A Two Layered Approach for Securing an Object Store Network

Alain Azagury* Ran Canetti† Michael Factor* Shai Halevi† Ealan Henis*
Dalit Naor* Noam Rinetzky* Ohad Rodeh* Julian Satran*

Abstract

Storage Area Networks (SAN) are based on direct interaction between clients and storage servers exposing the storage server to network attacks. Giving the client direct access to the storage servers requires verification that the client requests conform with the system protection policy. Today, the only available solutions enforce access control at the granularity of entire storage servers. This is an outcome of the way storage servers abstract storage: an array of fixed size blocks. The alternative approach of providing access control at the granularity of blocks is infeasible - there are too many active blocks in the server. Object stores (r.g., the NASD system [7]) provide means to address these issues. An object store control unit presents an abstraction of a dynamic collection of objects*, each can be seen as a different array of blocks, thus providing the basis for providing protection at an object level.*

In this paper we present a security model for the object store which leverages on existing security infrastructure. We give a simple generic mechanism capable of enforcing an arbitrary access control policy at object granularity. This mechanism is specifically designed to achieve low overhead by minimizing the cost of validating an operation along the critical data path, and lends itself for optimizations such as caching. The key idea of the model is to separate the mechanisms for transport security from the one used for access control and to maximize the use standard security protocols when possible. We utilize a standard industry protocol for authentication, integrity and privacy on the communication channel (IPSec for IP networks) and define a proprietary protocol for authorization on top of the secure communication layer.

Keywords*: SAN, Object Store Device, Storage Security.*

*IBM Haifa Research Lab, Haifa University, Mount Carmel, Haifa 31905, Israel. {azagury, factor, ealan, dalit, noamr, orodeh, satran}@il.ibm.com

†IBM T.J. Watson Research Center {canetti, shaih}@watson.ibm.com

1. Introduction

1.1. Security in Storage Area Networks

Storage area networks (SANs) place the storage servers on the clients network and enable direct access to the storage servers. This design aims at improving I/O performance and system scalability of distributed file systems[1] as it removes the file-server from the critical data path. Furthermore, the file-server is no longer responsible for delivering data; it mainly functions as a meta-data server that manages, among other tasks, the system's access control policy [7, 8, 9].

Letting clients interact directly with the storage servers raises new security concerns: since it is the client who initiates I/O requests, and not the file-server, the storage server can no longer trust that every request received was authorized by the file-server. In addition, placing the storage servers as first class network entities exposes them to similar types of attacks that only the file server faced before: malicious parties forging messages or tampering with message contents, replaying or recording messages, spoofing user's identity or denying service of valid requests. In order to achieve a security level comparable to this of traditional systems, the storage server has to take an active role in the security mechanism of the system. The storage server has to protect the integrity of the data it stores, specifically it should verify that the meta data server (the file server in its new role, sometimes called filemanager) has authorized each request and that no adversary has tampered with any request. Privacy, on the other hand, is optional, as it may be achieved at the application layer (e.g., file-system) [2, 7].

SAN Security essentially does not exist today. The only partial solutions use 'work arounds' such as Zoning and Fencing that are provided by the physical level in Fibre-Channel SANs. At best, such solutions can enforce Logical Unit (LU) access controls, at the granularity of 'all or nothing' for a particular LU.

[1] In this paper, we use only distributed file-systems for examples, but our work equally applies to distributed database systems.

1.2. Object Stores

The Object Store (ObS) [1, 7, 13] is a storage control unit (a storage server) that exposes an "object based" interface. Unlike traditional storage control units that provide interface to unrelated blocks of data, the object store virtually groups together data that is considered to be related by the client[2]. The ObS keeps an internal mapping between an object and its data disk blocks. The client can create (and delete) object, and read or write data from arbitrary offsets in the objects while being completely unaware of the object actual layout on the disk. This abstraction allows to provide protection at the granularity of objects, an intermediate granularity between the "all or nothing LU access", which is too "coarse", and the block level, which is too "fine" (merely the number of blocks in the system prohibits an effective definition of an access/protection policy at the block level.)

An object based storage network is comprised of many clients (e.g., hundreds) and many storage server (e.g., a few dozen). In principal, every client can communicate with every server in the network. In most realistic scenarios, the network also contains a centralized *meta data server*. The "critical data path" is the path between a client and storage server on which most of the communication is done, both in terms of volume and frequency.

1.3. Security Objectives

A viable security mechanism for object store networks has to address the following issues:

- Enforcement of (arbitrary) application-specific access control policies at object granularity[3].
- Protection against network attacks.
- Maintain good performance and resource management, primarily to minimize the performance penalty in the critical path (client-server). Resources include computational resources as well as message bandwidth.
- Simple administration.

1.4. Proposed Solution

Our security model builds on the aforementioned system topology. It assumes three types of functional entities: (i) the *Admin* is an authorization server and its role is to set the system access control policy and authorize client requests; (ii) the *ObS* is a storage server and it is responsible for storing the data and allowing clients to manipulate it (via the object interface); (iii) the *client*, the "consumer" of the stored data. As mentioned before, the system may contain many ObS'es and many clients but only one logical Admin.

In our *trust model*, we assume that the Admin and the ObS'es are trusted entities, but the clients are not. (Specifically, the Admin is trusted to authorize only valid requests, the ObS is trusted not to tamper with the data it stores). The network on which the system operates is not trusted, namely threats like eavesdropping to the transmitted data, actively tampering with the transmitted data, replaying messages and causing a denial of service should be considered. Exact definition of the trust model and threats is given in Section 2.1.

Our security mechanism is *credential based* [7, 14]: to access an object, a client needs to get a suitable credential from the Admin. The Admin decides whether the client is permitted to perform the requested operation (according to the system protection policy), and if so it generates a credential that certifies these rights for the client. To perform an action, the client sends the request and the credential to the ObS. The ObS verifies the validity of the credential, and allows the client to perform (only) the operations that the credential certifies. This scheme is depicted in Figure 1.

The security mechanism is cryptographically hardened against malicious clients: clients can neither forge valid credentials nor modify valid ones to gain unauthorized access. Our solution requires that the underlying transport layer be capable of authenticating (and in some cases encrypting) communication lines. We then implement our authorization protocol on top of the communication layer, taking advantage of the authenticated links. In our implementation, the transport layer is IP and we use IPSec to secure the links. A key idea of our proposed solution is to separate the mechanisms for transport security (which are relatively well understood, accepted and standardized) from the proprietary mechanism used for access control. Separating the two mechanisms provides opportunities to reduce the security performance overhead by utilizing schemes as credential caches and acceleration hardware (e.g., IPSec hardware), opportunities which cannot be used if the two mechanisms are coalesced. In addition the separation enables us to provide a unified model for IP networks and other non-IP networks such as Fibre-Channels. An important observation is that in our solution, the server does not need to authenticate the client. Furthermore, the fact that we rely on an authenticated channel between the server and client is not relevant to the enforcement of access-control; it is only used to ensure message integrity.

