Scalable Computing: Practice and Experience
Volume 12, Number 3, pp. 317–336. http://www.scpe.org
ISSN 1895-1767
© 2011 SCPE

SECURE ACCESS MECHANISM FOR CLOUD STORAGE

DANNY HARNIK, ELLIOT K. KOLODNER, SHAHAR RONEN, JULIAN SATRAN, ALEXANDRA SHULMAN-PELEG, AND SIVAN TAL *

Abstract. Emerging storage cloud systems provide continuously available and highly scalable storage services to millions of geographically distributed clients. A secure access control mechanism is a crucial prerequisite for allowing clients to entrust their data to such cloud services. The seamlessly unlimited scale of the cloud and the new usage scenarios that accompany it pose new challenges in the design of such access control systems.

In this paper we present a capability-based access control model and architecture appropriate for cloud storage systems that is secure, flexible, and scalable. We introduce new functionalities such as a flexible and dynamic description of resources; an advanced delegation mechanism and support for auditability, accountability and access confinement. The paper details the secure access model, shows how it fits in a scalable storage cloud architecture, and analyzes its security and performance.

1. Introduction.

The rapid growth in the amount of personal and organizational digital data is a major characteristic of information systems in this decade. This growth is accompanied by increased demand for availability, as users wish to run their software and access their data, regardless of their type and location, from any web-enabled device at any time. Cloud computing addresses these challenges by providing virtualized resources as an online service and by allowing them to scale dynamically. Services offered over the cloud include applications (Software as a Service), virtual machines (Infrastructure as a Service), and data storage (Storage as a Service). The latter service, along with the storage cloud upon which it resides, is the topic of this paper.

Storage cloud systems are required to provide continuous availability and elastic scalability [4] while serving *multiple tenants* accessing their data *through the Internet*. The storage cloud infrastructures are comprised of tens of geographically dispersed data centers (DCs), where each DC is comprised of many thousands of compute and storage nodes, and should be able to efficiently serve millions of clients sharing billions of objects. Furthermore, to achieve availability and scalability, each data object is replicated at multiple data centers across the cloud. Each object replica should be accessible for reads and writes, and to optimize access latency and data throughput, it is preferable that clients be directed to the replica closest to them.

1.1. Security and access control in the storage cloud.

Data security risks are a key barrier to the acceptance of cloud storage [25, 30, 43]. Many potential users are rightfully worried about moving their data to a storage cloud, where it will be stored by an external provider on an infrastructure also used for storing other customer's data. Storage cloud providers must therefore implement a secure access control system in order to reduce the risk of unauthorized access to a reasonably low level. *Access control* refers to managing identities, defining access policies, and controlling access to resources in a secure manner.

Security has its costs, as the structure of very large scale storage systems incurs a trade-off between performance, availability and security [23]. Balancing this trade-off is particularly challenging in the storage cloud environment, as providers must implement a secure access control system that scales elastically and meets the high availability requirements of the cloud as described above. Moreover, even though the consistency of the data itself can be reduced to improve availability [41], the system must always consistently enforce access control at all access points for all data objects.

In addition to the scale and availability requirements, today's new web applications such as mashups, introduce new data access practices. Namely, data is not necessarily accessed directly by its original owner but rather through various applications, in flexible sharing scenarios and with various billing methods. These applications put forth new functional requirements for the access control mechanisms. These include, for example, the ability of a client to delegate a flexible subset of his access rights to anyone he or she wishes and support for the related notion of *discretionary access control (DAC)* [1]. Also required is support for hierarchical management of rights, assigning administrators privileged access to a data domain and allowing them to delegate partial access to other principals under their control.

Traditional file-based storage systems employ a monolithic access control system. In such a system, a client is authenticated upon logging in to a session, and is then authorized by the data server for each file access, according to the access policies (e.g., *ACLs*) associated with the accessed file. This approach is insufficient for a cloud storage system for several reasons. First, HTTP is a stateless protocol so session-based security is

*IBM Haifa Research Lab, Haifa, Israel 31905, Email: {dannyh,kolodner,shaharr,julian_satran,shulmana,sivant}@il.ibm.com

not always applicable; even with secure HTTP sessions, it is inefficient and sometimes impractical to associate a session with each client. Second, performing identification, authentication, and authorization for each data access request creates a performance overhead especially in very large scale systems with high multi-tenancy. Furthermore, some of the new requirements, such as replication in an eventually-consistent environment, make the update of access rights more difficult. We advocate, instead, a capability-based approach, already used in distributed file systems (DFSs) such as Ceph [42], which seems more suitable for cloud-scale storage.

1.2. Our contributions. In this paper we propose a capability-based access control model designed to address the emerging challenges of storage clouds. Extending previous capability-based access control models [39, 11, 31, 23, 12] we introduce the following innovations:

- We present a model supporting fine grain and dynamic scope for access rights. We introduce a dynamic definition of the resources via a flexible description with pattern matching on object attributes (e.g. object name, type, or metadata). Thus, the accessible subset of resources can change as objects are added or removed from the system.
- We enable user-to-user and user-to-application delegation, where one user can directly delegate a subset of his access rights to a subset of resources without requiring the intervention of a system administrator or the storage cloud. We introduce mechanisms for auditing and access confinement that can serve as a basis for determining accountability, compliance and billing.
- We present an architecture of a data center that supports capability-based access control. It separates authentication and authorization from the data access path, allowing non-mediated and efficient access to data without the need to specify the accessed storage server (i.e. the same credential is valid for all the replicas regardless of their physical location).
- We enable integration with external identity or access management components used by enterprises and applications. This allows combining the advantages of capabilities with those of other access control models (e.g. ACLs). This enables fulfilling the following requirements that no single access control mechanism can satisfy alone (1) Compatibility with existing systems and APIs; (2) Centralized monitoring of the access rights granted to all users over specific resources; (3) Achieving the advantages of capability-based models (see Section 6).

We show that our protocol has the same security properties as other capability-based models and does not introduce new threats. We believe that the benefits of our access control mechanism, both in scalability and in functionality, remove obstacles preventing potential clients from adopting cloud storage.

1.3. Paper organization. Section 2 details the requirements from a storage cloud access control system, and provides an overview of state-of-the-art storage cloud systems, highlighting their current limitations. Section 3 describes our protocol, elaborating on the new features. Section 4 evaluates the security of our proposed method. Section 5 describes an overall architecture for a storage cloud system incorporating our protocol. Section 6 includes a discussion and evaluation of our system in light of the requirements set forth in Section 2.1. We conclude and discuss future work in Section 7.

2. Background. In this paper we introduce a solution for new requirements regarding secure access for cloud storage. We begin by describing the access control requirements for a secure cloud-scale storage system and the motivation for them. We then analyze how existing systems satisfy these requirements.

2.1. Access control requirements. The challenge of access control is to selectively control who can access and manipulate information and allow only users with the proper privilege to carry out operations. Here we focus on functional properties of access control specific to storage in a cloud environment.

Secure chaining of services. When a client uses an application or service on the cloud, this application often relies on another application to perform an action, which in turn may invoke yet another application and so on. When chaining services in this way, it is important to keep track of the access rights of the client at the end of the chain; otherwise, the system could become vulnerable to attacks like clickjacking and cross-site request forgery that can occur when accessing the web [7]. Such attacks can be prevented by giving each client the least authority [40] required to perform its request, and correctly maintaining the access permissions of the client along the chain, i.e., taking care that they are neither expanded nor restricted.

User-to-user access delegation. Cloud systems aim to provide their users with the ability to easily share resources on a very large scale. To allow a cloud system to scale arbitrarily, the role of system administrators needs to be minimal. In particular, a user should be able to delegate access rights to other users without

requiring the intervention of a system administrator. In addition, a user should be able to delegate access rights (or a subset of them) to his resources (or a subset of them) to whomever they choose, including users from other administrative domains, such as users registered to other cloud service provider, or not registered to cloud services at all. Cross-provider delegation makes it easier to spread one's data or services across multiple cloud providers. While this is a recommended practice for increasing availability and resistance to DDOS attacks, it is hard to implement using the current access control mechanisms [4]. Furthermore, a user who is granted access rights from the administrator or from another user should be able to further delegate the rights to other users; this is known as "transitive delegation" [29]. Additionally, to allow flexibility in delegation, the set of delegated resources should be defined using predefined criteria (e.g., regular expressions), in order to allow the set of delegated resources to change dynamically as the contents change (see Section 3.1.1 on page 321). Finally, in order to provide a secure environment that allows auditing and compliance to regulations, it should always be possible to determine who had access to a specific resource and who is responsible for the delegation.

Scalability and high availability. Cloud services are expected to always be available, even during failures and disasters, and to adjust themselves to significant load fluctuations within short periods of time. Distribution of resources and data is a key instrument in meeting these goals, and should be coupled with a corresponding distribution of the access control system. For example, a central authentication point or authorization point would not scale well: it could create bottlenecks, be a single point of failure and become an attractive attack target (e.g. [34]). Furthermore, any point in the cloud possessing a resource should be able to serve the request while enforcing access control. We also add the requirement that the access control mechanism must not affect the data access latency substantially.

