of prime order $q = |\mathbb{G}|$ and $\cdot$ be the internal operation of the group with multiplicative notation. We assume that $\mathbb{G}$ is generated by an element $g \in \mathbb{Z}_p$ (with $p = 2q + 1$ and p, q two prime integers) in such a way that $q = |\mathbb{G}|$ and $\mathbb{G} = \{g^e \mod p : 0 \leq e \leq q-1\}$.

Each user $u \in \mathcal{U}$ chooses a secret integer parameter $e_u \in [0, q-1]$, computes the value $g^{e_u} \in \mathbb{G}$, and inserts

g^{e_u} in a public catalog managed by the external service. The external service also keeps track of the public parameters g and q (note that for simplicity we assume that the algebraic cyclic group $(\mathbb{G}, \cdot)$ is unique in the system). Whenever user u needs to share a common secret with user u_i, user u can efficiently compute such a secret by querying the public catalog to retrieve the public parameters $g^{e_{u_i}}$ and q, and by applying the following *key agreement function*.

Definition III.1 (Key agreement function)*: Given a set $\mathcal{U}$ of users, a set $\mathcal{K}$ of keys, and a public algebraic cyclic group $(\mathbb{G}, \cdot)$ of prime order q, with generator $g \in \mathbb{G}$, the* key agreement function *of a user $u \in \mathcal{U}$ is a function $ka_u : \mathbb{G} \mapsto \mathcal{K}$ that takes the public parameter $g^{e_{u_i}} \in \mathbb{G}$ of a user $u_i \in \mathcal{U}$ as input and returns the common secret between u and u_i computed as: $ka_u(g^{e_{u_i}}) = (g^{e_{u_i}})^{e_u}$.*

Note that according to Definition III.1, for all pairs of users u_i,$u_j \in \mathcal{U}$, $u_i \neq u_j$, $ka_{u_i}(g^{e_{u_j}}) = ka_{u_j}(g^{e_{u_i}})$. In the following, notation $\mathcal{KA}$ is used to denote the set of key agreement functions of all users in $\mathcal{U}$.

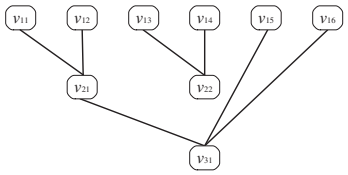
B. Key derivation

A key derivation method allows the computation of a key starting from the value of another key and a publicly available piece of information, called *token*. Given a set $\mathcal{K}$ of keys and k_i,$k_j \in \mathcal{K}$, a token $t_{i,j}$ between them is defined as $t_{i,j} = E_{k_i}(k_j)$, where E is a symmetric encryption function.[1] The existence of $t_{i,j}$ allows each user knowing k_i to derive key k_j by simply decrypting $t_{i,j}$ with k_i. Key derivation via tokens can be applied in chains: a chain of tokens is a sequence $t_{i,l}, \ldots, t_{n,j}$ of tokens such that $t_{c,d}$ directly follows $t_{a,b}$ in the chain only if $b = c$. The concept of key derivation via chains of tokens is formally captured by the following definition of *key derivation function*.

Definition III.2 (Key derivation function)*: Given a set $\mathcal{K}$ of keys, and a set $\mathcal{T}$ of tokens, the* direct key derivation function $\tau : \mathcal{K} \mapsto 2^{\mathcal{K}}$ *is defined as $\tau(k_i) = \{k_j \in \mathcal{K} : \exists t_{i,j} \in \mathcal{T}\}$. The* key derivation function $\tau^* : \mathcal{K} \mapsto 2^{\mathcal{K}}$ *is such that $\tau^*(k_i)$ is the set of keys derivable from k_i by chains of tokens, including the key itself (chain of length 0).*

Graphically, a set $\mathcal{K}$ of keys and a set $\mathcal{T}$ of tokens can be represented via a *key and token graph*, with a vertex v_i for each key $k_i \in \mathcal{K}$, and an edge (v_i, v_j) for each token $t_{i,j} \in \mathcal{T}$. We call *root* a vertex in the key and token graph that does not have incoming edges (i.e., a vertex whose key cannot be derived via tokens). Chains of tokens correspond to paths in the graph and the key derivation function $\tau^*(k)$ associates with each key $k \in \mathcal{K}$ the keys of vertices reachable from the vertex associated with k in the graph. Figure 1(a) illustrates an example of key and token graph, where notation v_{ij} is used to denote the j-th vertex (from left to right) on the i-th level of the graph, and k_{ij} denotes the key associated with vertex v_{ij}. The root vertices of the graph are at level 1 (i.e., v_{1j}, $j = 1, \ldots, 6$). Note that for readability of the figure, arrows do not appear in the graph. The graph is oriented from top to bottom.

[1]Tokens can be defined according to different strategies (e.g., [1]).



r	$\phi(r)$
r_1	k_{11}
r_2	k_{21}
r_3	k_{22}
r_4	k_{21}
r_5	k_{31}

(b)

Figure 1. An example of key and token graph (a) and key assignment function (b)

The definition of tokens can support the general goal of encrypting resources by using a single key. The idea is that whenever a resource r, with u=*owner*(r), must be accessible to n users $u_1, \ldots, u_n$, the owner u can encrypt r with a key $k \in \mathcal{K}$ and can compute a set of tokens that each user u_i, $i = 1, \ldots, n$, can then use for deriving key k. For instance, according to the authorization policy P_A in Example II.1, user A can encrypt her resource r_2 with a key $k \in \mathcal{K}$ and then can define two tokens, from $ka_A(g^{e_B})$ to k and from $ka_A(g^{e_C})$ to k, that users B and C, respectively, can exploit for deriving k.