We stress that the system that we describe here is *not a complete solution* in and of itself. For example, we do

[2] An object can be thought of as a file while the ObS can be seen as a flat-file system.

[3] We specifically exclude tasks that are related to the access control policy itself (such as the definition of such policy, consistency checking and expressibility).

not handle users authentication, or specify how the access-control policy is managed. Rather, we focus on providing the system with a flexible and powerful, yet simple, mechanism for enforcing any access-control policy.

1.5. Related Work

The problem of protecting a network attached storage system in an untrusted environment has received much attention lately. [12] provides an excellent survey as well as a framework for this topic. It has been identified as a critical factor in the success of such systems in the future. Two objectives are typically sought: authentication and encryption of the data.

The earliest comprehensive discussion of security for storage systems is the Cryptographic File System (CFS) suggested by Matt Blaze [2]. This work is concerned with protecting the data stored on an untrusted server via encryption on the server where the encryption is done by the file system. Other system such as TCFS [5] have further extended this idea. Key management issues become critical in such systems, mainly to allow data sharing; SFS [10] proposes self-certifying pathnames to handle this problem.

Distributed file systems like AFS provide access control to their files through a Kerberos-based system which requires a third trusted party to issue 'tickets'. H. Gobioff [7] in his thesis and the Network Attached Secure Disk architecture (NASD) system [8] base their access control mechanism on basic capability cryptographic primitives, which allow synchronous enforcement of security policy with asynchronous involvement of the server. In this solution, client requests target directly the storage device which can efficiently decide on the validity of the request, without connecting to the file manager on every request. The capability primitive is composed of a private and public credential and uses MAC (Message authentication Code) computation to prove authenticity rather than PKI-based signatures.

Our solution diverges from previously suggested credential-based approaches such as [7] mainly in the fact that it separates the transport layer security from the access control. In contrast, [7] utilize the cryptographic credential granted by Admin to protect the transport layer via a proprietary protocol to ensure message authentication and no-replays, guarantees that are provided already if a secure transport is used. Specifically, an exact comparison of the two solutions shows that our solution can often avoid any cryptographic (MAC) computation for credential validation, whereas in [7] at least one, sometimes two, MAC computations are required. In terms of communication complexity, our solution maintains a 'security window' which requires a handshake (two messages) to establish; however, this window is long-lived so most requests do not require to open a new window and therefore result in a request/reply exchange similarly to [7]. Another difference is that we rely on the transport layer mechanism to provide the basic key management upon which keys for credential generation and validation are generated; in contrast, key management is an integral component of a solution like [7] that is defined 'from scratch'.

The Authenticated network-attached storage [14] provides an architecture which mutually authenticates the network disks and clients. It is based on cryptographic one-way hash functions, mainly for performance reasons, and does not require additional key management schemes besides existing authentication mechanisms within the system. It is mainly concerned with determining the client's access rights, and as such provides a weaker sense of security.

We note that [7, 8], [14] as well as the solution proposed in this paper address the question of how to control access of clients to storage devices in an efficient way, while involving the storage device in a minimal way. They do not address, however, the question of data protection on the disk which seems to be orthogonal.

In [11], a security system for network-attached storage called SNAD is developed which stores and transfers encrypted data, and decrypts it only at the client. Despite this overall security, this system claims to perform within a reasonable overhead.

1.6. Outline of the Paper

Section 2 formally describes the network and trust model and defines the security goal. Section 3 presents our solution for the object store based network security problem, and outlines the proposed protocols; and Section 4 formally proves the security of these protocols within our trust model. In Section 5 we give a high level review of the system design as well as some initial performance measurements. Section 6 concludes.

2. Formal Model

At the heart of the system are two types of entities: many *clients* that contact many *servers* and request access to the data stored on the servers in object form. The main goal is to enforce the access control policy over the stored data, so that only permissible requests are granted. The access control policy is specified and maintained elsewhere and is beyond the scope of this work. We assume that a central entity, denoted by 'Admin', either sets the access control policy or has other means to find out whether a certain request is permissible. We envision the system in an environment which is vulnerable to some types of network attacks, defined below, and therefore need to design mechanisms against such attacks.

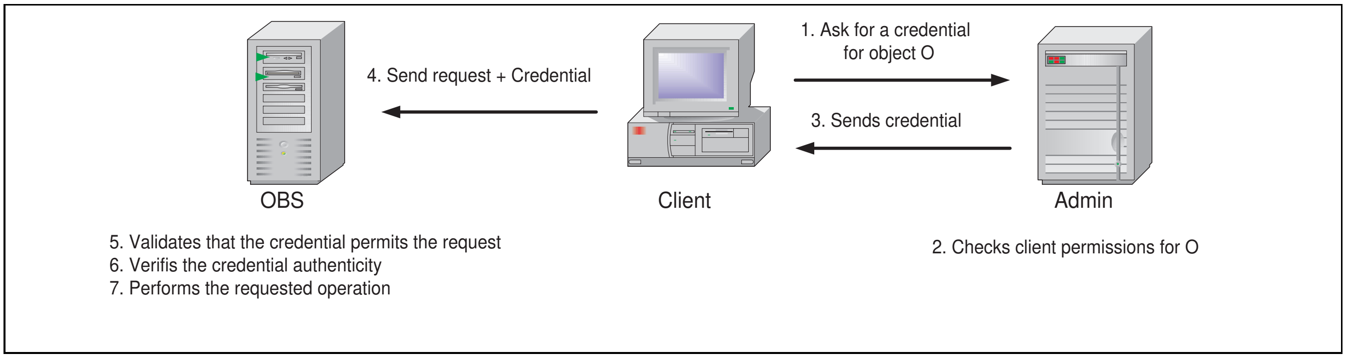

Figure 1. A simplified look of the security model

More formally, let $\mathcal{A}$ be the Admin, $c_1, \ldots, c_n$ be the set of clients and $s_1, \ldots, s_m$ the set of servers, where all these entities communicate over some transport layer, IP in particular. The clients initiate various requests, that are sent to the servers for processing. We assume that the ObS is "session-based" server, in the sense that a client establishes a session with the ObS and sends its requests within the session context. A typical request is $\mathcal{R} = [c_i, s_j, ID_{obj}, op, data(optional)]$ where ID_{obj} is the object ID stored on the j^{th} server and *op* is one of the allowed operations. (In our case, one of { *Create, Delete, Read, Write, Append, Truncate, GetMetaData, Format, GetServerInfo* }.)

