

Security Enhancement and Performance Evaluation of an Object-Based Storage System

Po-Chun Liu, Sheng-Kai Hong, and Yarsun Hsu

Department of Electrical Engineering
National Tsing Hua University Hsinchu, 30013, Taiwan
{pcliu,phinex}@hpcc.ee.nthu.edu.tw, yshsu@ee.nthu.edu.tw

Abstract. Object-based storage offloads some works of file systems to storage devices to improve security, scalability, and performance. Security is a main concern when sharing data over network. We examine the security model of object-based storage and find that there is some problem in the model. It can be disabled by modifying specific field in the command. We propose a solution to this problem by encryption that makes unauthenticated clients impossible to alter the field. The overhead of this encryption is quite low. Thus the performance of our enhanced object-based storage system is comparable to that of the original one while offering an enhanced security. In addition, we have compared the performance of OSD systems with that of iSCSI and NFS. The write performance of an object-based storage system is much better because it can offload some tasks to storage devices, and the CPU usage at client side is also largely reduced.

1 Introduction

The design of storage subsystems becomes an important issue with the improvement of computer system. Although direct attached storage (DAS) devices have much better performance compared to network attached storage (NAS), the advantages of NAS, such as scalability and file sharing, make it become important in large scale computer systems. However, when a storage subsystem is connected to internet, security is always a main concern for system designers. Object-based storage systems [1][2] provide a security model which protects the shared data in storage devices and eases the storage subsystem design in a very large scale computer system.

The main concept of object-based storage is to offload the space management component of existing file system to the storage device itself. Application clients thus request for an object (or file) instead of many disk blocks. The new interface can reduce traffics between application clients and storage devices. And the application clients only need to manipulate hierarchy management, naming, and user access control. The work data structure mapped to physical organization and disk space management is left to be done in storage device.

The object-based interface is defined in details by OSD T10 standard [3] and the aim of our work is to build an OSD (Object-based Storage Device)

system compliant to this standard. Our implementation is based on IBM Object Store Initiator [4][5] and Intel iSCSI/OSD reference implementation [6]. However, we find that there is a potential problem in their security model. The security method can be altered by unauthenticated application clients to disable the security model. In this paper, we propose an enhanced security model to resolve the potential problem. The main idea is to encrypt the security method field and make others impossible to alter the security method. The encryption does not add much computing overhead to the system. We will illustrate the architecture and design issues of our enhanced OSD system.

Section 2 gives the system architecture of our OSD system and briefly explains the design of the system. Section 3 discusses the security model of OSD T10 standard and points out the potential problem. In section 4 we discuss the possible solutions for the potential problem of the security model and give the performance result in section 5. We give the conclusion and what to do in the future in section 6.

2 OSD Environment

Currently the object-based hard disk drives compliant to the T10 standard are still not available and our OSD storage system is based on Intel iSCSI/OSD reference implementation, which consists of initiators and object-based targets, to emulate the underlying Object-based Storage Device (OSD). Initiators and targets are connected via iSCSI protocol over IP network [7][8]. The initiator prepares the OSD SCSI commands, requests for credential from security manager and packs it into OSD Command Descriptor Block (CDB) format which is a 200 bytes variable length CDB [9], and then issues it. The object based target receives the OSD CDB from the initiators, checks whether the command is illegal and then executes the command if it is a legal request. The architecture of the system is shown in Fig. 1, which consists of initiator, security manager, policy/storage manager, and storage device server.

2.1 Initiator

The initiator acts as an application client, which issues the OSD SCSI commands to access the storage device server. The difference between the OSD system and the traditional storage subsystem is that an application client of traditional storage systems needs to allocate and manage disk space for specific files while an OSD system does not care about it. An application client of OSD systems only needs to specify which object it wants to access and the mapping between the file name and the object ID. The storage space allocation and management are delegated to storage device server. The only thing an application client needs to do is specifying which object and its offset to be accessed. The application client does not concern about the disk space and thus its loading is eased compared to traditional storage system.