A *key assignment function* determines the keys used for encrypting resources and is defined as follows.

Definition III.3 (Key assignment function)*: Given a set $\mathcal{R}$ of resources and a set $\mathcal{K}$ of keys, the* key assignment function $\phi : \mathcal{R} \mapsto \mathcal{K}$ *associates with each resource $r \in \mathcal{R}$ the (single) key with which the resource is encrypted.*

Figure 1(b) illustrates an example of key assignment function defined over the resources of Example II.1. It is easy to see that the key used for encrypting r_5 (i.e., k_{31}) can be derived from keys k_{11}, k_{12}, k_{15}, and k_{16}.

C. Encryption policy

An encryption policy regulates which resources are encrypted with which keys and which keys can be directly or indirectly computed by which users. Formally, an encryption policy is defined as follows.

Definition III.4 (Encryption policy)*: Given a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources, an* encryption policy *over $\mathcal{U}$ and $\mathcal{R}$, denoted $\mathcal{E}$, is a 6-tuple of the form $\langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{T}, \mathcal{KA}, \phi \rangle$, where $\mathcal{K}$ is a set of keys, $\mathcal{T}$ is a set of tokens defined over $\mathcal{K}$, $\mathcal{KA}$ is the set of key agreement functions of all users in $\mathcal{U}$, and ϕ is a key assignment function.*

An encryption policy can be represented via a graph, called *encryption policy graph*, obtained from the key

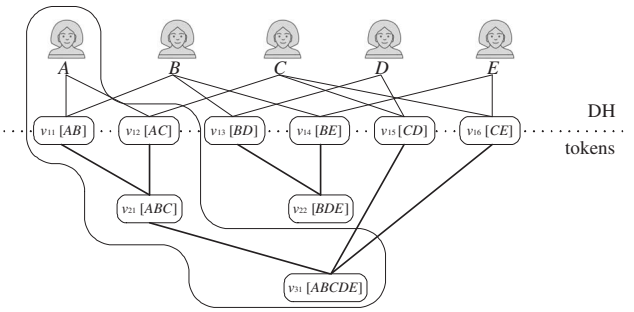


Figure 2. An encryption policy graph

User

user_id	public
A	g^{e_A}
B	g^{e_B}
C	g^{e_C}
D	g^{e_D}
E	g^{e_E}

Token

source	destination	token_value
AB	ABC	$E_{k_{11}}(k_{21})$
AC	ABC	$E_{k_{12}}(k_{21})$
BD	BDE	$E_{k_{13}}(k_{22})$
BE	BDE	$E_{k_{14}}(k_{22})$
CD	$ABCDE$	$E_{k_{15}}(k_{31})$
CE	$ABCDE$	$E_{k_{16}}(k_{31})$
ABC	$ABCDE$	$E_{k_{21}}(k_{31})$

Resource

res_id	owner	label	enc_res
r_1	A	AB	α
r_2	A	ABC	β
r_3	B	BDE	δ
r_4	B	ABC	ε
r_5	C	$ABCDE$	ζ

Figure 3. An encryption policy catalog

and token graph corresponding to $\mathcal{K}$ and $\mathcal{T}$ by adding a vertex for each user $u \in \mathcal{U}$, and by adding an edge from each vertex representing u to vertices representing keys $ka_u(g^{e_{u_i}})$, for all $u_i \in \mathcal{U}$, $u \neq u_i$. The vertices representing pairs of users are inserted in the graph if and only if there is at least a token starting from them. Figure 2 illustrates an example of encryption policy graph, where each vertex has been labeled with the set of users who know or can derive the corresponding key, thick edges represent tokens, and thin edges represent the computations of the key agreement functions. Note that the information about the users who can derive the key associated with a specific vertex does not necessarily coincide with the real identities of the users. As a matter of fact, each user can be identified via a pseudonym that may be selected by the user herself. Also, the root vertices of the key and token graph are the vertices representing the keys computed through the key agreement functions in $\mathcal{KA}$; these keys need to be directly computed by the users and do not exploit tokens.

It is easy to see that each user u can directly or indirectly compute the keys associated with vertices along the paths starting from the vertex representing u. The first step is always a Diffie-Hellman computation whose resulting key is the starting point of the token chains followed by the user. Formally, the set of keys that a user can derive is defined as follows.

Definition III.5 (User keys)*: Given an encryption policy* $\mathcal{E}=\langle \mathcal{U},\mathcal{R},\mathcal{K},\mathcal{T},\mathcal{KA},\phi \rangle$*, the set of keys that a user* $u \in \mathcal{U}$ *can compute, denoted* K_u*, is defined as* $\bigcup \tau^*(ka_u(g^{e_{u_i}}))$*:* $u_i \in \mathcal{U}, u_i \neq u$*, and* $ka_u(g^{e_{u_i}}) \in \mathcal{K}$*.*

Each user u can then access any resource r such that $\phi(r) \in K_u$. For instance, with respect to the encryption policy graph in Figure 2, the portion of the graph that user A can exploit for key derivation is delimited by a continuous line and contains the set $K_A=\{k_{11},k_{12},k_{21},k_{31}\}$ of keys she can compute.