The server's response consists of a return code for the operation and possibly some bytes of data. We assume that the Admin can compute, on a given pair of client and object (c_i, ID_{obj}), a vector $Perm(c_i, ID_{obj}) = \overrightarrow{P}$ representing the permissions of client c_i on object ID_{obj}[4].

2.1. Threats and assumptions

We specify our trust model by explicitly stating what an adversary can and cannot do. Roughly, the things that the adversary can do correspond to threats that we need to address, and the things that it cannot do correspond to assumptions that we make on the environment in which our solution is used. We view the network model as a point-to-point communication network, connecting the various entities. We have three types of entities in our network model (other than the adversary): Severs, clients and the Admin.

Servers are envisioned as trusted entities in the sense that (1) data integrity on the servers is preserved (2) servers behave according to the specified protocol (3) upon validating an authorized request the server properly performs the operation and, when applicable, sends the right data back to the client. To model this assumption, we explicitly do not allow the adversary to gain control of any of the servers (nor do we allow it to introduce its own servers in the system).

Clients, on the other hand, are not trusted. Some of these clients may in fact be written by the adversary, and others may run on machines that were compromised. We model this by letting the adversary gain control over clients at wish: At any point in the attack, the adversary may "point its finger" at a client c_i, thereby gaining a complete control over this client. This includes the ability to read the current state of c_i, as well as the ability to replace its code with a code written by the adversary. We note that once the adversary gains control over some client, it can inherently access all the objects that this client can access. Our solution is therefore geared towards preventing the adversary to access objects that are not available to *any of the clients under its control.*

The Admin is a highly trusted application, runs on a secured machine that is capable of storing long-lived keys; it truthfully determines the access rights by computing $Perm(c_i, ID_{obj})$. Again, we formalize this by not allowing the adversary in our model to compromise the Admin.

Communication links. Since we build our solution over IPSec, we assume that each pair of entities are connected via an *authenticated channel*. Namely, the adversary cannot inject, duplicate, or modify the traffic on the links. All the adversary can do is observe traffic on the network, and use in an arbitrary way the links that are available to clients under its control.[5]

For our solution, we also require that the links between the Admin and the other entities are *secure channels*, that use IPSec *with encryption*. In our model, we therefore deprive the adversary of the ability to read traffic that is sent

[4]Similarly, the Admin is capable of determining the client permissions for the *Format* and *GetServerInfo* operations on a given server.

[5]The adversary can potentially also block messages that are sent on the links, thereby causing a denial-of-service situation. However, we will make sure that blocking messages cannot help the adversary in getting extra permissions for objects.

to and from the Admin. The adversary can only do traffic analysis on these links (and potentially also block messages on them). Of course, the adversary can still read messages that are sent between the Admin and clients under the adversary's control.

Clients, machines, and links. In reality, the clients in our system reside on various machines in some domain, and it may happen that several clients reside on the same machine. On the other hand, the IPSec protocol, which we use to implement the authenticated/secure channels, can only establish channels between two machines. We therefore rely on the operating system of the client machines to effectively separate the different clients from each other, thereby multiplexing the single machine-to-machine channel into separate machine-to-client channels.

If a rogue client is able to compromise the separation mechanism of the OS, we then must view the machine as compromised and all the clients on this machine must be viewed as if they are controlled by the adversary. In other words, if an un-trusted code is run on a machine with weak sandboxing capabilities, one must assume that everything that happens on that machine is controlled by an attacker. We comment that this issue does not arise for servers. For one thing, we assume that the servers are trusted. Moreover, all the applications that we foresee have each server run on a separate machine.

Finally, we note that in our solution, we assume that the Admin can authenticate the clients (and servers) in the system[6], but the servers do not need to authenticate the clients (or even know who the clients are). In principle, we could therefore weaken the model, assuming that the links between clients and servers are *authenticated but anonymous* (from the server's point of view). Such links provide guarantee that all the messages arriving on the channel were sent by the same party and arrive without modifications, but they do not disclose the identity of that party. (In particular, this party could be the adversary.)

Security goal. Our notion of security is defined by means of a game that the adversary tries to win. An attack on the protocol is a run of the system, in the presence of an adversary, as described above. The adversary "wins" if at any time during this run, one of the clients under its control was granted access to an object, but of all the clients that the adversary controlled at that point in the run, none of them had permission for this access of that object.

Put in other words, at any point during an attack, the adversary can trivially get the union of permission of all the clients that it controls. It wins the game if at any point it manages to get a permission that is not in that union. We say that a protocol is secure is any feasible adversary has only a negligible probability of winning.

Note that our analysis (and, in particular, the security goal stated above) is aimed only towards guaranteeing the security of the access granting mechanism. The analysis does not explicitly address the integrity of the data on the disk against malicious modifications, nor does it address the authenticity of the data received over the link from the disk server. This restriction is made in order to simplify and focus the analysis. Indeed, guaranteeing date integrity and authentication, given a secure access-granting mechanism as described here, is straightforward. (In our solution it is done using IPSec.)

Two comments. Before concluding this section, we point out that a somewhat stronger requirement from the protocol may be possible in principle. Namely, we can extend the notion of a "win" for the adversary to include an event in which one of the clients under its control was granted access to an object, but *this client* does not have permission for this access of that object. (Even if other clients that are controlled by the adversary have this permission.) A more stringent notion of security would require that even the probability of such a "win" is still negligible. However, any protocol that satisfies this more stringent definition would necessarily disallow delegation of permissions: If clients are allowed to delegate permissions to other clients, then a client that is controlled by the adversary can delegate its permissions to any other adversary-controlled client, thereby violating this "more stringent" condition. By design, our protocol allows delegation, so it does not meet the "more stringent" notion. This choice also let us avoid the need for servers to authenticate (or even recognize) the clients.

Finally, we comment that it is also possible to use a simulation-based definition of security. In that approach, one defines an "ideal world" that has an ideal access-control functionality, and shows that anything that the adversary can do in a run of the real protocol can also be done in this "ideal world". We suspect that in our case these two notions coincide, but we did not check it.

3. Proposed Solution

3.1. Overview

Our system uses a credential based access control mechanism (see [7, 14]) to enforce the access control policy. To access an object, the client must provide a credential issued by the Admin. The credential serves as a capability, in that it stores a list of operations that the client is allowed to perform on a given object. The server can verify that the cre-

[6]Namely, the Admin is capable of mutually authenticating every client $c \in \{c_1, \dots, c_n\}$ and server $s \in \{s_1, \dots, s_m\}$ by some external mechanism, e.g. login/password or PKI certificates. The authentication mechanism itself is outside the scope of this work.

dential was generated by the Admin, and that it has not been altered.