When an application client wants to issue an OSD command, it prepares parameters such as partition ID, object ID, the length and the starting address of

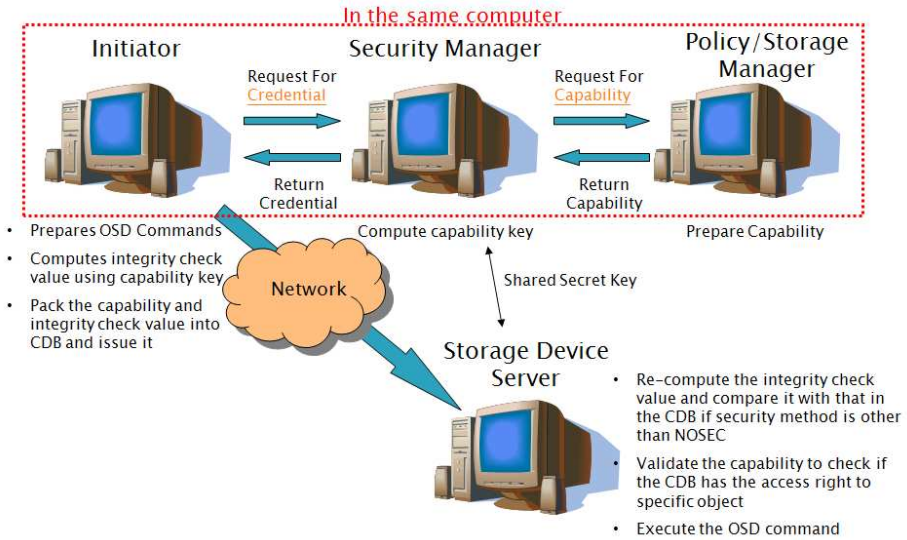


Fig. 1. The architecture of our OSD system

the object it wants to access. Before it issues the command, it needs to request a credential, which contains capability and capability key, from security manager. The capability is generated by policy/storage manager which coordinates requests from different application clients and determines their access rights. Once the application client gets the credential, it packs the capability contained in the credential into OSD CDB and sends the CDB via iSCSI initiator over network. The command it issues can be verified by storage device server and executed correctly with appropriate access right.

There are four supported security methods, NOSEC, CAPKEY, CMDRSP, and ALLDATA [3][10]. The NOSEC method means security model is disabled and all data is unprotected. The CAPKEY method protects the capability of the CDB and the CMDRSP protects entire CDB and responses from targets. The ALLDATA method makes sure all of the data between initiators and targets are under protection. If the security model is enabled, the initiator needs to compute an integrity check value before sending the command. In our system we use HMAC (Hash Message Authentication Code) [11] algorithm and use the secret key generated by security manager to compute integrity check value of specific data depending on the security method. The integrity check value is then packed into OSD CDB and the command is issued.

2.2 Security Manager

The main purpose of the security manager is to generate a key for initiators to compute the integrity check value if the security method is not NOSEC. Once the security manager receives a request from an application client, it hands over the request to policy/storage manager and requests for a capability. After the

security manager receives a capability back from the policy/storage manager, it packs the capability and OSD system ID of storage device server into credential. Then the security manager uses HMAC algorithm with the secret key shared with the storage device server to compute a check value of this credential. The check value, also called capability key, is the key for initiators to compute integrity check values. The security manager packs the check value into the credential and the credential is then sent back to the application client.

2.3 Policy/Storage Manager

The policy/storage manager receives requests from security manager and prepares capabilities depending on different requests. The capability contains parameters such as security method, integrity check value algorithm, read/write permissions, allowed object ID for specific command, and so forth. Without the appropriate capability an application client's access to the storage device server will be rejected. With an appropriate capability, the policy/storage manager can make application clients access only specific objects of the storage device server but not the entire disk space.