Our goal is then to translate the authorization policies defined by the users in $\mathcal{U}$ into a *correct* encryption policy $\mathcal{E}$. The concept of encryption policy correctness is formally defined as follows.

Definition III.6 (Correctness)*: Given a set* $\mathcal{U}$ *of users, a set* $\mathcal{R}$ *of resources, a set* $\mathcal{P}=\bigcup_{u \in \mathcal{U}} P_u$ *of authorization policies, and an encryption policy* $\mathcal{E}=\langle \mathcal{U},\mathcal{R},\mathcal{K},\mathcal{T},\mathcal{KA},\phi \rangle$ *over* $\mathcal{U}$ *and* $\mathcal{R}$*, we say that* $\mathcal{E}$ correctly enforces $\mathcal{P}$ *iff the following conditions hold:*

- Soundness*:* $\forall u \in \mathcal{U}$*,* $r \in \mathcal{R}$*:* $\phi(r) \in K_u \Rightarrow \langle u,r \rangle \in \mathcal{P}$*.*
- Completeness*:* $\forall u \in \mathcal{U}$*,* $r \in \mathcal{R}$*:* $\langle u,r \rangle \in \mathcal{P} \Rightarrow \phi(r) \in K_u$*.*

For instance, the encryption policy graph in Figure 2 and the key assignment function in Figure 1(b) represent an encryption policy correctly enforcing the authorization policies defined in Example II.1.

To allow users to derive the keys needed for accessing the resources, a portion of the encryption policy must be publicly available from the external service responsible for resource management. This public information is represented as a *catalog* composed of three tables: User, Resource, and Token. Table User contains a tuple for each user in $\mathcal{U}$ and has two attributes: *user_id* is the user identifier and *public* is the public parameter of the user. Table Resource contains a tuple for each resource in $\mathcal{R}$ and is characterized by four attributes: *res_id* is the resource identifier; *owner* is the identifier of the user who published the resource; *label* is the label of the vertex in the encryption policy graph whose corresponding key is $\phi(r)$; and *enc_res* is the encrypted resource together with its public-key signature.[2] Table Token contains a tuple for each token in $\mathcal{T}$ and is characterized by three attributes: *source* and *destination* are the labels of the corresponding source and destination vertices in the encryption policy graph; *token_value* is the token value computed as $E_{k_{source}}(k_{destination})$. Figure 3 illustrates the public catalog corresponding to the key assignment

[2]To enable assessing resources integrity, we assume resources to be signed by their owner with the *DSA* signature scheme as follows: *1)* Diffie-Hellman e_u and g^{e_u} parameters can be used as DSA private and public key, respectively; and *2)* the Diffie-Hellman public parameters g and q are chosen to satisfy the security criteria needed to use them also as DSA public parameters.

function in Figure 1(b) and the encryption policy graph in Figure 2.

IV. Resource management

Our approach provides the users with the functionality for *publishing* and *accessing* resources. The publish functionality allows data owners to compute the digest, sign, and correctly encrypt their resources (Definition III.6) and to deliver the encrypted resources to the service for their management. The realization of this functionality is complicated by the fact that each user u can only operate on a subset of the whole encryption policy graph. As a matter of fact, each user u is only able to first create and then use token chains whose starting points are the root vertices corresponding to keys that u can compute through Diffie-Hellman computations (i.e., root vertices representing u and another user in the system). Therefore, whenever user u needs to share a resource r with other users in the system, the user must first encrypt r with a new key and then must add the appropriate tokens that the other users in $acl(r)$ can use to derive the new key. The creation of these new tokens can exploit token chains previously created by u and ending in vertices representing a subset of the users in $acl(r)$. Note that if there already exists a key only derivable by users in $acl(r)$, it is sufficient for user u to compute such a key through the appropriate token chain and then encrypt r with the derived key.

The access functionality allows users to retrieve the resources that they are authorized to access and to verify their signature. In particular, every time an authorized user u needs to access a resource r, the service has to deliver the encrypted resource to u along with a token chain ending to the vertex representing $acl(r)$, which the user follows to derive the decryption key. User u can then decrypt the resource and use the public Diffie-Hellman parameter of $owner(r)$ to verify the signature of r.

A. Publishing resources

The publish functionality receives as input a resource r, the identifier of its owner o=$owner(r)$, the private Diffie-Hellman parameter e_o of o, and acl, with acl=$acl(r)$. It publishes an encrypted version of r obtained by signing the digest $h(r)$ of the resource with e_o and encrypting r and its signature with a key derivable only by users in $acl(r)$. Figure 4 illustrates procedure **Publish** that implements this function. Two cases may occur. In the first case, acl includes only user o and another user $u \in \mathcal{U}$. After retrieving from table User the public parameter g^{e_u}, the procedure assigns $ka_o(g^{e_u})$ to $\phi(r)$. It then computes the digest of r and its signature (function **Sign**) and encrypts r and its signature with $ka_o(g^{e_u})$. In the second case, acl includes more than two users. The procedure verifies whether there already exists a key that can be derived by users in acl. To this purpose, the procedure calls function **Find_Chain** that receives as input the owner o and acl and returns (if it exists) the shortest token chain from a vertex whose key can be directly computed by user o through ka_o to the vertex with label acl. This function first extracts from table Token all tokens potentially useful, that is, tokens starting and ending in vertices that represent sets of users containing user o and any other subset of acl. The function then operates on these tokens, which graphically form a graph, and computes a shortest path (through an improved version of Dijkstra working on DAGs that exploits the inverse topological order of vertices) from acl (if it exists as destination of a token in table Token) to a root vertex containing the owner. Let cur be the root vertex whose label includes user o such that the distance from acl is minimum. The function then builds the path from vertex cur (if it exists) to vertex acl. At each iteration of the while loop, the function follows $succ[cur]$, which is an array that contains the label of the successor of vertex cur in the path previously computed, and adds to queue $chain$ the token in Token from vertex cur to vertex $succ[cur]$. Finally, the function returns queue $chain$. Procedure **Publish** then proceeds in two different ways, depending on the value of $chain$.