However, if the credential was only a token expressing capabilities, then anyone seeing the token could gain access to the object. In particular, if the adversary could see the token sent on the network, it could later use the same token to access the same object. To prevent this, the Admin generates some additional secret information that is associated with the token. The credential that the client receives from the Admin (over a secure channel) contains both the token and the associated secret information. The client sends only the token to the server, and uses the secret information to validate this token.

Specifically, the token contains an "encryption" of a secret key K', under a key K_E that the server shares with the Admin. The key K' is the "additional secret information" that is sent to the client over a secure channel. The client forwards the token to the server, and appends to it a tag, computed as $MAC_{K'}$(channel-name).[7] The server can recover K' from the public token and check that this token indeed arrived on the right channel. This way, we ensure that this token cannot be re-sent over other channels, unless the sender knows the corresponding K' value.

There are three points worth noting about this solution. First, the server does not need to authenticate the client, or even to have any notion of "client identity". It is sufficient for the server to verify that whoever sits on the other side of the channel has a valid token, and it is able to use the K' value hidden in that token to tie it to the channel.

Second, delegation is very easy to achieve here. To delegate a permission, all a client needs to do is to forward both the token and K' to the intended recipient over a secure channel. (To delegate just part of its permissions for an object, the client must go to the Admin and ask for a token containing only these permissions.)

Third, notice the way in which the authenticated channel between the client and server is used. In fact, the reason that the server-to-client direction is authenticated has nothing to do with access-control. We just want to ensure that when the server sends some data (e.g., in case of a *read* request), this data is received unmodified by the client. The authentication in the client-to-server direction is essentially an anti-replay mechanism. The protocol itself prevents copying the token from one channel to another, and the authenticated channel prevents the adversary from re-sending the token of a good client *on the same channel*. It also enables an additional performance optimization, as we describe next.

Caches and credential expiration. To improve performance, our solution utilizes two cache mechanisms: a client cache and an ObS cache. On the ObS side, we use caching to save some work on the server side. Once a token is validated, the sever may cache the token and associate it with the channel on which it arrived. Future requests that arrive on this channel can be compared against the permissions in this token, without having to re-validate it every time. As the channel is authenticated, the server can trust that the party on the other end remains the same throughout, and there is no need to check its permissions again. Of course, credentials may be removed from the cache at any time (e.g., because of cache size limitations), so a client must send the token and validation tag with every request.

On the client side, caching valid credentials reduces connections to the Admin. Once a client receives a credential for some object(s), it can use this credential to perform many operations on the object(s) without having to contact the Admin. In order to reduce the computational load at the client side, we utilize the fact that the channel-name does not change throughout the session. Thus the client is required to re-authenticate (computes its tag) only once and cache the tag along with the token that the Admin sent.

As we explained above, credentials are constructed using shared keys between the Admin and the ObS. These keys are refreshed periodically (e.g., once an hour), thereby causing all the credentials that depend on them to become invalid. Such credentials are then flushed from the ObS cache and a client that tries to use an expired credential will get a "bad credential" error, forcing it to go to re-acquire the credential from the Admin. (We remark that a client cannot delegate the ability to get a credential from the Admin. Therefore, when the credential expires, the client has to get a new one from the Admin and re-delegate it.)

3.2. Detailed description

In the description below, we let $E_K(\cdot)$ be an "encryption function"[8]and $MAC_K(\cdot)$ be a keyed message authentication code. See Section 5 for some comments about the implementations of these primitives.

3.2.1 Credential structure

The credentials are based on a set of keys that are shared between the ObS servers and the Admin. For each server s_j, let K_{E_j} be an encryption key shared between s_j and $Admin$, and K_{A_j} be an authentication key shared between them. When there is no ambiguity, we will omit the subscript and use the notation simplified K_E and K_A.

Let c_i be a client, requesting the credential to operate on an object with ID ID_{obj}, which resides on server s_j. Recall

[7]The channel name is just a unique identifier that was chosen by the server when this channel was created.

[8]We put "encryption" in quotes, since in our implementation E_K is a deterministic function, and therefore cannot be seen as a secure encryption by itself. Formally, this is a pseudorandom permutation. Nonetheless, we will keep calling it "encryption", as it gives better intuition for the role that it plays in the protocol.

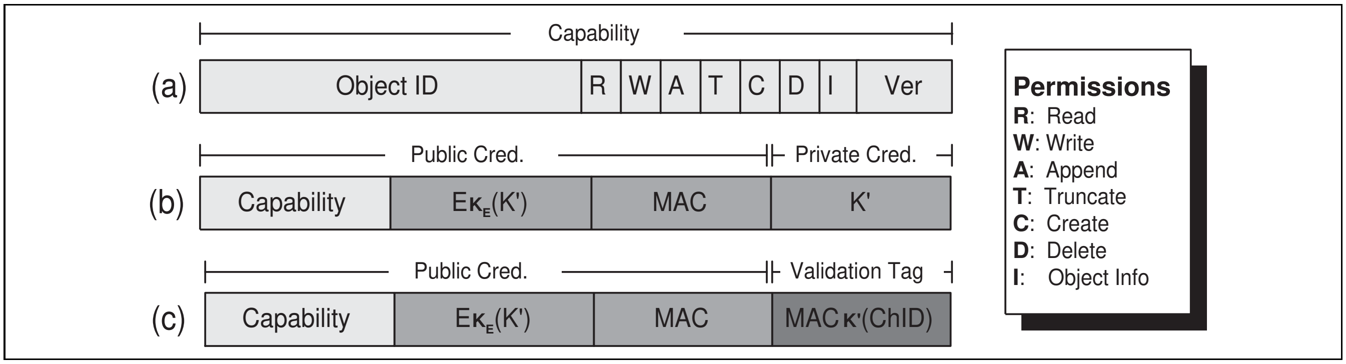

Figure 2. Credential Structure: (a) Capability structure; (b) The credential the Admin sends to the client. $E_{K_E}(K')$ is the credential secret (K') encrypted with K_E - the encryption key that the server shares with the Admin; (c) the credential sent by the client to the ObS. The public token is accompanied with a MAC tag, computed on the channel name (*ChID*) using key K'.

that the vector $Perm(c_i, ID_{obj}) = \overrightarrow{P}$ represents the permissions of c_i on object ID_{obj}[9]. As explained above, the credential $\mathcal{C}$ that the Admin issues for a client is comprised of two components, a "public token" $\mathcal{C}_{pub}$ and a "secret extra information" $\mathcal{C}_{priv}$. When issuing a credential with permissions $\overrightarrow{P}$ to client c_i for access of object ID_{obj} on server s_j, the Admin first picks an l-bits random string K' (in our implementation $l = 128$). The Admin sets

$$\begin{aligned}
&\mathcal{C}_{\text{priv}} \leftarrow K' \\
&T \leftarrow [ID_{obj}, \overrightarrow{P}, Ver\#, E_{K_{E_j}}(K')], \\
&\mathcal{C}_{\text{pub}} \leftarrow [T, \text{MAC}_{K_{Aj}}(T)]
\end{aligned}$$

and sends the credential $\mathcal{C} = [\mathcal{C}_{priv}, \mathcal{C}_{pub}]$ to client c_i over a secure channel.