2.4 Storage Device Server

The storage device server receives OSD commands via iSCSI target driver from application clients and handles the space allocation and management according to the requests. Once it receives a request, it will verify whether the command had been tampered with by checking the integrity check value if the security method is other than NOSEC. The storage device server will first reconstruct the credential by copying both the capability from the CDB and the OSD system ID from the Root Information attributes page. Then it uses the key shared with security manager and HMAC algorithm to compute a check value of this reconstructed credential. Finally, the storage device server computes an integrity check value of the data depending on different security method by use of this check value. It then compares the reconstructed integrity check value with that in the CDB computed by application clients. If the two integrity check values are identical, we can assure that the protected data has not been tampered with. And if the two values mismatch, the protected data has been altered unintentionally or maliciously and thus the storage server will reject the command.

If the security method is CAPKEY, we can assure that the capability is the one that is prepared by the policy/storage manager. If the security method is CMDRSP, we are confident that the OSD CDB is issued by authenticated application clients. The ALLDATA security method makes sure that all the data transmitted between authenticated application clients and storage device server are not be altered.

If the command is not tampered with, the storage device server then checks whether the accessed object is allowed and the access right of the command depending on the capability in the CDB. The storage device server will also check the parameters such as object created time. Moreover, the specific permission bits must be set according to the service action and object type. Only when all

the parameters of capability are verified can the storage device server execute the OSD command.

The storage device server translates the OSD commands to VFS (Virtual Filesystem Switch) system calls and exploits existing Linux file system such as ext3 to do the space allocation and management. For example, OSD-CREATE-PARTITION can be translated to `mkdir()` of Linux file system to create a directory to be treated as a partition of the object-based storage. We treat directories of Linux file system as partitions and ordinary files as user objects.

2.5 Potential Problem

The security model of OSD systems provide some level of protection to avoid malicious alteration of data over network. The credential integrity check value, also called as capability key, is generated by security manager to compute the integrity check value of protected data. The unauthenticated application clients have no idea how to reconstruct the capability key because the key used to compute the credential integrity check value is only known by the security manger and the storage device server. Thus any unauthenticated application client cannot generate a valid integrity check value without knowing capability key and this is why the security model can avoid any malicious alteration by unauthenticated application clients.

Even though the integrity check value based security model protects the data from being altered, the CDB still can be seen by others. Unauthenticated application clients can capture the capability by monitoring the CDB sent by authenticated ones. If they change the security method field in the capability to NOSEC and set integrity check value of CDB to 0s, then they can still access the objects that the captured capability allows. All unauthenticated clients have to do is to capture a valid capability, modify the security method, and send the command using the modified capability to disable the security model. That is to say, unauthenticated application clients can force storage device server not checking the integrity check value by changing the security method field to NOSEC.

Because the security method of certain command fully depends on the capability, the storage device server will check the integrity check value only if the security method field of capability is not NOSEC. The security model will not work if unauthenticated application clients can tamper the security method field. In this study, we propose a solution to resolve this potential problem and assure that the security model always works.

3 Enhanced Security Model

The most intuitive solution to prevent unauthenticated application clients from tampering the security method field is to make it impossible for them to alter the capability. Encrypting the transmitted data is one way to hide the contents of the data. Even if unauthenticated application clients can capture the CDBs sent by authenticated ones, they still do not know how to decrypt the data without

the key, which is only known by storage device server, and thus they cannot tamper the capability.

The public key crypto-system is a well known method to encrypt and decrypt transmitted data. The receiver has its own private key and publishes a public key, which is usually generated by the private key, to the one which wants to transmit data to it. The transmitter encrypts the data with the public key of receiver and the receiver decrypts the data by the private key. Others can't decrypt the cipher text without the private key thus they don't know what the data is and cannot tamper it.

A well known RSA algorithm uses the public key crypto-system with a set of public and private keys to encrypt and decrypt. But the computing complexity of RSA is very high because the encryption and decryption of RSA need to compute the power of some integer. Although there are some methods to speed up the computation, it still needs lots of computing resources. We combine the concept of public key crypto-system and error correction coding, which is first proposed by McEliece [12], to encrypt the transmitted data and decrypt the cipher text. The encryption and decryption complexity is very low because all operations are in binary field and addition can be done by logic exclusive OR and multiplication can be done by logic AND. This largely decreases the need of computing power in encryption and decryption.