2.2. Review of the state of the art. We now proceed to examine how the existing solutions satisfy these requirements. Distributed File Systems (DFS) have been used for file sharing within organizations for many years. However, as production DFS protocols (e.g., NFS, AFS, CIFS) were not designed for file sharing across organizational boundaries, they fail to meet the user-to-user delegation requirements we set above [29, pp. 25-26]. While some experimental DFSs (e.g., Fileteller [19] and DisCFS [28]) and file-sharing services (WebDAVA [24]) meet some of the delegation requirements, they are based on the Public Key Infrastructure (PKI), which has been shown to have multiple limitations for adoption in cloud-scale systems [34]. On the other hand, efforts like OAuth [16] which do not assume PKI, require that access delegation grants pass through the service provider, thus not meeting our user-to-user delegation requirement, and also imposing a communication overhead.

Some existing capability-based models, for example, OSD [11, 31, 23], SCARED [39], and CbCS [12], satisfy the requirements of secure chaining of services and user-to-user delegation. However, these protocols are not geared toward the scalability and high availability required by the cloud environment, as they were designed for storage systems with data models (e.g. device/object) that are not adequate for the cloud.

Production storage solutions designed for cloud environments have difficulties in achieving the described requirements. The major players in the market [3, 10, 26, 33, 37, 35] use *Access Control Lists (ACLs)* for authorization and do the authorization stage in the data access path. This approach has limitations both in its ability to easily scale and in its ability to chain services without losing track of the privileges of the principals, thus failing to enforce the principle of least privilege (as described in [7]). In addition, none of the services we examined enable a secure user-to-user delegation, with the ability to delegate a subset of access rights using predefined criteria.

Following is a brief survey of the most significant cloud storage solutions with respect to their secure access approach.

Amazon S3 [3, 2]. Amazon S3 has a simple method for granting access to other parties using ACLs, but the other parties must be registered S3 users or groups. Objects can be made public by granting access to the *Anonymous Group* but there is no way to selectively grant access to principals outside of the S3 domain. An additional limitation is that the S3 ACLs are limited to listing 100 principals. Authentication is based on user identifiers (unique within S3) and secret keys shared between the users and the S3 system. The requester includes his identifier in the request and signs selected parts of the request message with a keyed hash (HMAC) using his secret key. Since the authentication is embedded in the HTTP request, an authorized user can create an authenticated request and pass it to another user to get an access to a specific object. Furthermore, since the authentication information can be passed in a query string as part of the URL of the HTTP request, this method, known as *Query String Authentication (QSA)*, can be used by web applications interacting with web browsers to provide seamless access delegation.

EMC Atmos Online [9]. Atmos has an authentication method similar to Amazon S3, and the same limitations apply. The identity and authorization system is a little different. Atmos has a hierarchy of organizations (called *subtenants*) and users within them. ACLs are maintained within subtenants with no option to grant access permission to any principal outside the subtenant to which the object's owner belong. User groups are not supported, and objects cannot be made public for anonymous access.

Microsoft Windows Azure Storage [26, 27]. Azure has an authentication method similar to Amazon S3 and EMC Atmos. Like S3, Azure allows defining objects as public to allow anonymous read access. Like Atmos, the ACL support is minimal - there is no group support; moreover, the access policy is either public (anyone) or private (only the owner). On top of that, Azure supports limited capability-based access. This is done using a *Shared Access Signature*, which encodes a capability to perform specified operations on a specified object (container or blob) for a specified period of time. The Shared Access Signature is signed with the owner's secret key. It can then be passed to any user and used in a URL's query string to gain access.

Following is comparison of the cloud storage solutions with the requirements outlined earlier.

Secure chaining. One thing in common to all the systems described above, is that every data access request is authenticated using an authentication string based on a secret key that the user shares with the storage system. Existing systems do not provide a built-in mechanism that allows one service to chain the services of another, and therefore the solution should be implemented in one of the following ways: (1) Obtaining the secret keys or tokens that will identify their clients to the storage servers; or (2) Moving the control of the data from the end user to the service provider, who will access the data on his behalf. Both options are limited and inconvenient.

Access delegation. The basic delegation mechanism of most systems requires changing the ACL configuration at the storage servers and can not allow controlled access to users who are unknown to the system. Among registered users, transitive delegation is possible only by granting write permissions to the delegated resource's ACL. This has multiple security risks and delegates more rights than actually needed, thus violating the principle of least privilege. Furthermore, there is no mechanism for controlled transitive delegation among users that are unknown to the system. In addition, existing services limit the definition of resources for delegation to fixed criteria (e.g., names of specific objects or buckets/containers). They do not support dynamic criteria (e.g., all objects created after a certain date), which are important when handling billions of objects and million of users, as is common in cloud environments.

Scalability and availability. With the exception of Azure's Shared Access Signature, the authentication string authenticates the identity of the user, and authorization has to be done by the storage system when serving each request. That means that authorization is done on the data access path, which impacts the scalability and availability of the system. Furthermore, all these systems (including Azure) authenticate the client using its shared secret key for each access request, which adds more limitations on the scale and availability. The design and distribution of the authentication and the authorization services need to be tightly coupled to the distribution of the data objects.

In summary, today's cloud solutions fall short of satisfying the requirements we define above. We proceed to present a capability-based model that addresses these requirements without compromising the system's security, scalability, or availability.

3. Secure Access Protocol. In this section we present a capability-based access control mechanism for secure access to objects in a storage cloud. Extending the OSD security model [11] and enhancing the delegation technique of Reed et al. [39], our protocol satisfies the cloud storage requirements introducing the following key functionalities: a *dynamic capability scope*, *advanced delegation*, *auditing and access confinement mechanisms*.

We assume the simple and most common cloud storage data model [3, 10], which is comprised of the following types of resources: **Namespaces**, which are abstract containers providing context (such as ownership) for the data objects; and **Data objects** of arbitrary size[1], containing arbitrary content type. Every object belongs to a single namespace and has a unique identifier within that namespace. Replication degree and replica placement are determined at the namespace level. Nevertheless, the access control protocol described here is independent of the specific data model and does not depend on these specific architectural decisions.

Capability-based access protocols involve three entities that are illustrated in Figure 3.1: *clients*; a *security/policy manager*, which authenticates and authorizes the clients and grants the credentials; and *storage servers* that enforce access control by validating the credentials. The access flow in such systems consists of

[1]Typically an implementation has a limit on object size, but this is not our concern here.

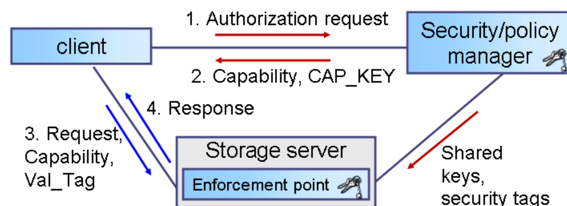


Fig. 3.1: **The OSD [31] security protocol.** The client authenticates with the security manager and receives a *credential* comprised of a *capability* and a *capability-key* (CAP_KEY). It then sends its request together with the received capability and the validation tag (Val_Tag) to the storage server, that validates the request and then performs it.

two stages: (1) the client requests a credential from the security manager and then (2) uses it for accessing the data on the storage servers. The *enforcement point* is the component in a storage server that enforces access control.

Below we describe the capability structure and basic protocol, present the cloud-related enhancements in detail, and then illustrate them with specific examples.

3.1. Capabilities. A *capability*, in the context of this work, is a data structure that encodes certain access rights on one or more identified resources. The full details of the capability fields used in our system are provided in the Appendix. Table 3.1 presents the capability arguments that enable the new functionalities that we describe in the sections below.

3.1.1. Flexible and dynamic capability scope. In our model, the capabilities can provide a flexible description of resources based on attributes such as object name, type and metadata. To allow this, the $ResourceDescriptor$ component in the capability can contain regular expressions and other selection criteria. This enables efficient data access management by allowing creation of capabilities with dynamic scope describing resources that can change over time (e.g. as new objects matching the selection criteria are added). An interesting example is for Information Lifecycle Management (ILM): a "housekeeping" application may receive a capability to move, delete or compress objects that are older than a specified age, or have not been accessed for a specified amount of time.

3.1.2. Flexible access rights. The $Permissions$ field in the capability can specify any type of operation which is properly defined to the security manager and the enforcement point. For example, in addition to standard operations such as $Create$, $Read$, $Update$, $Delete$, or even $UpdateMetadata$, it can specify operations exclusive to specific content types such as $Compress$, $Resize$, $RotateFlip$ or $ReduceResolution$. Such operations are introduced in some storage cloud offerings, for example, Nirvanix [32] already added methods like $RotateFlip$ to their API. Using our proposed protocol, the granularity of the access policy can match the functionality supported by the storage service.

3.2. Credentials and access control flow. Below we describe the two stages of access: (1) obtaining credentials and (2) accessing the resource.