(The field $Ver\#$ is the version number of the keys K_{A_j} and K_{E_j}. This field is not essential for the correctness of the credential, but makes the verification procedure at the server more efficient.)

Notice that an "encryption" of K' (under the shared key between the server and Admin) is included in the public part of the credential. It is the knowledge of K' that lets the server and client compute and test the tag value to validate the credential. Figure 2 depicts the credential structure.

3.2.2 The Client – Admin Protocol

The protocol between the client and the Admin, depicted in Figure 3(a), is rather straightforward:

1. The client asks for a capability (i.e. an ID_{obj} and a request)

2. The Admin verifies the clients permissions, generates a credential $\mathcal{C}$ as above, and sends it to the client.

Note that the credential contains the key K', hence it must be sent to the client over a secure channel. To establish this channel (and also to let the Admin identify the client), the client and the Admin should authenticate each other in a preliminary step. This authentication is not part of the protocol.

3.2.3 The Client – ObS Protocol

The protocol between the client and the ObS is depicted in Figure 3(c). It consists of a handshake stage that established a *Security Window*; once established, many requests/replies are exchanged spanning multiple objects and credentials; finally the window closes. The scope of the security window is related to the lifetime of the keys K_E and K_A - namely, a window expires once new keys have been exchanged between the ObS and the Admin. The security window is also confined to a single connection (IPSec connection for example). The three parts are:

Part 1 - Open Security Window.

1. The client requests an 'open security window' with the ObS.

2. The ObS responds with a randomly chosen l-bit channel name $ChID$.

Part 2 - Request/Reply.

3. The client sends a request to the ObS along with a public credential $\mathcal{C}_{pub}$ and a validation tag $V = MAC_{K'}(ChID)$.

[9] For simplicity, we assume that object IDs are globally unique, otherwise s_j should be taken as an argument when calculating the permissions.

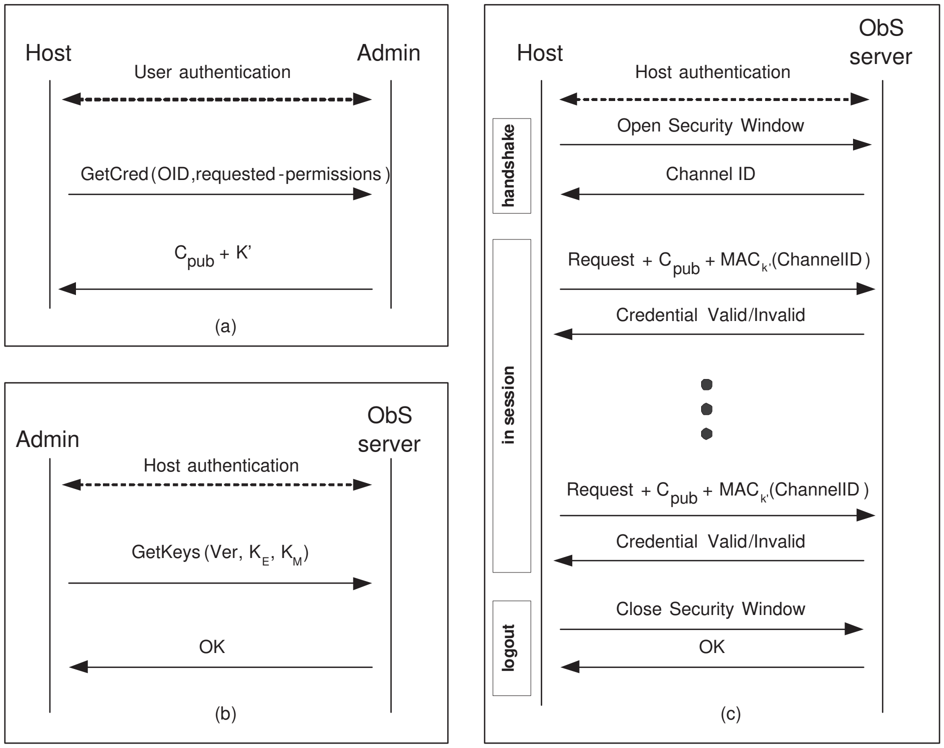

Figure 3. Security protocols: (a) Client-Admin; (b) Admin-ObS protocol; (c) Client-ObS.

4. Upon receiving a request $\mathcal{R}$, token $\mathcal{C}_{pub}$, and validation tag V on channel $ChID$, the ObS first verifies that the permission vector $\overrightarrow{P}$ in the token $\mathcal{C}_{pub}$ matches the request. If not, the request is denied. If the permission vector matches the request the ObS looks for $\mathcal{C}_{pub}$ in its cache, associated with this channel. If $\mathcal{C}_{pub}$ is found in the cache, it grants the request.

   If $\mathcal{C}_{pub}$ is not found in the cache, the server parses it as $\mathcal{C}_{pub} = [T, A]$, with $T = [ID_{obj}, \overrightarrow{P}, Ver\#, C]$. It then checks the following:

   (a) The version number $Ver\#$ matches the authentication and encryption keys that it shares with the Admin.

   (b) The authentication tag A equals $MAC_{K_A}(T)$.

   (c) The validation tag V equals $MAC_{K'}(ChID)$, where K' is obtained as $K' \leftarrow E^{-1}_{K_E}(C)$.

   If any of the checks fails, the request is denied. If all of them pass, the server *may* cache the token $\mathcal{C}_{pub}$, associated with channel $ChID$.

Part 3 - Close Security Window

5. The client closes the security window with the ObS.

6. The ObS clears the security window cache and acknowledges the request.

3.2.4 The ObS – Admin Protocol

This is the exchange protocol of the pair of keys K_E and K_A over a secure (authenticated and secret) channel. The protocol is depicted in Figure 3(b).