We design an encryption procedure in the initiator. It encrypts the specific field of the capability including security method with the public key of the OSD storage server before issuing the CDB. To encrypt specific data, just like the encoding process of error correction code encoder, it just multiplies the specific data by the public key G' of the OSD storage server, where G' is generated by the three private keys. Then it randomly flips t bits of encrypted data, which stands for t errors during transmission. The encrypted data can be described by equation $c = mG \oplus e$, where m is K bits data and e is error pattern with weight t . The public key G' is a $K \times N$ matrix, and K is the number of bits of data to be encrypted and N is the number of bits after encryption. We pack the redundant bits $N - K$ into the reserved fields of OSD CDBs and thus there is no transmission overhead.

There are three private keys, G , S , and P generated by OSD storage server. G is a generating matrix of a (N, K) code, P is an N by N permutation matrix and S is a nonsingular matrix with dimension K . Currently G is a generating matrix of a Hamming Code with easily generated generating matrix, thus how to design a random non-singular matrix is our main concern. We first generate a matrix with entries of random 0s or 1s and examine whether it is invertible. If the inverse of the randomly generated matrix exists, then we have found the non-singular matrix; if not, we regenerate another random matrix and examine it again. We can find the non-singular within four tries because the probability of a randomly generated matrix being non-singular is more than 25% [13][14]. Then the public key is generated by multiplying S , G , and P , $G' = SGP$.

The OSD storage server must decrypt the received CDB before computing the integrity check value. Because $G' = SGP$, we can rewrite the encrypt data

$c = mSGP \oplus e$. The server first multiplies c with the inverse of P and gets the equation: $c' = (mSGP \oplus e)P^{-1} = mSG \oplus eP^{-1} = m'G \oplus e'$. Here e' has the same weight as e and then the server can decode c' with error pattern e' to m' using generating matrix G . We compute the syndrome of c' and find the least weight error pattern and then decode c' to m' . Once m' is found, the original message can be decrypted by multiplying m' by the inverse of S because $m' = mS$ and then $m = m'S^{-1}$.

Both the encryption and the decryption process are very simple because the matrix multiplications can be replaced with logic AND in binary field. With this insignificant computing overhead we can provide an enhanced security model without adding much cost. Unauthenticated clients now cannot capture a valid capability to generate an unauthenticated CDB by just changing the security method in the capability.

4 Performance Evaluation

In this section we will show the performance of our system and compare it with other file systems. We use two identical machines with configuration listed in Table 1 and connect them with a 3COM gigabits switch. We use the program `bonnie++` version 1.03a [15] with file size set to 1 GB and block size 8 KB for all of the performance evaluations. We evaluate the performance under five environments:

OSD-NOSEC: In this environment we use the OSD system according to OSD T10 standard with security manager disabled. The keys and integrity check values are all set to zero. In the initiator we implement an OSD file system based on the Intel iSCSI/OSD reference implementation, and in the target we exploit existing `ext3` file system to manage the disk space. The initiator and target are connected through a gigabits network.

OSD-CAPKEY: In this environment we use the OSD system with security method set to `CAPKEY`. The security manager needs to generate a key for initiators to compute integrity check value. The initiator and target are implemented in the same way as `OSD-NOSEC`.

ENHANCED: This uses the enhanced security model we have proposed. The architecture is the same as an OSD system except an additional encryption process.

iSCSI: We use a storage subsystem which supports dual 1 Gbits iSCSI ports as the underlying storage medium. In the initiator we use Linux-iSCSI driver [16] to connect to the iSCSI storage through a gigabits network and mount it as a local disk.

NFS: A well known Network File System. Clients and file server are also connected through the same network as above.

In order to reduce variations between different machines, in all of the tests we use the same initiator and target.