1. *Obtaining the credentials.* A client sends a request for a capability to a security manager, which in turn validates the identity of the requestor and performs an authorization process. If the request is approved, the security manager responds with the credential comprised of the *capability* and the *capability-key*. The capability (CAP) denotes the public part of the credential specifying one or more resources and the granted access rights. The capability-key (CAP_KEY) is the cryptographic hardening of a capability with a secret key and pseudorandom function (PRF), which can be a keyed cryptographic hash such as HMAC-SHA1 or HMAC-SHA256. The key used for this operation, $KeyNS$, is shared by the security manager and the storage server, for the addressed namespace. Thus, $PRF_{KeyNS}(CAP)$ denotes the pseudo-random function calculated with the key $KeyNS$ over the fields defined by the capability CAP. Since the capability-key is the secret part of the credential it should be sent to the client in encrypted form, either as part of the protocol or by using an encrypted communication channel.
$Credential = [CAP, CAP_KEY]$
$CAP_KEY = PRF_{KeyNS}(CAP)$

Field	Description
Resource – Descriptor	Resource(s) to which the capability applies (see Section 3.1.1).
Permissions	Operations allowed on the specified resource (see Section 3.1.2).
Discriminator	A nonce ensuring the uniqueness of capabilities (see Appendix).
Identity	Optional field (see Section 3.5).
Audit	Optional field (see Section 3.6).
SecurityInfo	The security method and protocol control information (see Section 3.2).
Delegatability	A boolean value allowing/disallowing delegation.
PolicyAccessTag	Used for revocation purposes (see [11]).

Table 3.1: **Capability Structure.**

Security Method	*Val_Tag* calculation	Setting	Benefits
CHID	Channel identifier	Secure channel (not necessarily encrypted), e.g. IPSEC, HTTPS.	Protects against replaying the same message outside the specific channel.
MSGH	Message fields (HTTP method, URI, Date, Content-Type and Content-MD5)	HTTP requests over an open channel.	Authenticates the message. Prevents modification and replay attacks.

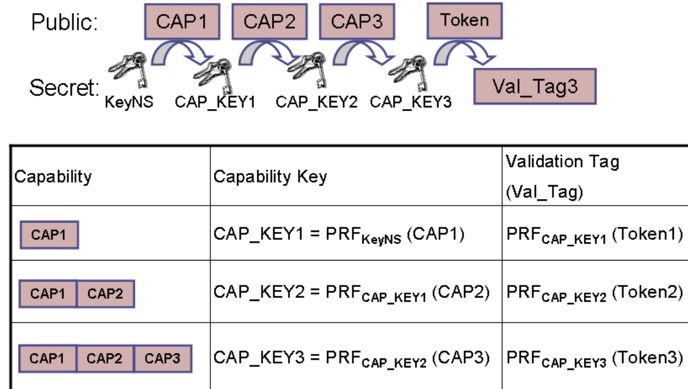
Table 3.2: **Security methods.** Summary of the security methods intended for secure and insecure channels.

2. *Accessing the resource.* To access the object of interest the client attaches to the *Request* (typically a combination of a method and data, e.g., *write* and the contents of the object to be written), a credential consisting of two parts: the capability and the validation tag (Val_Tag). The validation tag is a keyed hash that is computed with the client's capability-key; it is used to authenticate the capability and additional information as defined by the security method described below. The parameter used for the *token* depends on the security method that is encoded in the capability's $SecurityInfo$ field.
$Message = [Request, CAP, Val_Tag]$
$Val_Tag = PRF_{CAP_KEY}(token)$
The choice of the security method depends on the guarantees of the underlying communication channel. We consider the following two security methods to be the most suitable for a storage cloud environment– one for a secure channel and the other for an insecure channel:
Channel identifier (CHID) This method is similar to the CAPKEY method of the OSD protocol [31]. It is suitable for use over a protected communication channel such as IPSEC or HTTPS. In this case the $Token$ is taken to be the unique channel identifier of the protected communication channel. The channel identifier should be unique to the combination of client, server, and the particular link on which they communicate, and should be known by both end points.
Message headers (MSGH). Suitable for access via an open HTTP protocol. In this case the *token* contains some of the HTTP message fields that are significant in terms of access rights. These include the standard HTTP headers such as the HTTP method and resource URI, as well as Host, Date, Content-Type and Content-MD5. As explained in more detail in Section 4, this method provides additional security in an insecure network environment at the cost of having to re-compute the Val_Tag for every command. In conjunction with the Content-MD5 field, it authenticates the data in addition to the request and capability.

Table 3.2 summarizes the two security methods and their typical use and guarantees, which are discussed further in Section 4.

When a request message is received at the enforcement point in the storage server, it is validated by performing the following steps:

1. *Calculating the capability-key.* The capability-key is computed based on the namespace key shared with



Capability	Capability Key	Validation Tag (Val_Tag)
CAP1	CAP_KEY1 = PRF_{KeyNS} (CAP1)	PRF_{CAP_KEY1} (Token1)
CAP1 CAP2	CAP_KEY2 = PRF_{CAP_KEY1} (CAP2)	PRF_{CAP_KEY2} (Token2)
CAP1 CAP2 CAP3	CAP_KEY3 = PRF_{CAP_KEY2} (CAP3)	PRF_{CAP_KEY3} (Token3)

Fig. 3.2: **Delegation chaining**. Access delegation among 3 different users. The security manager uses the namespace key, $KeyNS$, to generate a capability, $CAP1$, and a capability key CAP_KEY_1 for the first user. Using this key the user can create validation tags for his requests as well as generate the credentials for the second user, who in turn can generate the credentials for the third.

the security manager and the received capability.

2. *Calculating the expected validation tag.* The capability-key computed in the previous step is used to compute the expected validation tag, according to the security method specified in the capability.
3. *Comparing the validation tags.* The received validation tag is compared to the expected validation tag.
4. *Validating the capability and request.* The received capability is validated to be formatted correctly and to match the requested operation. After the credential is validated, other required conditions and request parameters may be checked as done normally (e.g., timestamp validation, Content-MD5 validation and so on).

3.3. User-to-user delegation. In this section we describe an advanced delegation mechanism that enables a user to create (and pass to another user) a new capability that encodes a subset of the rights on a subset of the resources encoded in an existing capability. It should be noted that delegation is an inherent feature of capability-based systems. Since the credential encodes a capability rather than an identity, it can be delegated to any principal. However, it is often desirable for a principal to delegate a subset of his rights to another principal. For example, a user that has full control over a namespace may want to allow another user to access a subset of the namespace's objects, perhaps for *read only* access. We follow a technique presented by Reed et al. [39] and adapt it to the storage cloud environment. Below we detail how the delegated credential is built and validated. Section 3.4 further illustrates these mechanisms with specific examples.

3.3.1. Delegation mechanism. During delegation a user creates and passes to another user a new capability, that is derived from the original capability he received from the security manager. Since an important goal of generating this new capability is the reduction of access rights, we term it a *reduced capability.* Having a credential $[CAP, CAP_KEY]$, a *reduced capability credential* can be created by appending a reduced capability to the original capability, and creating a new capability-key by hashing the appended part using the original capability-key as the key.

Formally: A delegation chain is created such that $Credential_i = [CAP_i, CAP_KEY_i]$ is defined as follows:

$$
\begin{aligned}
&Credential_1 = [CAP_1, PRF_{KeyNS}(CAP_1)] \\
&For\ n > 1: \\
&\quad CAP_KEY_n = PRF_{CAP_KEY_{n-1}}(CAP_n) \\
&\quad Credential_n = [(CAP_1, ..., CAP_n), CAP_KEY_n]
\end{aligned}
$$

Notes:

- A Client cannot generate $Credential_1$ because that requires knowledge of $KeyNS$.

- Generating CAP_KEY_n requires knowledge of CAP_KEY_{n-1}. Yet, CAP_KEY_n does not reveal any useful information regarding CAP_KEY_{n-1}.
- It is crucial that all capability keys be sent over authenticated and encrypted channels.
- Our key generation technique computes CAP_KEY_n based only on CAP_n as opposed to $(CAP_1, \ldots, CAP_n)$ used by Reed et al. [39]. This reduces the cryptographic overhead, while preserving the same level of security.

This mechanism of creating new capabilities and capability-keys based on existing ones is illustrated in Figure 3.2, which depicts the structure of the credentials in the delegation chain.

3.3.2. Reduction of access rights. When allowing user-to-user delegation it is important to ensure that each user can only reduce his own access rights and cannot delegate access beyond his own permission. We define one capability CAP_{i+1} to be a subset of another capability CAP_i if and only if the value of each field in CAP_{i+1} does not describe a higher privilege than the corresponding field in CAP_i. For example, we require that during delegation the new *ResourceDescriptor* describes a subset of the resources, and *Permissions* describes a subset of the operations, and the *ExpiryTime* field contains a lower or equal expiration time. Two fields are an exception to this rule: (1) We require that the $SecurityInfo$ of the two capabilities contain exactly the same security method; (2) we do not compare the *Discriminator* fields.