1. The Admin sends a new pair of keys to the ObS along with their version number.

2. The ObS stores the new keys, and removes permissions with the lowest key version.

In case no network communication between the Admin and the Server is provided (Fiber Channel, for example), it is possible to piggyback these keys, encrypted with a shared key between the Admin and the server (or any other method), on the credential itself. The client is then responsible for sending a credential, along with the new version of keys K_E and K_A, in an encrypted form.

4. Security of Protocol

To prove the security of our solution, we need to argue that an adversary in our model cannot access objects other than those that are permissible to clients under its control. For that, we describe a reduction to the security of the underlying primitives (E_K and MAC_K). In this proof we ignore the key-distribution mechanism between the Admin and the other entities, and simply model these keys as random and secret. We prove the following:

Lemma 1 *Assume that there exists an adversary that "wins" the security game (as described in Section 2) with probability ϵ. Then at least one of the following two things must hold.*
(a) There exists an attacker that breaks the MAC algorithm with probability at least min($\epsilon/3M$, $\epsilon/3N$), where M is the total number of authentication keys that where exchanged between the Admin and the servers, and N is the total number of credentials that the Admin generates in the run.
(b) There exists an attacker that distinguishes the "encryption" function from a random permutation with advantage of at least $\epsilon/3N$.

Proof: Recall that an adversary "wins" in a particular run of the protocol, if at some point it is granted access to an object, but this access is not in the union of permissions of all the clients under the adversary's control at this point in the run.

Denote the set of clients that the adversary controlled at the point when the "win" access was granted by CL^*, let $ChID^*$ be the name of the channel on which the "win" request arrived, and let s_{j*} be the server that granted it. Denote the token that was sent to the server along with the "win" request by

$$\mathcal{C}^*_{pub} = [T^*, A^*], \quad \text{where } T^* = [ID^*_{obj}, \overrightarrow{P}^*, Ver\#^*, C^*]$$

and denote $K^* = D_{K_E}(C^*)$ (where K_E is the shared encryption key between the ObS s_{j*} and the Admin, corresponding to the version number $Ver\#^*$).

By inspecting the Client–ObS protocol, we see that the server would only grant a request on channel $ChID^*$ if the permission vector $\overrightarrow{P}^*$ matches the request, and the token $\mathcal{C}^*_{pub}$ has passed all the tests in Step 4, either at this request, or at an earlier request on the same channel (when $\mathcal{C}^*_{pub}$ was entered into the cache). Denote the validation tag that was sent with $\mathcal{C}^*_{pub}$ when it passed the test 4(c) by V^*.

Looking at the history of the run up to this point, we distinguish between two possible scenarios. We say that this run is a *win of type 1* if a token $\mathcal{C}_{pub} = [T^*, A]$ with *the exact same value* T^* as in $\mathcal{C}^*_{pub}$ was sent at an earlier point in the run by some client $c_i \notin CL^*$ to s_{j*}. Otherwise, it is a *win of type 2*. Since the probability of a win is ϵ, we know that either wins of type 1 happen with probability at least $2\epsilon/3$, or else wins of type 2 happen with probability at least $\epsilon/3$. The next two lemmas complete the proof.

Lemma 2 *If wins of type 1 happen with probability δ, then either there exists an attacker that breaks the MAC algorithm with probability at least $\delta/2N$, or there exists an attacker that distinguishes the "encryption" function from a random permutation with advantage of at leasts $\delta/2N$.*

Lemma 3 *If wins of type 2 happen with probability δ, then there exists an attacker that breaks the MAC algorithm with probability δ/M.*

■

Lemma 3 is a bit easier to prove, so we start with it, and then prove Lemma 2.

Proof sketch for Lemma 3: The intuition here is as follows: consider a run of the protocol in which the adversary has a win of type 2. On one hand, since $\overrightarrow{P}^*$ matches the "win" request, then it means that it includes some permission that none of the clients under the control of the adversary has. Therefore, we know that the Admin never sent to any of these clients the token $\mathcal{C}^*_{pub} = [T^*, A^*]$ (or any other valid token with the same T^* value), destined to server s_{j*}. On the other hand, the adversary never saw any token with the same T^* value sent by any of the honest players to s_{j*}. Thus, the adversary never saw a valid authentication tag for T^* with respect to the appropriate authentication key $K_{A_{j*}}$, and yet it was able to produce the valid tag A^*. This violates the security of the MAC algorithm.

Translating this intuition into a formal proof is straightforward. The factor of M in the lemma statement is due to the fact that there are total of M authentication keys, and the adversary can win by breaking any of them. ■

Proof sketch for Lemma 2: Here the intuition is as follows: We know that T^* appeared in a token that was sent by an honest client c_i to s_{j*}, so the admin must gave T^* to that client. Note that the Admin chooses all the K' values in the tokens independently at random, and the adversary never gets to see the internal memory of c_i (or anything on the channel between c_i and the Admin). Therefore, the only information that the adversary has about K^* (which is "the K' value inside T^*") was obtained by observing the channel between c_i and s_{j*}. And the only things on that channel that are related to K^* are its "encryption" C^*, and the validation tag $V = MAC_{K^*}(ChID)$ (where $ChID$ is the name of the channel between c_i and s_{j*}).

Yet, from the fact that T^* was used in a "win request", we learn that the validation tag V^* passed the test in step 4(c), namely, $V^* = MAC_{K^*}(ChID^*)$. Now, $ChID^*$ and $ChID$ are the names of two different channels of server s_{j*}, so $ChID^* \neq ChID$, but still, the adversary could come up with the value V^*. Thus, there are two options: either the "encryption" C^* leaks some information about K^* that is not leaked by a truly random permutation, or the MAC function is broken, and the adversary can generate an authentication tag for $ChID^*$ from an authentication tag for $ChID$.

Translating this intuition into a formal proof is not as easy as for the other lemma. This can be done as follows: We consider an "imaginary game" in which we run the adversary against the following modified protocol: when the Admin gets a request from client c_i to server s_j, it prepares the credential in the same way, but instead of putting in the token $E_{K_{E_j}}(K')$, it puts there $E_{K_{E_j}}(K'')$, for some independently chosen l-bit key K''. The Admin now sends K' and $\mathcal{C}_{pub}$ to both the client *and the server* (over secret channels). When the server receives the token $\mathcal{C}_{pub}$ from the client, it ignores the K'' in that token, and instead uses the key K' that it received directly from the Admin in order to check the validation tag V.