If the returned $chain$ is not empty, the procedure calls function **Compute_Key** that by following the chain derives the key associated with vertex acl. This derivation is performed by first applying a Diffie-Hellman computation (the starting key of the chain) and then by following the token chain. The computed key is finally assigned to $\phi(r)$.

If the returned $chain$ is empty (i.e., there is no key derivable by users in acl), a new key is randomly generated and assigned to $\phi(r)$. Procedure **Publish** then verifies whether the encryption policy graph includes token chains that can be exploited for allowing users in acl to derive key $\phi(r)$. Intuitively, a token chain is useful when it starts from a vertex whose key can be directly computed by user o (i.e., the label of the vertex includes o) and terminates with a vertex representing a set acl' of users that is a subset of acl. In this case, user o can follow the chain and derive the key associated with vertex acl' that is then used for computing the token from vertex acl' to the new vertex representing acl. The presence of this new token permits to all users in acl' to derive key $\phi(r)$. The search of the useful token chains is realized through function **Find_Sources** that takes as input owner o and acl and returns a set $parents$ of labels of vertices that represent subsets of acl containing o. Procedure **Publish** then computes the new tokens from the vertices whose labels are in $parents$ to the new vertex representing acl as follows. Variable to_cover is

```
PUBLISH(r,o,e_o,acl):void /* Owner */
Case |acl| of
= 2: /* Case 1: acl with two users */
     let ut∈USER : ut[user_id]=acl\{o} /* query USER table */
     φ(r) := ka_o(ut[public])
> 2: /* Case 2: acl with more than two users */
     chain := Find_Chain(o,acl) /* query TOKEN table */
     Case chain of
     ≠ ∅: /* Case 2.1: ∃ a key derivable by users in acl */
          φ(r) := Compute_Key(o,chain)
     = ∅: /* Case 2.2: ∄ a key derivable by users in acl */
          randomly generate a key key_r
          φ(r) := key_r
          parents := Find_Sources(o,acl)
          to_cover := acl
          for each p∈parents do
            chain := Find_Chain(o,p) /* query TOKEN table */
            k_p := Compute_Key(o,chain)
            t[source] := p
            t[destination] := acl
            t[token_value] := E_{k_p}(key_r)
            insert t into TOKEN
            to_cover := to_cover \ p
          for each u∈to_cover do
            let ut∈USER : ut[user_id]=u /* query USER table */
            k := ka_o(ut[public])
            t[source] := {u,o}
            t[destination] := acl
            t[token_value] := E_k(key_r)
            insert t into TOKEN
signature := Sign(r,e_o)
rt[res_id] := r
rt[owner] := o
rt[label] := acl
rt[enc_res] := E_{φ(r)}(r, signature)
insert rt into RESOURCE

ACCESS(u,r,o):resource content /* Authorized users */
/* query RESOURCE table */
let rt∈RESOURCE : rt[res_id]=r and rt[owner]=o
if u ∉ rt[label] then return(∅)
r^k := rt[enc_res]
Case |rt[label]| of
= 2: let t∈USER : t[user_id]=rt[label]\{u} /* query USER table */
     k := ka_u(t[public])
> 2: chain := Find_Chain(u,rt[label]) /* query TOKEN table */
     k := Compute_Key(u,chain)
⟨r',signature⟩ := D_k(r^k)
let ut∈USER : ut[user_id]={o} /* query USER table */
if Verify(signature, ut[public],r') then return(r')
else notify integrity breach

FIND_CHAIN(u,acl):token chain /* Service */
V := E := ∅
for each t in TOKEN do /* find potentially useful tokens */
  if u∈t[source] ∧ t[source]⊆acl then
    V := V ∪ t[source]
    if t[destination]⊆acl then
      V := V ∪ t[destination]
      E := E ∪ t
if acl∉V then return(∅)
topologically sort V
/* retrieve the shortest path from acl to a root vertex of u */
for each acl'∈V do
  dist[acl'] := ∞
  succ[acl'] := NULL
dist[acl] := 0
for each acl_i∈V do/* visit vertices in inverse topological order */
  for each (acl_j,acl_i)∈E do /* the weight of each edge is 1 */
    if dist[acl_j]>dist[acl_i]+1 then
      dist[acl_j] := dist[acl_i]+1
      succ[acl_j] := acl_i
chain := ∅
let cur ∈ V : dist[cur] is minimum, u∈cur, cur is a root vertex
while cur≠NULL ∧ cur≠acl do
  let t∈TOKEN : t[source]=cur ∧ t[destination]=succ[cur]
  Enqueue(chain,t)
  cur := succ[cur]
return(chain)

COMPUTE_KEY(u,chain):key /* Authorized users */
t := Dequeue(chain)
let ut∈USER : ut[user_id]=t[source]\{u}/* query USER table */
k := ka_u(ut[public]) /* first key of the chain computed via DH */
repeat /* key derivation through token chain*/
  k := D_k(t[token_value])
  t := Dequeue(chain)
until t=NULL
return(k)

FIND_SOURCES(u,acl):sets of users /* Service */
candidates:= ∅
for each t∈TOKEN do /* find potentially useful tokens */
  if u∈t[source] ∧ t[source]⊆acl then
    if t[destination]⊆acl then
      candidates := candidates ∪ t[destination]\t[source]
    else candidates := candidates ∪ t[source]
sort sets in candidates in decreasing order of their cardinality
parents := ∅
for each acl'∈candidates do
  if acl'∩acl≠∅ then
    parents := parents ∪ {acl'}
    acl := acl \ acl'
return(parents)
```