3.3.3. Delegation validation. When a chained capability credential is processed at the enforcement point it is validated by the following mechanism, which is an extension of the one described above.

1. *Calculation of the chain of capability keys.* First, CAP_KEY_1 is calculated based on CAP_1 and $KeyNS$. Then, each subsequent CAP_KEY_i is calculated from CAP_KEY_{i-1} and CAP_i as described above (see Figure 3.2).
2. *Calculation of the validation tag.* The last capability-key in the chain is used to calculate the expected validation tag.
3. *Compare the validation tags.* Compare the received vs. the expected tags.
4. *Validation of the capability delegation chain and the request.* Validate that each capability in the chain is a proper subset of the preceding capability as defined above and that the request conforms with the last capability. This ensures one cannot increase the access granted by a given credential.

3.4. Examples. Using the scenario illustrated in Figure 3.3, in this sub-section we describe some specific examples that demonstrate the use of the delegation mechanism.

Let *SP* be a service provider using the storage cloud as its back-end to provide data services to its clients. *SP* creates a namespace *SP1* and receives a capability credential from a security manager that grants it full control over *SP1*, as shown below:

$$CAP_{SP} = \begin{bmatrix} ResourceDescriptor\ of namespace\ SP1, \\ Permissions = full\ control\ , \ldots \end{bmatrix}$$
$$CAP_KEY_{SP} = PRF_{KeySP1}(CAP_{SP})$$
$$Credential_{SP} = [CAP_{SP}, CAP_KEY_{SP}]$$

Delegating access to a subset of resources. Alice is a client of *SP* wishing to have full control over her resource, object *A*, which is stored in the aforementioned namespace *SP1*. *SP* is able to act as a manager of all access rights to *SP1* without any intervention of the storage cloud security manager. In our example, *SP* generates and sends Alice the following credential granting her the desired access rights[2]:

$$CAP_{Alice} = \begin{bmatrix} ResourceDescriptor\ of\ Object\ A \\ in\ namespace\ SP1, \\ Permissions = full\ control\ , \ldots \end{bmatrix}$$
$$CAP_KEY_{Alice} = PRF_{CAP_KEY_{SP}}(CAP_{Alice})$$
$$Credential_{Alice} = \left[(CAP_{SP}, CAP_{Alice}), CAP_KEY_{Alice}\right]$$

When Alice uses this credential to access her resource, the enforcement point validates it in the following way: it calculates the expected validation tag (via computing the capability keys CAP_KEY_{SP} and CAP_KEY_{Alice}), compares it to the tag received, and finally it checks that CAP_{Alice} is a subset CAP_{SP} and that CAP_{Alice} is a valid capability that authorizes the requested operation.

[2]Essentially, the service provider acts as the security manager for his managed namespace

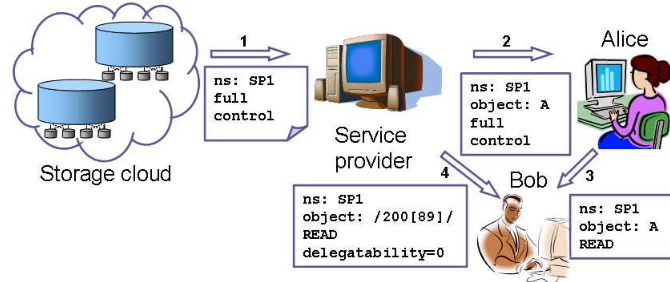


Fig. 3.3: **Delegation chaining**. Examples of user-to-user delegation: (1) The service provider (SP) uses a storage cloud as its back-end. SP receives credentials allowing it full control over its namespace $SP1$. (2) SP grants Alice a credential allowing full control over her object A stored in the storage cloud through SP. (3) Alice grants Bob a credential allowing read-only access to her object A. (4) SP grants Bob full control over objects having the strings "2008" or "2009" in their title.

Delegating a subset of the access rights. Let Bob be a user, unknown to the storage cloud or to the service provider, to whom Alice wishes to grant permission to read her object. Using the same mechanism as above, Alice can delegate a subset of her rights to Bob, creating a new credential with read-only access to her object.

$$CAP_{Bob} = \begin{bmatrix} ...\ ObjDescriptor\ of\ A, \\ Operations = READ\ ... \end{bmatrix}$$
$$CAP_KEY_{Bob} = PRF_{CAP_KEY_{Alice}}(CAP_{Bob})$$
$$Credential_{Bob} = \left[(CAP_{SP1}, CAP_{Alice}, CAP_{Bob}), CAP_KEY_{Bob}\right]$$

When Bob uses this credential to access the resource, the enforcement point validates it as described above, with the added steps due to the additional capability in the chain.

Dynamic capability scope and delegation control. In another scenario, let Bob be an external financial accountant, preparing the reports for years 2008-2009. The service provider should be able to grant Bob access to the subset of objects relevant for these years. For example, these can be described as all objects with object name matching the regular expressions $/200[89]/$ or $/\hat{}report. + 200[89]\$/$ in their name. When new objects matching the pattern are added to the system, this credential automatically provides access to them. In addition, by setting the value of the field *Delegatability* to zero, the service provider may prevent Bob from further delegating the access and generating new credentials. Figure 5.2 presents a sample access request and a credential that can be used in this example.

3.5. Identity Testing and Access Confinement. A key design point in our capability-based access control system is that it relieves the enforcement point from the duty of authenticating the identity of the request sender. Removing this requirement (as well as authorization tasks) from the data access path is instrumental in allowing high scalability and performance of the system (see further discussion in Section 6). However, at times it may be desirable to limit the use of a capability granted to a specific user or group as an extra measure of security, typically for more sensitive data. One consideration is that one may grant a user access to an object, but not trust him to keep his capability credential securely without leaking it. For this purpose, we add an optional *Identity* field in the capability that can be used to confine the credential and make it usable only by the specified identity. It may contain the identity of a principal or a group. The *Identity* is filled in by the delegating person and is verified by the enforcement point. Using this field allows enforcing the *access confinement* property, defined by Lampson [22] (see more in Section 4).

For credentials in which the *Identity* field is used, the enforcement point needs to authenticate the requesting client's identity, and potentially to validate his membership in a group. How the client is authenticated is out of the scope of this protocol. Two points worth mentioning: (1) the authorization is still separated from the enforcement point; and (2) caching of relevant identity information at the enforcement point may help improve the performance in the cases where the *Identity* field is used (as pointed out by Karger [20]).

3.6. Auditability and accountability. In most cases, an access control system is required to provide mechanisms for auditing and accountability of all accesses to a resource. The technical aspects of auditability

and accountability are similar, as both are achieved by generating information that later provides proof of access control decisions and operations. Auditing is of special concern in capability-based access control systems, since a capability-based credential, once obtained, can be used anonymously without having to authenticate the requester's identity (unless the *Identity* field is used, see 3.5). To address this concern, an *Audit* field is added to the capability. For instance, the security manager can store in this field the identity of the client to which the capability was issued. Since the *Audit* field is part of the capability, it is authenticated by the capability-key and cannot be modified. The enforcement point can log this information so it can later be used to learn which capabilities were used and when. This can serve as a tool for *billing* in the cloud, as it may record who is accountable for each access, and is especially useful in implementing today's new pay-per-access policies.

Note that the *Audit* field does not provide identity information about the entity actually accessing the resource. Rather, it provides the identity of the entity that received the credential. The entity that was granted the credential is responsible for securing it and in most cases will be accountable for all usage of the credential. During user-to-user delegation, the delegating client can use the *Audit* field to add further information on the accountability of the delegated credentials. For example, an audit log entry might record that a resource was accessed using a credential that was given to Alice at time T1 and was delegated from Alice to Bob at time T2. In this case, Bob can be held accountable for the access, and Alice can be held accountable for granting the access right to Bob. This method allows for high flexibility in billing schemes as it records the full path of delegation and access. For example, a bill for accessing data can be split between an application providing a service on the data and the end client providing credentials for accessing this data.

4. Security assessment. The access control scheme essentially inherits the security properties of the OSD protocol when translated to the cloud setting. In this section we briefly overview these properties and emphasize the new features in our protocol and their effect on security.

The basic property of an access control mechanism is naturally the assurance that an operation will only take place when it is invoked by a user that is privileged to execute it. In our system privileges are embodied in credentials at varying granularity (such as object, group of objects, read, write, etc.). Security includes denying inadvertent access attempts by users as well as attempts by malicious eavesdroppers monitoring the communication channels. It also includes preventing attacks on the network ranging from simple network errors to attempts at modifying the content of messages, and malicious replaying of messages (replay attacks).

The security of the protocols presented in this paper hinges on two basic properties, both are guarantees of the underlying pseudorandom function (PRF) [14]. The first is the inability of an adversary without knowledge of a namespace key $KeyNS$ to gain any information on a capability key CAP_KEY for a capability CAP of his choice (even when knowing keys for other capabilities). The second is the inability of an adversary that does not possess the capability key CAP_KEY for a capability CAP, to generate a request with a validation tag that corresponds to CAP.