Fig. 2(a) shows the performance when writing a character at a time of a total 1 GB file. Our enhanced security model has comparable performance to that of

Table 1. Configurations for initiator and target

| | |
|--------|--------------------------------------|
| CPU | AMD Sempron Processor 2500+ |
| Memory | 512MB DDR 400 RAM |
| HDD | WD IDE 7200rpm 160GB with 8MB buffer |
| NIC | 3COM gigabits NIC 3C200-T |

OSD-CAPKEY. The figure shows that our enhanced model adds almost zero overhead compared with the OSD T10 standard, but our proposed OSD system offers stronger security. This is because the matrix multiplication process can be replaced with logic AND in binary field. The overhead of security method CAPKEY over NOSEC is about 0.15 % . The overhead is insignificant because the HMAC process is quite simple and thus requires little computing power. Compared with iSCSI, our system is about 20 % faster when writing a character at a time. This is because that the iSCSI storage is block based, and the local file system packs a character into a block at a time, thus there is a huge overhead to write only a character at a time. When the OSD file system issues a WRITE command, it can issue another one as soon as it receives an acknowledgement response from the target without waiting for the target server to process the command. That is why the writing performance of an OSD system is better than iSCSI storage. The same reason accounts for the poor performance of NFS. Moreover, NFS is based on RPC (Remote Procedure Call) a client, after issuing a write command, needs to wait until receiving an acknowledgement from NFS server and this is why NFS is about 35.95 % worse than an OSD system.

Fig. 2(b) shows the performance when writing a block at a time for a 1 GB file. The performance improvement for both OSD systems and iSCSI storage are significant because a block now contains 8 KB data to be written instead of 1 byte in the per character write, thus the writing overhead is largely reduced. We can see that our enhanced model still offers stronger security with little performance degradation. Again we can see that the overhead of CAPKEY is still very small over NOSEC, and our OSD system performs better than the iSCSI storage because application clients do not have to manage the disk space. We find that the CPU usage for iSCSI storage in such a test is about 3 times of

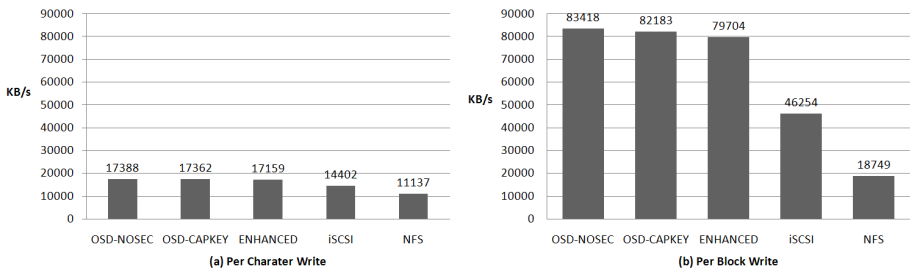


Fig. 2. (a)Performance when writing a character at a time (b)Performance when writing a block at a time

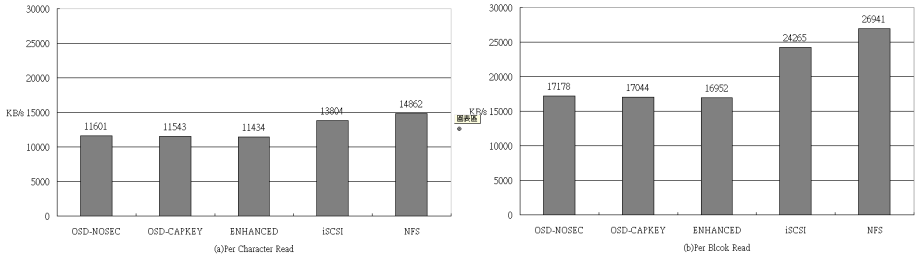


Fig. 3. (a)Performance when reading a character at a time (b)Performance when reading a block at a time

that for an OSD system (9.7% for an OSD system and 31.7% for iSCSI storage). This is because clients need to allocate and manage the disk space before issuing a write command but nothing of this sort has to be done in the OSD system. The performance difference between the OSD system and iSCSI storage in this test is due to disk performance, where performance of iSCSI storage is limited by disk. We can improve the performance by about 65 % if we configure the iSCSI storage to a RAID 5 storage subsystem. The performance improvement for NFS is smaller compared with OSD system and iSCSI storage due to RPC procedure because every command is limited by the RPC response time. The write performance of our OSD system is much better in this test.