Figure 4. Procedures and functions for publishing and accessing resources

used to keep track of the users that are not yet able to derive key $\phi(r)$ and is then initialized to *acl*. For each p in *parents*, procedure **Publish** first computes, through functions **Find_Chain** and **Compute_Key**, key k_p associated with p, and then computes the new token from p to *acl*, which is inserted in table TOKEN and removes p from *to_cover*. If after the analysis of all elements in *parents*, *to_cover* is not empty, for each user $u \in$ *to_cover* the procedure inserts a token from the vertex representing users $\{o, u\}$ to the vertex representing *acl*. Finally, function **Sign** computes the signature for r, encrypts r and its signature with $\phi(r)$, and inserts the resulting ciphertext into table RESOURCE.

Example IV.1*: Figure 5 illustrates the evolution of an encryption policy graph following a sequence of publish operations. There are five users, $\mathcal{U}$=$\{A,B,C,D,E\}$, and at the initial state no resource has yet been published. Upon each publication, if there is no key for the involved acl, a new key is generated together with the tokens allowing derivation of the key from all the users in the acl. The figure also reports the keys that must be computed between pairs of users for ensuring such a derivation.*

As stated by the following theorem, the procedure and functions for publishing resources in Figure 4 preserve

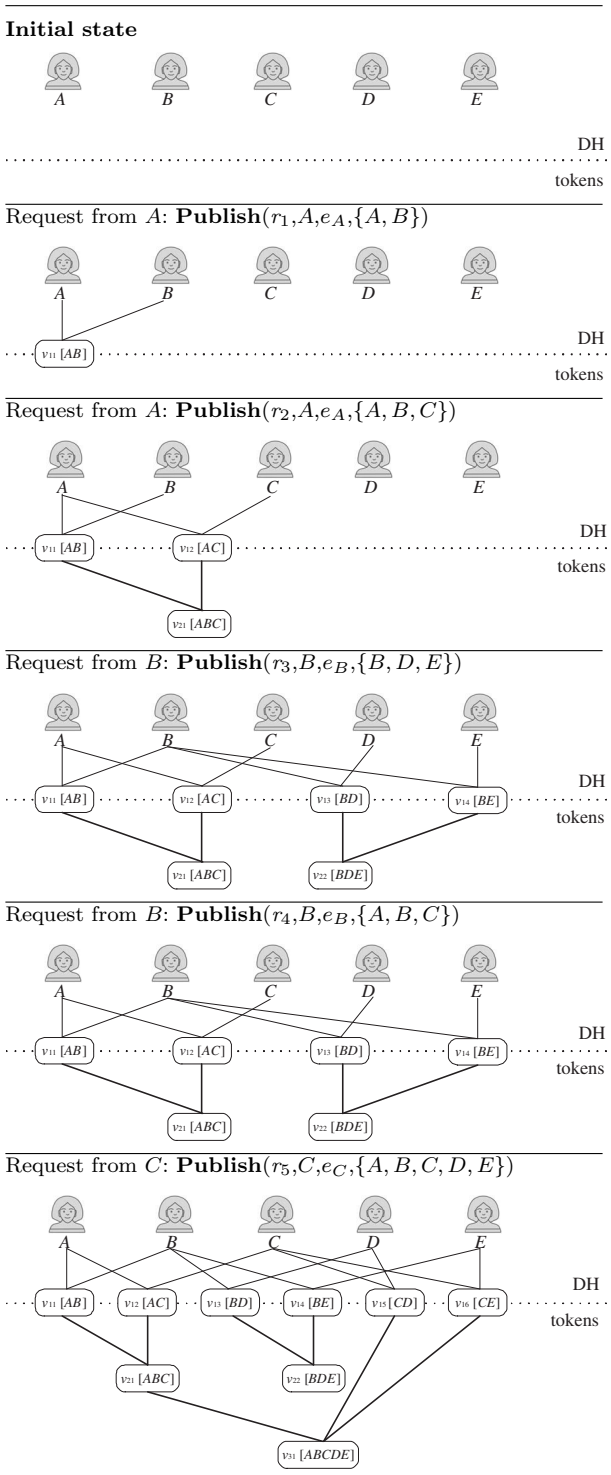


Figure 5. A sequence of publishing operations

the correctness of the encryption policy.