Therefore, a crucial assumption for our setting is that all communication of keys in the protocol is done on a secure encrypted channel. This includes communicating keys to clients of the cloud, and communicating keys within the cloud–between the security manger, the key manager, and the various enforcement points. This also includes client to client communication, such as in the case of delegating capabilities.

Confinement of Corruption. We note that the architecture described in Section 5 attempts to limit the distribution of keys to the necessary minimum. For example, an enforcement point only holds the keys that pertain to the namespaces that it holds. This property ensures that in the case that a server or component are compromised, the damage is confined only to the namespaces that were directly related to this component.

4.1. Security Methods. In the next paragraphs we consider all communication, other than obtaining capabilities and keys from the security manager. We discuss the security for the two possible methods of computing validation tags.

The CHID Method. This method, called CAPKEY in the OSD standard, assumes that clients contact the storage cloud by initiating an anonymous IPSEC channel, an HTTPS connection or an equivalent secure channel. The guarantee of such a channel is that its originators are the only ones able to send messages on it. Namely, an eavesdropper cannot modify messages on the channel, replay messages on it or initiate new messages. The channel may further encrypt all communications, but this is not mandatory for access security. When working in such an environment, basic access control is achieved since an eavesdropper cannot use the messages passed on a different channel because the validation tag binds the messages to the original channel. As pointed out, generating a validation tag for a different channel, message or capability is impossible without

knowledge of the capability key. In addition, the properties of the channel protect against the various network attacks. Altogether, the protocol's use of the channel ID guarantees that no useful information is gained toward manipulation of data in other channels, even though the channel does not have to be encrypted.

The MSGH Method. This setting is very relevant for public clouds, and uses parts of the HTTP message in order to generate the validation tag. In particular, this includes the specifics of the operation, and therefore forms a binding between the credential and a specific command. During write operations, including a content-MD5 field in the generation of the validation tag also guarantees that data being written to an object cannot be modified. Essentially, this offers a guarantee that an eavesdropper intercepting the message can only use it to repeat the exact same command, in the same time frame as specified. While this does constitute a replay attack, it is very limited. There are two key techniques in further limiting replay attacks:

Timestamping The timestamp included in the computation of the validation tag creation, is used at the enforcement point to reject both outdated and future operation requests. This renders messages useless for replay outside of a small time window. However, since clock synchronization is imperfect, the policy employed by the enforcement point is to allow operations that are reasonably close to the timestamp (this clock skew parameter can be modified according to the quality of clock synchronization in the cloud). Note that there is a tradeoff between more lenient and harsher timestamping policies. A harsh policy will suffer more from occasional rejections of legal operations due to message delays and clock skews; resolving this requires regeneration of a new command. A lenient policy increases the time window in which a replay of messages can happen.

Versioning and Nonces While timestamping reduces the possibility of a replay attack significantly, there are techniques for further eliminating such risks. This is particularly relevant in the case of write operation as they may allow some harmful adversarial manipulations. For example, if a delete request for an object is followed closely by a write request, then replaying the delete request will overwrite the last operation. Such mishaps (of out of order write operations) are not restricted only to replay attacks and are also an inherent risk in systems with *eventual consistency.*[3] We suggest to piggyback methods aimed at improving the consistency model (and specifically for ensuring monotonic session write consistency) in order to further eliminate the mentioned replay attacks. For example, by the inclusion of an object version field, write commands can be accepted only for new versions (and deletes for old ones), thus guaranteeing monotonic write consistency. Including this information in the computation of the validation tag ensures that replay attacks will be ruled out altogether since no write or delete command can be repeated twice with the same version number. A similar technique is used in the CMDRSP and ALLDATA modes in the OSD protocol. These methods are similar in spirit to the MSGH method, only MSGH is tailored towards usage in a cloud environment. Specifically, the OSD protocol used a sophisticated combination of nonces and timestamping in order to eliminate replay attacks. However, this technique is less suitable for the setting of a cloud. Note that some operations, e.g., read, are less adapt to such nonce or versioning mechanisms as they do not include inherent replication: one replica is read and the others need not be notified about this.

Altogether, the MSGH method essentially provides authentication on an open and insecure channel, such as a standard HTTP request, and prevents replaying the same request by a malicious party.

4.2. Delegation Security. A central feature in our protocol is the extensive delegation capability. The main concern is the fact that in delegation, capabilities can actually be created by an external principal (external to the cloud) rather than the security manager component. Nevertheless, this additional feature does not compromise the overall security of the protocol. An attack on the system consists of a malicious party performing an operation that it is not allowed to do. For this to happen it must produce a series of capabilities along with a corresponding validation tag for its request. Moreover, this series must deviate from all such other series that it may have obtained legally. There are basically two options for an adversary to present such a series. The first option is to generate a validation tag without knowledge of the end capability key, which as above it is incapable of doing. The other option is that somewhere along the chain of delegation, the adversary obtained a capability key CAP_KEY_i for a capability of its choice without knowledge of the previous key in the chain CAP_KEY_{i-1}. This too is ruled out by the properties of the underlying pseudorandom function [14].

4.3. Limiting Delegation and Access Confinement. Lampson [22] defined the problem of *confinement* (we term this *access confinement*). In essence, it means that not only must no principal be allowed to access

[3]Eventual consistency is a consistency model that seems inevitable in a storage cloud where data is replicated over different geographical regions (see survey by Voegels [41]).

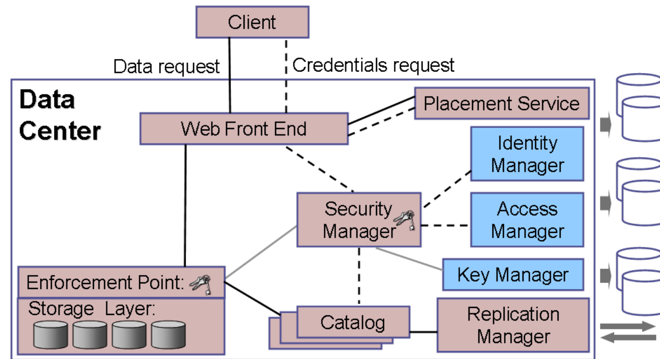


Fig. 5.1: **Secure access architecture.** Dashed lines show the interaction between the components involved in requests for credentials. Solid lines indicate the flow during the data operation request. There are several possible deployment models of these components and some of them may be distributed to several physical machines. The three access management components that are colored blue (Identity, Access and Key Managers) can be deployed at customer premises externally to the data center.

data it was not privileged to, but also that no series of supposedly legal operations by privileged users can end in leakage of information. Karger and Herbert [21] and Boebert [5] showed that the basic capability-based systems are inherently incapable of solving the access confinement problem. The problem stems from the inherent attacks allowed by delegation of credentials. Halevi et al. [15] showed how the OSD protocol can be modified to achieve access confinement, proving this statement by rigorously defining confinement and proving it under the universal composition framework [6]. Due to the similarity of our capability model to that of OSD [11], our protocol achieves the same type of security when invoked with the Identity field in the capability. Recall that when this field is used, the enforcement point is required to validate the identity of the requester and verify that it matches the one in the field. Performing identity checking at the enforcement point achieves the desired access confinement according to the definitions of Halevi et al. [15] (using essentially the same proof).

5. Architecture. We describe an architecture of a storage cloud consisting of multiple data centers (DCs), supporting the presented capability model and satisfying the requirements described Section 2.1. We consider a cloud comprised of a set of geographically dispersed DCs, collectively storing billions of objects and peta-bytes of storage capacity[4]. At this scale, no component that takes part in the data access service can be realized with a single physical entity. Every component has to be distributed and should be able to easily scale. Furthermore, every DC stores and manages only subsets of cloud resources and there is no single component contains all the information about all the objects in the storage cloud.

Figure 5.1 illustrates the main components of our architecture. We start this section with a description of the components required for the general functionality of the DC performing its basic functions such as data serving, replication and placement. Then, we describe the components of the secure access architecture and describe their integration with the rest. Finally, we discuss the scalability issues of our secure access architecture and how we address them.

5.1. General architecture. *The Web Front End* receives client requests and propagates them to the appropriate component inside the data center, or redirects them to the right target DC as needed. It integrates with the web server (httpd) and handles the HTTP protocol related functions. It is designed as a stateless component. It can dynamically scale with load balancing capabilities. According to the type of the received message, which can be either a credential or a data request, the Web Front End communicates with the Security Manager or the Enforcement Point respectively.

The Catalog is a key-value store containing object metadata, both external (such as user metadata) and internal (such as mapping to a storage location). It is accessed by other components and allows efficient lookup of data attributes. For example, supporting the dynamic capability scope, it allows recognizing the objects

[4] Amazon S3 is reported to store more than 100 billion objects as of March 2010 [http://www.datacenterknowledge.com/archives/2010/03/09/amazon-s3-now-hosts-100-billion-objects/]

with the attributes matching the specified selection criteria. It is designed to be distributed across the nodes of the same DC and dynamically scale to handle the large number of objects managed by each DC. It plays an important role in data management and replication across DCs.