We now ask what is the probability of a "type-1 win" in this new game. If it is still more than $\delta/2$, we can break the MAC algorithm, since now the adversary computes $MAC_{K^*}(ChID^*)$ from $MAC_{K^*}(ChID)$ *without any other information about* K^*. If, on the other hand, the probability of "type 1 win" in the new game is less than $\delta/2$, we could use the $\delta/2$ difference between the win probabilities in both games to distinguish the "encryption function" $E_K(\cdot)$ from a random permutation. The factor N in both cases comes from the fact that there are total of N credentials that the Admin sends to honest clients, and the adversary can win by breaking any of them. ■

5. Prototype Implementation

We implemented our proposed security mechanism for *Antara*, IBM's implementation of an ObS [1]. *Antara* is a "session-based" ObS, in the sense that a client can sends its requests only within a session context it establishes with the server. Admissible requests to the server are *Create, Delete, Read, Write, Append, Truncate*, and *GetObjectInfo*. *Antara* provides the basic object store functionalities and has a virtue that it is easily extensible. Utilizing the last property, we extended *Antara*'s basic client and server to provide security according to our model. In addition, we implemented a prototype of the Admin. Thus, the implementation is comprised of three different modules: the Admin, the client security extension and the server security extension, all using a core module the provides common functionalities.

Our implementation is multi-platform and cryptographic library independent: all modules can be used in Windows and Linux user space. The client extension, can be used inside Linux kernel. In addition any cryptographic library that provides the required primitives (AES block encryption and HMAC-SHA1[4]) can be used.

Sections 5.1-5.4 describes the (high level) design of each of the modules and reviews several implementation points of importance. Section 5.5 provides preliminary experimental results.

5.1. The Admin

The Admin has two roles: (i) authenticating clients, authorizing their requests according to the system protection policy, and generating suitable credentials; (ii) refreshing the ObS's keys on an hourly basis. Since the system protection policy and user authentication is outside the scope of this work we implemented only the credential generation facility and key refresh mechanism. We implemented the Admin as a library that provide the aforementioned operations and intend it to be linked with any conforming metadata servers.

5.1.1 Credential Generation

The Admin keeps a table of the ObS's in the system and their current encryption key, authentication key and key version (see Section 3.2). Given an object ID, the server id, and the permissible operation, the Admin generates the credential in five stages:

1. Locating the server keys in the Admin's ObS table.
2. Generating capability in an host independent format.
3. Creating a random string K' of length 128 bits.
4. Encrypting K' with the ObS encryption key.
5. Calculating a keyed MAC (using HMAC-SHA1) of the concatenation of the capability and the encryption of K'.
6. Concatenating the capability, the encrypted secret, and the first 96 bits of the calculated MAC value, thus generating the credential.

One implementation detail worth mentioning is the generation of the random string K'. This is done by defining a random number generator from an AES function as follows: let K_G be an AES key. The Admin generates K' by encrypting $AES_{K_G}(cnt++)$ where *cnt* is a counter. Since the Admin may generate a very large number of credentials during its lifetime and each credential needs a unique string K', a new K_G is generated periodically (possibly at boot time) where K_G is derived from a master secret AES key K_R, that is $K_G = AES_{K_R}(time)$, where time is the number of seconds since Unix epoch as returned by the standard system call *time()* .

5.1.2 Key exchange

The Admin provides key generation functionality (for both encryption and authentication keys). It is the meta-data server responsibility to send the new keys to the ObS. The encryption key (128 bits long) is generated in the same way a secret is generated, the HMAC-SHA1 key (160 bits long) is generated by creating two secretes, concatenating them and taking the first 160 bits.

The ObS accepts the sent keys only if they are accompanied with a special key-exchange credential (not described before in order to simplify the presentation). This credential is cryptographically hardened using the ObS current keys. The Admin also keeps a long lived set of authentication and authorization keys with every ObS that are used for bootstrapping: the first key-exchange request the ObS received are hardened using these long lived keys. These keys can be used only for this purpose (the ObS will not authorize any other request "hardened" by these keys) in order to minimize their use.

5.2. The Client Security Extension

The client security extension is built on top of the "basic" ObS client. When a session is established with the ObS it negotiates for a "channel name" (see Section 3.1). After establishing the session it piggybacks every message sent by the "basic" client with a public credential that authorizes the request and a validation tag (see Section 3.1). In order to reduce the computational overhead at the client, we calculate the validation tag only once and cache it. Since neither the "channel name" nor the credential secret change throughout the session, the same validation tag can be used until the credential itself expires.

5.2.1 Validating the Credential

To simplify the presentation above, we described the generation of the validation tag by applying the same MAC function as for the authentication tag (i.e., HMAC [4]). However, in the implementation we instead compute it as

$$V \leftarrow AES_{K'}(ChID)$$

We note that since AES is assumed to be a pseudorandom permutation, it is in particular a good MAC (for fixed-size messages). The reason that we use AES rather than HMAC-SHA1 is to save on key-size: AES keys are only 128 bits keys, whereas HMAC-SHA1 keys are 160-bit long. Since the public credential contains an "encryption" of K' using AES, we need the size of K' to be exactly one block. Since the client calculates the validation tag only once for every credential, the added cost of performing an AES block encryption rather than HMAC calculation is practically insignificant.

5.3. The ObS Security Extension

The ObS security extension is used by the ObS basic server as a "request filter". Before honoring any client request (except of opening or closing a session) the security extension is asked to approve the request. The request is performed only if the security extension approves it. Otherwise a "request rejected" error response is sent back to the client.

In order to authorize the client requests the security extension manipulates several tables:

Key table The extension stores the last 255 pairs of keys sent by the Admin, however only the last two pairs received are effective. The extension associates each pair with a key version which is a cyclic 8 bits counter set by the Admin (key version 0 is used for the bootstrapping keys - see Section 5.1). Whenever a new pair of keys is received it overwrites the previous content of the table and become, with the last received pair, the new active keys. The server accepts credentials that were generated with the current key and its predecessor. This is done in order to prevent congestion at the Admin, as suggested in [7].

Session table The security extension keeps a map between sessions and "channel names". When a client requests to open a new session, the security extension generates a new name for this session and store it in the table. The "channel name" (see Section 3.1) is generated by concatenating a 4 bytes counter with the last time the counter had value 0 (happens when the system boots and every time the counter cycles) and sends it to the client.

5.3.1 Credential Verification

A credential is verified in the following manner:

1. Verify that the credential contains enough permissions for the requested operation.
2. Verify that the credential's key version is active.
3. Perform cache lookup to see if a credential containing the required permissions as verified for the session (a detailed description appears below).
4. Find the key version encryption key and authentication key in the key table.
5. Computing the HMAC of the credential capability concatenated with the encrypted secret by applying HMAC-SHA1 with the authentication key and comparing the first 96 bits with the public credential MAC value.