Theorem IV.1 (Correctness)*: Given an encryption policy $\mathcal{E}$ correctly enforcing a set $\mathcal{P}$ of authorization policies, procedure* **Publish** *generates a new encryption policy $\mathcal{E}'$ that is correct (Definition III.6) with respect to $\mathcal{P}'=\mathcal{P}\cup\{\langle u,r\rangle : u\in acl(r)\}$.*

Proof: Omitted for space constraints. ■

B. Accessing resources

The access functionality is implemented by function **Access** in Figure 4 that takes as input a user u, a resource r, and the identifier of the owner $o=owner(r)$ of r. It returns the resource content if u is authorized to access r and r has not been modified by a user different from o. The function first extracts from table RESOURCE the tuple rt corresponding to r (i.e., $rt[res\_id]=r$ and $rt[owner]=o$). If u is not a member of $rt[label]$ (i.e., u does not belong to $acl(r)$), the function returns the empty set. Otherwise the function checks the cardinality of acl. If acl includes two users, the function computes the decryption key through ka_u. Otherwise, the decryption key is derived by first calling function **Find_Chain** for retrieving the token chain, and then by calling function **Compute_Key** for computing the decryption key. Function **Access** decrypts r^k with the computed key, obtaining both resource r and its signature. If the signature is correctly validated with the public parameter of the owner o, function **Access** returns r, otherwise it notifies an integrity breach.

Example IV.2*: Consider the state resulting from the sequence of operations of Example IV.1 and the request of user B to access r_2. Since $B\in acl(r_2)$, function* **Access**(B, r_2, A) *extracts ${r_2}^k$ from table* RESOURCE*. The function then calls function* **Find_Chain***, which returns the shortest path from vertex v_{11} to vertex v_{21} (i.e., the vertex whose key is used for encrypting r_2). The function then derives k_{21} from k_{11} by calling function* **Compute_Key**. *${r_2}^k$ is decrypted using k_{21} and the resulting signature is verified with the public parameter of user A. Finally, r_2 is returned to B.*

V. SECURITY ANALYSIS

Traditionally, threats in key-agreement or key-distribution schemes are represented by adversaries who can eavesdrop, replay, or substitute messages that are transmitted over a single communication channel. In our scenario, there is no single channel to protect, since each user accesses the cloud storage service by an independent, and possibly varying, location.

We consider the threats due to a subject (service or user) that aims at acquiring access to a resource for which she does not have authorization (i.e., she does not belong to the acl of the resource). We assume *encryption to be robust* and the *key derivation method to be secure* [1], [2]. We also assume, as it is typical for Internet services interested in security, that communication between the user and the service occurs over a *SSL channel* where the service presents a certificate signed

by an authority recognized by the user; impersonation of the service can therefore be excluded.

The way a malicious party can obtain access to a key is by *masquerading* as a legitimate user so that other users in the system are provided the public DH parameter of the malicious party instead of the genuine one. We do not consider the problem of users claiming an identity they do not own (like the phony celebrity pages in Facebook and MySpace), since it lies outside of the technical realm we are considering in this analysis. We are instead interested in the technical problem of preventing the service to behave maliciously, compromising the confidentiality of resources by presenting DH parameters that the service controls. This problem can be addressed by applying different strategies. A first possibility consists in adopting traditional techniques, such as off-band user-to-user communication on a trusted channel of the public DH parameters, or their distribution via certification authorities, which however are characterized by significant costs. A second possibility consists in exploiting the *Identity-Based Encryption* (IBE) paradigm to ensure a non-forgeable binding between a DH parameter and the identity of the user. An IBE-based solution avoids the use of certificates but it does not fit properly the cloud storage scenario, since it basically shifts user trust from the storing service provider to a trusted authority who is responsible for the management and distribution of private keys to users.

A third possibility exploits the ability of users to query the public table USER *anonymously*. The cloud storage service is able to control all the communication channels employed by users and can fully manipulate the content of the public tables. While the service appears extremely powerful, we assume the service has a strong incentive to have a good reputation among the users, as it is sufficient for a limited number of users to report an improper behavior of the service to have all other users lose confidence in it. According to this observation, our approach is based on the execution of random checks by the users on the honest behavior of the service as follows. Suppose a service S behaves maliciously towards the goal of accessing a resource (or a set of resources) to which a legitimate victim B should have access. The attack can be directed to resources owned by a particular user A or to all resources that B could access. Let us consider first the attack aimed at acquiring the resources of a specific user A, and then generalize the treatment. To play a *man-in-the-middle attack*, masquerading as B with respect to A, upon A's request to retrieve the public parameter of B (i.e., g^{e_B}), the service will have to respond with a fake (own) public parameter $g^{e'_B}$. After this, the service will be able to derive the common key between B and A, and therefore access resources whose acl is equal to, or contains $\{A,B\}$. However, to avoid