The Storage Layer provides persistent storage for objects. It is responsible for allocating space and storing object data. Our prototype implementation uses a clustered file system (within each DC but not spanning DCs) but it can also use underlying block storage directly.

The Placement Service maintains the information regarding the placement of resources in data centers. In accordance to our data model, it maintains the list of data centers that hold a replica for each namespace in the storage cloud. We choose to replicate this information across all the DCs. Each data center has an always consistent list of the replicas it is responsible for; the list of other replicas that are not in its responsibility is eventually consistent.

The Replication Manager is responsible for the replication of object updates across the DCs. The replication mechanism is designed to work asynchronously, providing high availability for write with eventual consistency for read.

5.2. Secure access architecture. Here we describe the main components of the architecture related to access control. Some of the components handle the capability-based access control model described in Figure 3.1, while the others are supporting components that handle the security functions required by the model: authentication, authorization, and key management.

The Enforcement Point processes all data access requests. It enforces access control by validating the client requests and credentials. It is responsible for denying a request that does not include valid credentials authorizing it. When the CHID security method is used, the validated credentials can be cached at the Enforcement Point to improve performance by eliminating the recalculation of capability-keys and validation tags for every request validation.

*The Security Manager*is responsible for handling authorization requests and generating credentials. It orchestrates the authentication and authorization procedures as well as credential revocation, key management and key exchange with the Enforcement Point. Each Security Manager is responsible for a certain subset of resources, which usually reside in the same DC. However, to allow federation of resources, a Security Manager in one data center may serve an authorization request for the data stored in another DC. This is especially useful when two enterprises federate their storage resources without federating the identity or the access management components.

The Identity Manager authentication component responsible for verifying the identity of a principal requesting authorization, as well as group membership inquiries. Under the orchestration of the Security Manager, multiple different Identity Managers can be deployed in the cloud. This allows enterprise customers to continue using their own identity management servers, while other customers, like those providing Web 2.0 and Mashup applications, may rely on the identity management offered by the cloud provider.

The Access Manager responsible for authorization and access policy management, which serves as the basis for the decisions of the security manager that either grants or denies the requests for capabilities. Similarly to identity managers, the cloud architecture allows deployment of multiple external access managers. Each server may use a different model to manage the access control policies (e.g. ACLs, RBAC, etc) allowing compatibility with existing customer infrastructures. This allows the customers to maintain their own access policies, while relying on the cloud security manager as the credential generation authority. However, this is not mandatory and applications may select to maintain internal access managers that will be responsible for the delegation of credentials generated by the cloud provider.

The Key Manager is the key management component responsible for the secure generation, serving and storage of the cryptographic keys shared between the Security Manager and the Enforcement Point. Every namespace has a set of keys associated with it, and efficient key management is provided by storing a copy of the namespace keys in each DC that contains a replica of that namespace.

5.3. Secure access flow. To illustrate the interactions between the components we detail the flow of the two-stage access control model described in Section 3.2.

Obtaining the credentials. Upon receiving a credential request, the Web Front End checks with the Placement Service whether the local DC has a replica of the pertinent namespace (and thus has a replica of the

authorization information for it). If it has not, the message is forwarded to another DC[5]. However, if such a replica does exist in the local DC, the request is passed to the Security Manager, which in turn, communicates with the Identity Manager and the Access Manager components to authenticate the user and authorize his request. Using the key generated by the Key Manager, the Security Manager generates the capability and capability-key, which are returned to the client.

Accessing the resource. When the Web Front End receives a data access request, it also calls the Placement Service to find the replica locations for the pertinent namespace. If it resides in the local DC, the request is passed to the Enforcement Point, that validates the credential and handles the request. It may be required to replicate updates to other data canters, and the Replication Manager is called to perform this task. Since the replication mechanism is out of the scope of this paper, we do not provide details for it.

5.4. Scalability of Access Control. In this sub-section we describe how we address the scalability challenges described in Section 2.1. To ensure availability we distribute the Security Manager across all the DCs. Each Security Manager instance serves only authorization requests pertaining to the namespaces that reside in the respective DC.

Similarly, the Access Manager is distributed across the cloud, with an instance in each DC holding access policy information and serving authorization requests only for namespaces that have a replica in the local DC. This is aligned with the design of the Security Manager. It should be noted, that this distribution of access control is mainly suitable when the Access Manager components are owned by the Cloud provider. When integrating external access managers that are owned and managed by the customers, this distribution may be more difficult due to the trust establishment problems.

The Identity Manager is also distributed across the cloud. However, in contrast to the access policy information, the identity-related information cannot be partitioned as the namespaces are, because users and groups have a global scope and do not pertain to certain namespaces. Therefore, the Identity Manager component includes the ability to retrieve information from (or forwarding requests to) a remote Identity Manager, as well as the ability to cache retrieved information locally for enhanced performance. The ability to integrate with external Identity Providers is important for supporting a federated environment, although this was not part of our prototype implementation.

The Key Manager component generates and stores highly sensitive information, which enables accessing any resource in the cloud. It is divided into two parts: (1) A central Key Manager that generates keys and maintains their backup and lifecycle management. This is implemented as a set of synchronized instances set up in master-slave hierarchy with failover capability; (2) A local key management component in each DC holds encrypted copies of the keys required to serve the namespaces residing in the DC. This local "key vault" component can be reused by both the Security Manager and the Enforcement Point.

It should be noted that a credential generated and provided by the Security Manager in one DC is valid across the storage cloud and can be used to access the pertinent objects regardless of where they are replicated and accessed. The distribution of security and access control functionality is totally transparent, similar to the distribution of the data.

5.5. RESTful Implementation. We developed a prototype of a storage cloud comprised of data centers with the architecture described above. One of our design principles was to support the philosophy of REST (REpresentational State Transfer) [13], which became widely adopted due to its simplicity, scalability and easy deployment [36, 17]. In order to implement the capability-based access control model as part of a RESTful web service, we followed all the fundamental principles of REST, which in our set up can be formulated as follows: (1) All resources, including objects, namespaces and credentials are referenced using a uniform resource identifier (URI); (2) All resources are manipulated by the basic HTTP methods (GET, PUT, DELETE and POST); and (3) All resource requests are stateless.

To support this, our system provides a RESTful API for the communication with the Security Manager, addressing the requested credential as a resource. In essence, an authorization request is sent using an HTTP GET request to a resource of type *credential*. To allow the easy integration of credentials in data access requests, we embed them in the HTTP headers of namespace and data object requests. They are represented as JavaScript Object Notation (JSON) strings, as illustrated in Figure 5.2, which presents an example of a user request to access (GET) an object. It shows the capability and the validation tag that are sent in an HTTP request of a

[5]It may also be redirected to another DC with HTTP redirect.

```
GET SP1/report-March-2009.doc HTTP/1.1
host: www.cloud.acme.com
x-acme-credential:
  [{capability: [
    {ResourceDescriptor :
      [{nsid:SP1},{security tag: 1}]
      [{oid=~/^report.+200[89]$/}]
    {Operations : read, add},
    {Discriminator : 0xFDCED0016EE87953472},
    {Identity :   }, {Audit : Bob},
    {ExpiryTime : "Thu, 31 Jan 2011 17:15:03 GMT},
    {Delegatability: 0 }]
  {Val_Tag: 0xAABF099916EE87953472}]} ]
```

Fig. 5.2: HTTP request to GET an object from the storage cloud. A credential containing a capability and a validation tag are added to the HTTP header. The credential value is represented as a JSON string.

client to the server. Our prototype shows that it is possible to implement the capability-based access control model as a RESTful protocol in which all the resources (including the credentials) are controlled by components that communicate via a standardized HTTP interface and exchange representations of these resources.

6. Discussion. In this section we begin with a general discussion of the architecture of access control systems. Then we revisit the requirements set forth in Section 2.1, and compare the capability-based access control architecture described in Section 5 to alternative access control mechanisms showing how it satisfies the requirements.

6.1. Architectural Overview of Access Control. Every secure access control system has three components:

1. *Authentication*, which authenticates the client entities and reliably assigns them identities.
2. *Authorization*, which makes authorization decisions allowing or denying access by clients to resources.
3. *Enforcement* component, which enforces access control on resources based on information coming from the client, authentication and authorization.

Clearly, enforcement must be done in the data access path. However, authentication and authorization can be done outside the data access path. Access control systems differ on where and when authentication and authorization are done. We identify three main access control architectures and name them accordingly.

Monolithic access control. In this type of access control architecture authentication, authorization and enforcement are all done on the data access path. The client passes authentication information, which the data server uses to identify and authenticate the client, and authorize it to access the requested resource. Although the authentication or the authorization components may execute separately from the enforcement point, they are used by the data server as part of the enforcement in the data access path.

Separation of authentication. In this type of access control architecture, authentication is taken out of the data access path. The authentication component authenticates a client and provides it with a token that vouches for its identity. The token is validated on the data access path without the client needing to provide its authentication information (other than the token). Authorization decisions are still made for each data access request.