6. locating the session's "channel name" in the session table.

7. Decrypting the credential secret using the encryption key; encrypting "channel name" with the credential secret and comparing the result with the validation tag

Note that the cache lookup is done after the extension verifies that the credential permits the requested operation and that the credential is still valid. This is done in order to provide protection against erroneous, but maybe not malicious, requests (the client is expected to send a valid credential with each request).

5.3.2 Credential Cache

The credential cache is used to speed up the verification process by caching verified credential for each session. The cache is implemented as two separate caches - one for each active key version. Each table contains a subset of the credentials that were successfully verified using its associated key version. When keys are replaced, the table associated with the key that becomes inactive is cleared, and then associated with the new key version.

Each of the two "key version" caches is comprised of an ObS credential cache and object credential cache:

The ObS credential cache. The ObS credential cache contains for each session the operations it is authorized to perform on the entire ObS or on every object (in other words, the permissions incorporated in credential of type "entire server credential" that the client presented (see Sections 3.1 and 5.4)). Since the number of session is bounded a-priori this part of the cache is implemented as an array which contains an entry for each session.

The object credential cache. The object credential cache contains for each session the operations it was authorized to perform on specific objects. It is implemented based on the marker algorithm [6]. Every cache line contains a (bounded) map of sessions to permissions pertaining to single object ID. When the "map" is organized in a LRU manner. Thus, when it overflows, the session that did not use the object for the longest time is ejected from the cache. In order to find the cache line with the request object ID in an efficient way, we keep a map of object IDs to cache lines implemented by a hash table.

5.4. Miscellaneous

Wildcard Credentials. So far we defined credentials to contain a permission for an object specific operation. Our system actually allows a more general type of credentials, the *wildcard* credentials. These credentials provide permissions for an entire Object Server e.g., *Format*. Also supported are "super-user" credentials, that provide a permission on an entire ObS and. e.g., read, effectively authorizes its owner to perform the operation on any object in the ObS. For that, an added field in the credential, the "type" field, is used to distinguish between credential containing rights for a specific objects and credential that provides permissions for operations on an entire server.

Cryptographic Session. Our implementation takes advantage of the fact that the object store is session-based, namely that the client establishes a session with the ObS in order to send requests. Hence, the cryptographic session is build upon the object-store session and adds the necessary cryptographic parameters to the already existing messages.

Cryptographic Primitives. Our implementation utilizes two main cryptographic primitives, the AES and SHA1. For that, we use the RSA BSafe-C6.0 cryptographic library [15].

5.5. Experimental Results

In order to estimate the performance overhead of the security mechanism we ran some initial tests to measure the cost of generating and verifying a credential. We specifically do not include IPSec in our measurements since we would typically expect IPSec to be supported by a dedicated hardware that will minimize its effect on performance. Running our prototype implementation on a Pentium-IV 2.0Ghz with 1GB memory running Linux RedHat with kernel version 2.4.18, we were able to:

- Generate 21,000 credentials in a second.
- Verify 20,000 credentials in a second (without a cache).

Our experimentation shows that the credential cache improve the performance by up to a factor of 50, depending on the cache hit ratio.

6. Conclusions

We have described our design and implementation of a security mechanism for an object store. Our model has address the issues enumerated in Section 1.3 successfully. Our security mechanism, which separates the security of the transport from the access control enforcement, can be applied to both Fibre-Channel and IP networks and as such is novel.

We are currently conducting performance tests of our implementation. Preliminary performance measurements demonstrate that while work is still required, the security overhead is reasonable and does not degrade the I/O performance substantially. Further optimization is needed in caching strategies, key refresh strategies (to avoid congestion at the Admin) and possibly richer credentials.

The use of object stores instead of traditional block-based storage devices is a paradigm shift. We believe we are now at the point where the ability to leverage the benefits of an object store to democratize the access to data suffices to justify the cost of the shift. Security and its performance is a key aspect in achieving this goal.

Acknowledgments We would like to thank V. Dreizin, A. Tavory, and L. Yerushalmi for the use of the *Antara* ObS. We would like to thank Ted Anderson, Ralph Becker-Szendy, Mark C. Davis, Leo Luan, Benjamin Reed and Miriam Sivan-zimet for reviewing an earlier version of the paper. Finally, we thank the anonymous referees for valuable comments.

References

[1] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, L. Yerushalmi. Towards an Object Store. Submitted for MASS 2002.

[2] M. Blaze. A Cryptographic File System for Unix. *Proceedings of the First ACM Conference on Computer and Communications Security,* Fairfax, VA, November 1993

[3] M. Blaze. "Key Management in an Encrypting File System." USENIX Summer 1994 Technical Conference, Boston, MA, June 1994

[4] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions: The HMAC construction", RSA Laboratories' CryptoBytes Vol. 2, No. 1, Spring 1996.

[5] G. Cattaneo, L. Catuogno, A. D. Sorbo and P. Persiano, The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In Proceedings of the Freenix Track: USENIX Annual Technical Conference, 2001. pp. 199-212.

[6] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator and N. E. Young, "Competitive Paging Algorithms", J. Algorithms, 1991, Vol. 12, No. 4, pp. 685-699.

[7] H. Gobioff, Security for High Performance Commodity Storage Subsystem, Ph.D thesis, CMU, July 1999.

[8] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka. "A Cost-Effective, High-Bandwidth Storage Architecture". In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998: pp. 92-103.

[9] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, J. Zelenka: File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the ACM International Conference on Measurement and Modelling of Computer System*, Seattle, WA, June 1997.

[10] D. Mazieres, M. Kaminsky, M. F. Kaashoek and E. Witchel, Separating key management from file system security. In *Symposium on Operating Systems Principles*, 1999, pp. 124-139.

[11] E. Miller, D. Long, W. Freeman and B. Reed, Strong Security for Network-Attached Storage. In *Proceedings of the FAST 2002 Conference on FIle and Storage Technologies*, January 2002.

[12] E. Reidel, M. Kallahalla and R. Swaminathan, A framework for evaluating storage systems security. In *Proceedings of the 1st conference on File and STorage Technologies (FAST)*, January 2002.

[13] T10 Working draft, "Object Based Storage Devices Command Set (OSD)", http://www.t10.org/drafts.htm

[14] B. C. Reed, E. G. Chron, R.l C. Burns, D.E. Long, Authenticating Network-Attached Storage, *IEEE Micro* Vol. 20(1), pp. 49-57, January 2000.

[15] RSA BSAFE(R) - CryptoC - Cryptographic Components for C Reference Manual - Version 6.0, December 2001, www.rsasecurity.com