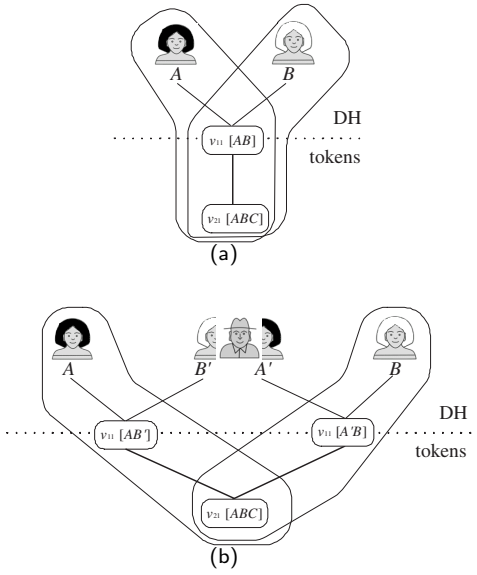


Figure 6. Man-in-the-middle attack

being detected by B, the service should ensure B's ability to access the resources that A wishes to share with B. This implies that service S needs also to encrypt the resources with a key that both S and B can determine. Hence, S will have to masquerade as A for B and will have to: *1)* sign a copy of A's resource (which S can now acquire) with a fake Diffie-Hellman private parameter e'_A (instead of the genuine e_A); *2)* encrypt the copy of A's resource and its signature to be made accessible to B; *3)* provide to B the fake public parameter $g^{e'_A}$ allowing B to verify the signature and determine the key with which the copies of the resources have been encrypted; *4)* create a new copy of all the tokens that were supposed to originate from the authentic common key between B and A so that they originate from the fake key (actually agreed between B and the service). These tokens are needed to ensure B will be able to access all resources whose acl properly contains $\{A,B\}$. This attack then implies that the service will always have to return the fake B's public parameter $g^{e'_B}$ to A, and the fake A's public parameter $g^{e'_A}$ to B while instead returning the correct g^{e_B} and g^{e_A} to other users. Note that due to the symmetric nature of the attack, by aiming at acquiring access to A's resources accessible to B, the service acquires also access to B's resources accessible to A. Note also that this symmetric behavior makes the attack applicable only to pairs of users that have never shared resources before. If the service aims at accessing all the resources to which B has access, S should mount a similar attack for all other users in the system. Figure 6 illustrates an example of encryption policy graph in the case a malicious service mounts a

man-in-the-middle attack between users A and B. Our protection relies on two easily enforceable assumptions. First, we assume that public parameter requests to the service can be made anonymously (e.g., via a proxy or a mixing protocol [6]), so that the service will not be able to know which user is submitting the request for a key. Second, we assume users can randomly query the service for their own key or for other keys they already know. With respect to our example, the service will not know if the request for B's public parameter comes from a user different from A, including B, (and therefore it should respond g^{e_B}) or from A (and therefore it should respond $g^{e'_B}$). While the service can try to guess which is the source of the request, it is reasonable to expect a non-negligible probability p_w of wrong guess. It is then possible to put constraints on the number of attacks that the service would be able to realize without being detected; with a simple statistical model, we obtain that $\lceil 1/p_w \rceil$ checks will be sufficient to detect with at least $1 - 1/e$ probability (0.632) the illicit behavior.[3] The probability of the attack going detected quickly increases with the increase of the number of anonymous retrievals of keys. Hence, since the service has a strong incentive to keep its reputation intact, it is clearly forced to avoid the man-in-the-middle attack.

VI. Experimental results

We implemented the algorithms presented in Section IV, realizing the service storing resources in Java and the client application requesting access to them as a Mozilla Firefox™ extension, with a binding to binary libraries written in C++ for the realization of the cryptographic primitives. We ran the suite in a general purpose 1Gbps Ethernet, using machines with an Intel® Core™ 2 Duo CPU at 2.40GHz, with 4GB RAM, and a Seagate Barracuda 7200.9 3Gb/s SATA 160GB hard drive with measured sustained data rate of 75.27 MB/s (write), 93.73MB/s (read), 4.2ms average latency, 11ms average seek time and 2MB cache.

Given an encryption policy graph with $|\mathcal{U}| \geq 2$ users, the number of resource groups (vertices of the graph) ranges in the interval $[1,\ 2^{|\mathcal{U}|} - |\mathcal{U}| - 1]$. The depth of an encryption policy graph depends on the sequence of the publish operations and ranges in the interval $[1,\ |\mathcal{U}|]$ with an average value equal to $|\mathcal{U}|/2$. Previous experiments in a different dissemination scenario have shown that key derivation graphs are usually characterized by a depth at most equal to 4 [4]. We then set a representative case study with an encryption policy graph of depth 4, considering $|\mathcal{U}| = 8$ users and publishing a total of 128 groups of resources having acls randomly selected in the power set of users. We stored several documents with size in the range [1KB, 100MB] for each group of resources.

[3]If p_w is equal to $1/n$, the probability for the server of guessing n times is $(1 - 1/n)^n$, which is a known mathematical series approximating e^{-1}.