Separation of authorization. In this type of access control architecture, both authentication and authorization are separated from the data access path. A client submits a request to access a resource (or a set of resources). After it is authenticated by the authentication component, the authorization component makes the access decision based on its authenticated identity and the access policy for the requested resources. Then the client receives an access token that encodes a *capability* to access the resources. This access token is validated by the data server as part of the enforcement in the data access path.

The capability-based access control model proposed in this paper separates both authentication and authorization from the enforcement point, streamlining the data access path and allowing maximum flexibility and scalability of access control as required in the cloud environment. We explain the benefits of capability-based access control in the following subsection.

6.2. Advantages of capability-based access control.

Chains of services. Only the capability-based access model allows propagating the end user access rights securely through a chain of services all the way down to the accessed resource. When resources are accessed by services on behalf of clients, possibly through a chain of services, capability-based access control enables propagating the end-client access token and enforcing the access control based on the authorization of the end-client. This allows preventing attacks like clickjacking and cross-site request forgery that can occur due to the violation of the principle of least privilege possible in the ACL-based models [7].

When authentication is not separated from the data access path, authentication information has to be propagated through the chain of services all the way down to the enforcement point that calls the authentication component. When authorization is not separated from the data access path, identity information has to be propagated all the way down to the enforcement point that calls the authorization component. Propagating authentication and identity information through the chain of services is cumbersome and reduces the overall security. Furthermore, this information is variable in size and format, since there are numerous standard authentication and authorization methods, each with its own definitions and models. Propagating a capability-based access token allows a unified service protocol, while enabling different authentication and authorization components to be plugged into the system.

User-to-user access delegation. Capability-based access control facilitates access delegation and empowers each client entity to play the authorization role for resources to which it has access. This facilitates several use-cases that are essential to cloud environments. Discretionary access control (DAC) is facilitated by providing every user the ability to delegate his/her access rights to other end users, even when they do not even have a recognized identity in the storage system. (The delegation of access can be limited and controlled as explained in Section 3). Hierarchical authorization structure is facilitated by providing access credentials to domain and sub-domain administrators, allowing them to delegate their access to other entities (subordinate administrator and end-client entities) using authentication and authorization components of their choice and control.

Performance. Capability-based access control systems pay the price of authentication and authorization up-front, and then streamline the data access path by using a capability-based access credential that is propagated from the client entity to the enforcement point. If every access to a resource requires the client to get a credential for the resource, the capability-based system suffers from an overhead that stems from the fact that the client has to perform two operations serially - first get a credential and then access the resource. However, when an access credential is reused in many resource requests, capability-based access control is beneficial and improves the performance. The improvement is particularly significant in large scale environments in which the enforcement point is relieved from accessing remote centralized authentication and authorization servers in the data access path.

When a client sends a request to a data server, a credential is embedded in the request, and the validation of the credential is done locally at the enforcement point without interactions with remote entities. Thus, the overhead is primarily due to the cryptographic calculation required for validation. In particular, the expected Val_Tag is calculated and compared to the one sent by the client (see Section 3.2). The time overhead depends on the pseudorandom function used, the size of the capability, c, the length of the delegation chain d, and the size of data over which the Val_Tag is computed e. Let us denote the time it takes to compute the pseudorandom function over data of length n as $PRF(n)$. Then the time to compute the validation tag at the server side can be described by the following formula:

$$Val_Time(c, d, e) = d \cdot PRF(c) + PRF(e)$$

The time increases with the number of delegations. We considered two implementations with HMAC-SHA1 and HMAC-SHA256 used to calculate both the CAP_KEY and Val_Tag. On an average capability size of 400 bytes, the average CAP_KEY calculation time with these functions was 14μ and 23μ respectively[6]. The calculation of the validation tag depends on the security method and when using the $MSGH$ method the size of the input message headers may be about the same size of a capability. Thus, the overall validation time of a message with delegation chain length d can be approximated by $(d + 1) \cdot PRF(c)$. For example, when using HMAC-SHA256 the validation of delegations of length 4 and 5 will take 115μ and 138μ respectively. That is the time it takes on the server side. On the client side it only takes $PRF(e)$ because the CAP_KEY is already computed when it receives the credential. Since the data access latencies in scalable and distributed storage cloud is usually measured in tens of milliseconds [8], we consider the overhead of the security mechanism to be insignificant.

[6]The measurements were taken on a system with a quad core 2.83 GHz processor

High availability and scalability. Taking authentication and authorization services out of the data access path improves the availability and scalability of the data by eliminating potential points of failure. Intuitively, it may seem that tying authentication and authorization to the enforcement point (as in monolithic access control) improves the availability of the system, as the client interfaces with a single service rather than two different services in order to access the resource. However, this is not really the case when the storage system scales horizontally. In this case the authentication and authorization components need to be (highly) available to the enforcement point at all times. When separating the authentication and authorization from the data access path, these components can more easily be designed for high availability independent of the data service itself. In addition, it should be noted that once a client receives a capability-based credential, it can use it even when the authentication and authorization components are unavailable. The credential can be validated by the enforcement point without requiring the availability of authentication or authorization services.

Our architecture for the data service is distributed across multiple data servers, where each namespace can be served by a set of data centers (typically a small number). From an availability perspective, we designed the components in a way that does not reduce the availability of the data itself. Therefore, we have a security manager component at each DC. But, as with the data itself, not all the authorization information is replicated across all the DCs. Access policy information (i.e., governing authorization), is replicated at granularity of a namespace and aligned with the data placement and replication. Thus, wherever a resource is stored, so is the policy information to authorize its access. In terms of authentication, since the scale is much lower, we replicate all the identity management information across all the DCs. Unlike the access policy data, which changes rapidly as objects are created and deleted, the client information is much more stable and doesn't change as frequently. Therefore, the instance of the security manager available in each DC is capable of authenticating any client and authorizing any access to the namespaces with a replica in that DC.

Revocation. Systems with ACLs allow manual revocation which is achieved by changing the access control configuration. With the recent progress of ACL-based models which allow richer and more complicated description of access rights [29] this mechanism becomes more exposed to human errors. For example, consider a situation where a user changes the ACLs configuration for delegation purposes to grant a one time access to a resource. The granting principle can easily forget to delete this configuration to revoke the unnecessary access rights. Unlike ACL-based systems, the proposed capability-based model has the advantage of supporting three mechanisms for automatic and manual revocation with different level of granularity: (1) Automatic revocation through the "ExpiryTime" field, which determines the time period during which the capability is valid; (2) Manual per resource revocation which is achieved by changing the field "PolicyAccessTag" as described in [11]; (3) Per namespace revocation which is achieved by replacing the keys shared between the security manager and enforcement point.

6.3. Overcoming the limitations of capability-based models. While capability-based access control has multiple advantages it is also important to discuss its limitations. Below, we point out the main barriers in the adoption of capability-based models and describe the solutions that allow to overcome the shortcomings of capabilities while benefiting from their advantages.

Interoperability. Resource-oriented access control model, such as those based on ACLs, are the most common access control mechanism in use and pure capability based systems are implemented mostly in research systems [29, 18]. Thus, pure capability based systems have interoperability problems, which limit their potential adoption. Thus, we propose an architecture that allows the integration of external identity or access management components that are already used by enterprises and applications. These can support ACL-based, RBAC or any other access control model that fulfill the monitoring or compatibility requirements of the enterprise. The architecture that we propose in this paper allows to utilize the access control configurations and decisions made by these servers to grant capabilities that will provide some extra functionality such as the user-to-user delegation of a subset of access rights that is not provided by currently existing servers. We believe that such a hybrid system, which allows to combine the advantages of capabilities with other access control models, has higher chances to be adopted in commercial production systems.

Per-resource auditability. Unlike ACL based systems, the capability models allow an easy tracking of all the resources that a specific user can access. However, they do not allow an easy monitoring of all the users who have an access to a specific resource [18]. To support this type of monitoring a capability-based system should implement an additional layer of tracking the granted access rights. The data center architecture proposed in this paper addresses this requirement by allowing to incorporate external access management servers, which

commonly implement the support the required monitoring capabilities either with the help of policy lists or ACLs.

7. Conclusions and Future Work. We present an access control mechanism, which addresses the new challenges brought forth by the scale and applications introduced in the cloud setting. Observing that most ACL-based systems are limited in their ability to support such features, we developed a capability-based system that addresses them. We present a general architecture of a data center in a storage cloud with integrated security components that addresses the scalability requirements of storage cloud systems. Our architecture allows integration of existing access control solutions toward hybrid architectures that combine the benefits of capability-based models with other commonly used mechanisms such as ACLs or RBAC. We built a prototype implementing each of the components.

In the future we intend to work on the integration of this architecture with existing enterprise systems and commonly used access control standards such as SAML [38]. We hope to use our experimental system to evaluate the overall performance of the access control mechanism on real workloads. We are encouraged by our initial evaluation and are working to incorporate this work in a production storage cloud system. Lastly, we aim to address the challenges raised in federation scenarios, where several enterprises need to federate their resources across geographically distributed administrative domains, while ensuring comprehensive and transparent data interoperability.