Fig. 3(a) shows the performance of reading a character at a time. The poor performance in all environments is now again due to the huge overhead if it only reads a character at a time because the entire block needs to be transferred while only one character is needed. Now the OSD system is about 17 % slower than the iSCSI storage and this is due to the overhead of OSD device server when processing a READ command. Unlike in the writing case, where the OSD file system does not need to wait for the target server to process the command, it now needs to wait for the target server to process the command and return accessed data. The overhead of the target server processing the OSD command includes security check, capability validation and command execution. Thus the overhead causes the performance degradation in reading data compared to iSCSI storage. But the CPU usage of initiator is reduced by about 18 % (78.2 % in iSCSI storage and 63.7% in the OSD system with NOSEC) in the OSD system due to the offloading of disk space management to the target server. The NFS performs better than both iSCSI storage and OSD systems due to the use of a client cache.

Because these machines share the switch with other computers in our laboratory, the traffic of the switch can impact the performance of this system. The original read performance of ENHANCED is somewhat better than OSD-NOSEC and OSD-CAPKEY due to different traffic of the switch. Thus we re-measure the performance of these three systems by disconnecting all the other computers and get the result shown in Fig. 3(a).

Fig. 3(b) shows the performance of reading a block at a time. The overall performance is better than per character read performance as shown in Fig. 3(a). The iSCSI storage and NFS improve a lot due to lower overhead when reading

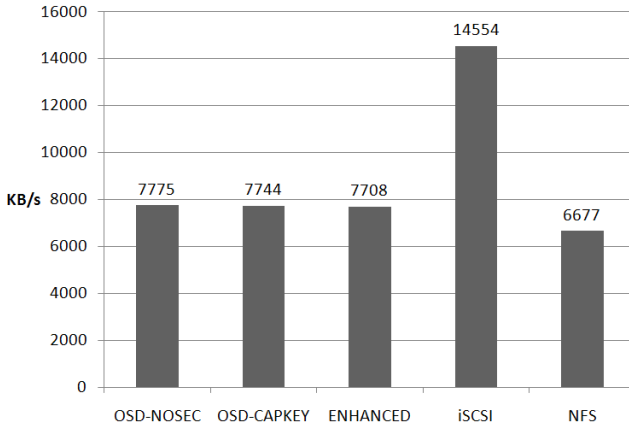


Fig. 4. Performance when rewriting an existing file

a block at a time. An entire block now contains valid data and traffics between client and server are reduced. The improvement in OSD systems is limited by the overhead to process an OSD command.

Fig. 4 shows the performance result of rewriting an existing file. The performance of our proposed system is still comparable to that of the OSD T10 standard. The rewriting performance of the OSD system is not as good as that of iSCSI storage because it needs to copy the data from storage server to a client's local buffer and then send back the rewritten data. The performance degradation is due to poorer read performance of the OSD system. Another reason is that there are two file systems in the OSD systems, OSD file system for initiator and ext3 file system for target. We have to use ext3 file system in the target to emulate an OSD storage device because currently no object-based disks are manufactured and available. We expect the performance of rewrite to be better if the underlying storage medium is real object-based disks. Another method to improve rewrite performance is to design a REWRITE command, which is a new command not defined in OSD T10 standard. Initiator can issue this new command with appropriate data and let the storage server process the rewriting process. In this way, the data needs not to be copied to application clients and thus the rewriting performance can be improved. This is left as our future implementation.