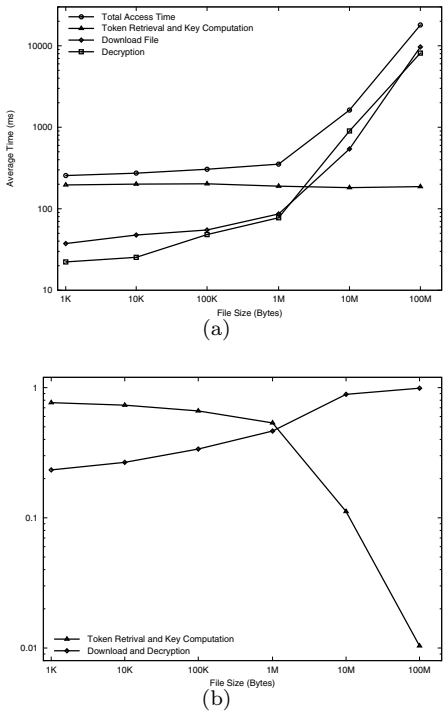


Figure 7. Ratio between the token retrieval–key computation time and the total access time (a), and between the download–decrypt time and the total access time (b)

Figure 7(a) shows the average response time in accessing the outsourced documents as a function of the data size. The data are depicted using a logarithmic scale on both axes. Experiments show that the technique is able to quickly produce resources asked by the users, demonstrating how the management of the encryption policy graph has a limited impact on the performance of the data retrieval operation. The resource transfer time and the resource decryption time shown in Figure 7(a) are close because the specific constraints of the Web browser platform force to operate the decryption only after the downloaded resource has been written to disk, having disk access rates as a bottleneck in both cases. The overhead due to the retrieval of the tokens and computation of the key from the service is, as expected, constant (200 ms) and appears negligible with respect to the data transfer and decryption operations for documents with size greater than 1MB. Figure 7(b) shows the ratio between the time for token retrieval and key computation operations and the total access time, and the ratio between the time for download and decryption operations and the total access time. From the figure, it is immediate to see that the token retrieval and key computation operations require at most 10% of the total

access time for large documents. Moving to a large network scenario, with the significant decrease in network bandwidth, it is expected that key computation time will increase less than resource transfer time, further reducing the overhead of the system.

VII. Related work

Previous work is related to proposals devoted to the secure handling of database encryption in distributed, Web-based scenarios, where data management is outsourced to external services (e.g., [7]). However, most of these proposals only address the problem of efficiently executing queries directly on the encrypted data, while only a few works are focused on access control enforcement on outsourced data [4], [5], [8]. In [8] the authors present an approach for regulating access to XML documents by using different cryptographic keys over different portions of the XML tree and by introducing special metadata nodes in the structure. In [5], the authors present a selective encryption strategy for enforcing the authorization policy specified by the single owner of the data and to delegate to the external server the management of policy changes. In [4], the authors address the problem of preserving the confidentiality of access control policies. All these proposals fall short in the cloud storage scenario, since they are based on the assumption that all the data are owned by a single party. Our model enforces a complete resource sharing solution, guaranteeing data confidentiality with respect to the service provider. Also, it can easily support resource and authorization updates and policy confidentiality, leveraging on the models presented in [4], [5]; our prototype already includes these functionalities. A different, but related, line of work is represented by the proposals addressing the problem of preserving data integrity, both in the data outsourcing (e.g., [9]) and, more recently, in the cloud storage scenarios (e.g., [3]). These proposals however do not address the problem of selectively sharing resources in a multi-owner environment.

An additional solution for enforcing an authorization policy via encryption may exploit a PGP-like approach. A resource r may be encrypted with an arbitrary key k and then the encrypted resource may be extended with a descriptor that contains as many copies of k as the number of users in $acl(r)$, where each copy of k is encrypted using the public key of an authorized user. This solution however requires a large descriptor and does not exploit the fact that different resources might have the same acl.

VIII. Conclusions

We propose an approach for enabling users to regulate access on resources they wish to share in a selective way with other users in a community. Our approach exploits encryption to attach the access control restrictions to the resources and relies on key agreement and key derivation techniques to ensure manageability and scalability of key management. The result is an efficient selective encryption and dissemination approach that responds to the needs of the user community whose attention on privacy and selective dissemination of information has been considerably increasing in the last few years.

Acknowledgements

This work was supported in part by the EU within the 7FP project "PrimeLife" under grant agreement 216483 and by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4). The work of Sushil Jajodia was partially supported by the National Science Foundation under grants CT-20013A, CT-0716567, CT-0716323, CT-0627493; by the Air Force Office of Scientific Research under grants FA9550-07-1-0527, FA9550-09-1-0421, and FA9550-08-1-0157; and by the Army Research Office DURIP award W911NF-09-01-0352.

References

[1] M. Atallah, K. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *Proc. CCS'05*, Alexandria, USA, Nov. 2005.

[2] G. Ateniese, A. De Santis, A.L. Ferrara, and B. Masucci. Provably-secure time-bound hierarchical key assignment schemes. In *Proc. CCS'06*, Alexandria, USA, Oct. 2006.

[3] K. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proc. CCS'09*, Chicago, USA, Nov. 2009.

[4] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati. Preserving confidentiality of security policies in data outsourcing. In *Proc. WPES'08*, Alexandria, USA, Oct. 2008.

[5] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption Policies for Regulating Access to Outsourced Data. *ACM TODS*, 2010 (to appear).

[6] R. Dingledine, N. Mathewson, and P. Syverson. TOR: the second-generation onion router. In *Proc. USENIX Security'04*, Berkeley, USA, Aug. 2004.

[7] H. Hacigümüs, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. ICDE'02*, San Jose, USA, Feb. 2002.

[8] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *Proc. VLDB'03*, Berlin, Germany, Sept. 2003.

[9] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM TOS*, 2(2):107–138, May 2006.

[10] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *Proc. SCIS'00*, Okinawa, Japan, Jan. 2000.

[11] P. Zimmermann. *The official PGP user's guide*. MIT Press, 1995.