8. Acknowledgments. We would like to thank Guy Laden and Eran Rom for their contribution to the development and implementation of the data center architecture presented in this paper. We would also like to thank Dalit Naor for her contribution to the ideas presented, and Aviad Zuck for the prototype implementation performance measurements. The research leading to these results has received funding from the European Community's Seventh FrameworkProgramme (FP7/2007-2013) under grant agreement n 257019.

REFERENCES

[1] Trusted computer system evaluation criteria. Technical Report DoD 5200.28-STD, Department of Defense, December 1985. http://csrc.nist.gov/publications/history/dod85.pdf, accessed Jan 20, 2010.

[2] *Amazon Simple Storage Service Developer Guide (API Version 2006-03-01).* Amazon, a. http://docs.amazonwebservices.com/AmazonS3/2006-03-01/, accessed Jan 12, 2010.

[3] *Amazon Simple Storage Service (Amazon S3).* Amazon, b. http://aws.amazon.com/s3/.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[5] W. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *7th DOD/NBS Computer Security Conference*, pages 291–293, 1984.

[6] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[7] T. Close. ACLs don't. http://www.hpl.hp.com/techreports/2009/HPL-2009-20.pdf.

[8] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM Press, 2007.

[9] *Atmos Online Programmer's Guide.* EMC, a. https://community.emc.com/docs/DOC-3481, accessed Jan 12, 2010.

[10] *EMC Atmos Online Services.* EMC, b. http://www.emccis.com/.

[11] M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The OSD security protocol. In *IEEE Security in Storage Workshop*, pages 29–39, 2005.

[12] M. Factor, D. Naor, E. Rom, J. Satran, and S. Tal. Capability based secure access control to networked storage devices. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 114–128, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3025-7. doi: http://dx.doi.org/10.1109/MSST.2007.6.

[13] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2): 115–150, May 2002. URL http://dx.doi.org/10.1145/514183.514185.

[14] O. Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2000.

[15] S. Halevi, P. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Cryptology ePrint Archive, Report 2005/169, 2005. http://eprint.iacr.org/.

[16] E. Hammer-Lahav. *The OAuth 1.0 Protocol.* Internet Engineering Task Force, February 2010. http://tools.ietf.org/html/draft-hammer-oauth-10.

[17] H. Han, S. Kim, H. Jung, H. Y. Yeom, C. Yoon, J. Park, and Y. Lee. A RESTful approach to the management of cloud infrastructure. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09, pages 139–142, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3840-2. doi: http://dx.doi.org/10.1109/CLOUD.2009.68. URL http://dx.doi.org/10.1109/CLOUD.2009.68.

[18] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn. *Assessment of Access Control Systems*. NIST IR 7316, September 2006. csrc.nist.gov/publications/nistir/7316/NISTIR-7316.pdf.

[19] J. Ioannidis, S. Ioannidis, A. D. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In M. Blaze, editor, *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 2002. ISBN 3-540-00646-X.

[20] P. Karger. Improving security and performance for capability systems, ph.d. dissertation. Technical Report 149, Cambridge, England, 1988.

[21] P. Karger and A. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *IEEE Symposium on Security and Privacy*, pages 2–12, 1984.

[22] B. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[23] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: http://doi.acm.org/10.1145/1362622.1362644.

[24] A. Levine, V. Prevelakis, J. Ioannidis, S. Ioannidis, and A. D. Keromytis. WebDAVA: An administrator-free approach to web file-sharing. In *WETICE*, pages 59–64. IEEE Computer Society, 2003. ISBN 0-7695-1963-6.

[25] E. Messmer. Are security issues delaying adoption of cloud computing? *Network World*, April 2009. http://www.networkworld.com/news/2009/042709-burning-security-cloud-computing.html, accessed Jan 21, 2010.

[26] *Windows Azure Platform*. Microsoft, a. http://www.microsoft.com/windowsazure/windowsazure/.

[27] *Windows Azure Storage Services API Reference*. Microsoft Corp., b. http://msdn.microsoft.com/en-us/library/dd179355.aspx, accessed Jan 17, 2010.

[28] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith. Secure and flexible global file sharing. In *USENIX Annual Technical Conference, FREENIX Track*, pages 165–178. USENIX, 2003. ISBN 1-931971-11-0.

[29] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis. Decentralized access control in distributed file systems. *ACM Comput. Surv.*, 40(3):1–30, 2008. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1380584.1380588.

[30] R. L. Mitchel. Cloud storage triggers security worries. *Computerworld*, July 2009. http://www.computerworld.com/s/article/340438/ Confidence_in_the_Cloud, accessed Jan 21, 2010.

[31] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4-5):401–412, 2008.

[32] *Nirvanix Web Services API Developer's Guide*. Nirvanix, a. http://developer.nirvanix.com/sitefiles/1000/API.html, accessed Jan 13, 2010.

[33] *Nirvanix Storage Delivery Network*. Nirvanix, b. http://www.nirvanix.com/.

[34] Z. Niu, H. Jiang, K. Zhou, T. Yang, and W. Yan. Identification and authentication in large-scale storage systems. *Networking, Architecture, and Storage, International Conference on*, 0:421–427, 2009.

[35] *ParaScale Storage Cloud Supports Virtual File System*. ParaScale. http://www.parascale.com/index.php/products/data-access-security.

[36] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. doi: http://doi.acm.org/10.1145/1367497.1367606. URL `http://doi.acm.org/10.1145/1367497.1367606`.

[37] *The Rackspace Cloud: Cloud Files*. Rackspace. http://www.rackspacecloud.com/cloud_hosting_products/files/.

[38] N. Ragouzis, J. Hughes, R. Philpott, E. Maler, P. Madsen, and T. Scavo. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. OASIS Committee Draft, March 2008. http://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf.

[39] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. Long. Authenticating network-attached storage. *IEEE Micro*, 20:49–57, 2000. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/40.820053.

[40] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *In Proc. IEEE*, 63(9):1278–1308, 1975. http://web.mit.edu/Saltzer/www/publications/protection.

[41] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008. ISSN 1542-7730.

[42] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, pages 307–320. USENIX Association, 2006.

[43] T. Wilson. Security is chief obstacle to cloud computing adoption, study says. *DarkReading: Security*, November 2009. http://www.darkreading.com/securityservices/security/perimeter/ showArticle.jhtml?articleID=221901195, accessed Jan 21, 2010.

9. Appendix. Below are the full details of the capability arguments used in our system.

$$CAP = \begin{bmatrix} ResourceDescriptor,\ Operations, \\ Discriminator,\ Identity,\ Audit, \\ ExpiryTime,\ SecurityInfo,\ Delegatability, \\ DerivedFrom \end{bmatrix}$$

$$ResourceDescriptor = \begin{bmatrix} NamespaceIdentifier,\ ObjectIdentifier \\ ResourceType,\ ResourceCreationTime, \\ SecurityTag \end{bmatrix}$$

- *ResourceDescriptor* - defines the resources to which the capability applies. In our data model, this construct consists of the following fields:
  (1) *NamespaceIdentifier* - a unique identifier that describes the namespace resource; (2) *ObjectIdentifier* - an identifier, uniques within the namespace, that describes the object resource or resources; (3) *ResourceType* - the type of the resource to which the capability applies. For example, this field can be 'namespace', in a capability allowing to add objects to a namespaces. (4) *ResourceCreationTime* - specifies the resource creation time, preventing the situation where a resource is deleted and a another resource with the same name is created; (5) *SecurityTag* - is a security tag, which is used for revocation. In order to revoke the capabilities of a certain resource, the security manager increases the security tag and notifies the enforcement point. The enforcement point checks the security tag and considers all the capabilities with an old tag as invalid.
- *Operations* - describes the operations allowed to the client possessing the credential on the described resource. It can describe any type of operation, which is properly defined to the security manager and the enforcement point. For example, in addition to standard operations such is *read*, *write* and *execute* it can describe new operations like *zip* or *reduce resolution*.
- *Discriminator* - is a unique nonce which allows distinguishing between different capabilities even when they allow similar access rights. To ensure uniqueness even when generated by different security managers, discriminators can consists of such parameters as timestamp and IP address.
- *Identity* - optional field containing the identity of a principle or a group allowed to use the capability. When this field is empty the capability can be used by anyone who posses it (see Section 3.5).
- *Audit* - optional field containing the client or application information required for auditing (see Section 3.6).
- *ExpiryTime* - the capability expiration time.
- *DerivedFrom* - prior capabilities used for delegation chaining (see the example below).
- *Delegatibility* - setting this field to zero or one, prevents or allows the user-to-user delegation respectively.
- *SecurityInfo* - identifies the security method and the calculation of the message validation tag as detailed in the next section.

Edited by: Dana Petcu and Jose Luis Vazquez-Poletti
Received: August 1, 2011
Accepted: August 31, 2011