5 Conclusion and Future Work

The OSD system is a new architecture for storage subsystem. It offloads some tasks to storage devices to improve security management, scalability, and even performance. However, the security model of OSD T10 standard can be disabled by altering specific field in the capability. In this paper we point out a problematic security issue and propose to resolve the problem with encryption. The encryption process can be simplified in the binary field by using McEliece based

cryptosystem. The overhead of our OSD system is almost zero while offering an enhanced security.

We have also studied storage subsystem performance under five different environments using Bonnie++ benchmark suit. In general the CPU usage in the client side of OSD systems can be reduced significantly. We also find that our OSD system performs much better than the others when writing a file because a client's OSD file system dose not need to manage disk space. An application client can issue another command right after receiving an acknowledgement response from storage device server without waiting for it to execute the command. The read and rewrite performances are not as good as write performance due to the overhead of the file system used to emulate object-based disk in the storage server. The degradation is expected to be improved significantly when real object-based disk becomes available in the future.

The first task to do in the future is to improve the rewrite performance. We want to build an OSD system with performance comparable to iSCSI storage, thus we first need to improve the performance when rewriting a file. The reason that rewrite performance is not as good as write performance is that data needs to be read from the server to the client side, and the data is then modified and sent back to the server. We propose to implement a new command REWRITE to be used to eliminate this unnecessary data read from a storage server to an application client. The new data is sent to the server and the modification is done in the server side directly.

The ultimate goal is to design a real object-based disk. Currently the OSD server emulates object-based storage disk by utilizing the Linux ext3 file system to manage the disk space of the server and its overhead is very large. If we can design an intelligent disk which can manage disk space by itself, the read performance improvement will be significant. And this will take the storage technique to another era.

References

1. Mesnier, M., Ganger, G.R., Reidel, E.: Object-Based Storage. *IEEE Communications Magazine* 41(8), 84-90 (2003)
2. Du, D., He, D., Hong, C., Jeong, J., Kher, V., Kim, Y., Lu, Y., Raghuvver, A., Sharafkandi, S.: Experiences Building an Object-Based Storage System based on the OSD T-10 Standard. In: *MSST 2006* (2006)
3. SCSI Object-Based Storage Device Commands - 2(OSD-2), Project T10/1731-D, revision 0 (2004)
4. IBM Object Store
<http://www.haifa.il.ibm.com/projects/storage/objectstore/index.html/>
5. Azagury, A., et al.: Towards an Object Store. In: *MSS 2003* (2003)
6. Intel iSCSI/OSD reference implementation
<http://sourceforge.net/projects/intel-iscsi/>
7. Satran, J., et al.: Draft-ietf-ips-iscsi-20 (2003)
8. Meth, K.Z., Santran, J.: Design of the iSCSI Protocol. In: *MSS 2003* (2003)
9. SCSI Architecture Model - 4(SAM-4), Project T10/1683-D, revision 5(2006)

10. Factor, M., Nagle, D., Naor, D., Reidel, E., Satran, J.: The OSD security protocol. In: Proceedings of 3rd International IEEE Security in Storage Workshop (2005)
11. The Keyed-Hash Message Authentication Code. Federal Information Processing Standards Publication 198 (2006)
12. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. JPL DSN Progress Report, 42-44, pp. 114–116 (1978)
13. Li, Y.-X., Li, D.-X., Wu, C.-K.: How to Generate a Random Nonsingular Matrix in McEliece's Public-Key Cryptosystem. Singapore ICCS/ISITA, vol.1, pp. 268–269 (1992)
14. Preneel, B., Bosselaers, A., Govaerts, R., Vandewalle, J.: A Software Implementation of the McEliece Public-Key Cryptosystem. In: Proceedings of the 13th Symposium on Information Theory in the Benelux, Werkgemeenschap voor Informatie- en Communicatietheorie, pp. 119–126 (1992)
15. Bonnie++ benchmark suit, <http://www.coker.com.au/bonnie++/>
16. Linux-iSCSI Project, <http://linux-iscsi.sourceforge.net>