SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •
Special Focus

June 2011 Vol. 54 No. 6: 1119–1128
doi: 10.1007/s11432-011-4259-y

Corslet: A shared storage system keeping your data private

XUE Wei[1,2*], SHU JiWu[1,2], LIU Yang[1] & XUE Mao[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084*, China;*
[2]*Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing* 100084*, China*

Received September 16, 2010; accepted March 8, 2011

Abstract With the exponential growth of digital data, it is becoming more and more popular to store data in shared distributed storage systems inside the same organization. In such shared distributed storage systems, an ordinary user usually does not have the control permission over the whole system, and thus cannot secure data storage or data sharing of his own files. To solve this issue, this paper proposes a new system architecture to secure file storing and sharing efficiently over untrusted shared storage and network environments. Based on this architecture, this paper designs and implements a stackable secure storage system called Corslet. Corslet can run directly on deployed underlying storage systems without modification, while bringing end-to-end confidentiality and integrity as well as efficient access control for user data. For individual users, Corslet is easy to use, and does not require users to maintain or manage any keys on their client machines locally. The Bonnie++ and IOzone benchmark results show that the throughput of Corslet over NFS can achieve more than 90% of native NFS throughput in most tests, proving that Corslet can provide enhanced security for user data while maintaining acceptable performance.

Keywords storage security, access control, key management, shared file system

Citation Xue W, Shu J W, Liu Y, et al. Corslet: A shared storage system keeping your data private. Sci China Inf Sci, 2011, 54: 1119–1128, doi: 10.1007/s11432-011-4259-y

1 Introduction

With the exponential growth of digital data, modern storage systems are increasingly networked and distributed in both organization and operation [1–6]. Many of these systems, however, pay insufficient attention to data security, and merely rely on untrusted remote storage servers for data integrity and access control. Such practice cannot adequately protect sensitive personal and business data, such as medical records.

In practice, a variety of threats exist in the computational environments where our data reside [7]. Existing storage systems usually store and distribute data in plaintext, without sufficient integrity guarantee, user identification or access control mechanism, leaving data easy for snooping and tampering. Even more remarkable, the issue of data security has intensified due to the popularity of large-scale shared storage systems, and data leakage accidents have been reported more frequently [8].

*Corresponding author (email: xuewei@tsinghua.edu.cn)

Encrypt-on-disk [9, 10] (which is against encrypt-on-wire) is the leading technique used to secure data. However, most storage systems require users' trust in data servers and system administrators, which may not be accepted in practical applications, for the following reasons:

First, a storage system may be shared among different parties, and none of them has full control on the shared system; in other words, they have lost control on the physical resources. In such scenarios, we cannot assume the system administrators who have the highest privilege are always neutral, benevolent, and free of operational mistakes. Second, even if the sharing of storage resources happens inside the same organization or company which possesses its own storage system, different internal departments would wish to guarantee the confidentiality and integrity of their private data while separating access control groups. Third, even if users can trust administrators and data servers of shared storage systems, users' data in plaintext can be still at risk once the storage systems are infiltrated.

In this paper, we propose a novel system architecture for secure file systems. With this architecture, the ordinary users can secure file storing and sharing efficiently under untrusted multi-party shared storage and network environments which are out of their control. We then design and implement Corslet, a stackable secure storage system based on this architecture. Corslet is designed to run on top of existing file systems without modification to provide with extended end-to-end security and efficient access control, which are independent of those file systems and their administrators.

The rest of the paper is organized as follows. Section 2 describes the design criteria and assumptions of our secure storage system. Section 3 presents the key mechanisms and design of the system. Section 4 describes the system implementation. Section 5 evaluates the performance. We discuss related work in section 6 and conclude in section 7.

2 Design criteria and assumptions

2.1 Design criteria

The design criteria of our secure storage system are listed below.

Independent of underlying file systems. Corslet must provide data security and secure file sharing over existing file systems while leaving them unmodified. This makes Corslet have good portability to different types of storage systems, and use widely like securing removable storage devices.

File sharing and access controls. Corslet must enable users to share files easily, efficiently and safely with other users of the system. That means it should offer fine-grained sharing and access controls at file level, and not rely on authentication mechanism of underlying file systems.

End-to-end confidentiality and integrity. Only those who are granted permissions can access the content of data; other users without proper permissions, even the administrators of underlying system and those who can access the physical storage devices, should not get the information of data. Unauthorized modifications to file data should also be detected before authorized users see tampered data, to ensure the data passed to users are correct.

Key management. A delicate key management mechanism is critical to a secure storage system. Corslet should not require users to remember, store or manage any keys to access files locally after they log in the system.

Lazy revocation. Corslet will adopt lazy revocation [11] to amortize the enormous cost of immediately re-encrypting all affected files after revoking users that would otherwise jeopardize the system performance. This is a tradeoff between performance and security, but it would not weaken the security of the whole system, because the revoked users cannot read updated parts of files or modify files without being detected.

Performance. Corslet will exploit methods to reduce such penalty and should have comparable performance to its underlying file systems. In addition, Corslet must minimize storage space and bandwidth overheads resulting from encryption and integrity verification.

2.2 Threat model

In Corslet, there are different components such as underlying file storage servers, trust domain server and clients with various users. The threat model is discussed below.

Untrusted underlying file storage servers. In Corslet, we do not trust underlying file servers for keeping data confidentiality and integrity. We assume that these file servers can leak information to adversaries, tamper or delete data stored on them, or even deceive authenticated users with fake data.

Trusted trust domain server. In Corslet, we introduce the trusted trust domain server (TDS) to enforce access control and part of key distribution. This TDS plays as the global trusted entrance to data. The security of Corslet is dependent on it, so it has to be deployed safely.

Trusted file owners and untrusted readers and writers. A file owner is responsible to setting up and modifying the access permission of its file, so it is trusted. Meanwhile, readers and writers may attempt to get or tamper those data which they are not allowed to access, and thus are untrusted.

3 Design

3.1 Overview

Corslet employs trusted TDS to provide access control and key grant. Owing to the TDS, Corslet can carry out all secure operations with symmetric-key cryptography in lieu of its asymmetric counterpart, and users do not need to maintain any keys after they log into Corslet.

Corslet stores security control information associated with the encrypted data in the form of normal file in the underlying file system. We refer to this file which keeps information of access control, confidentiality, and integrity of data as security control file (or sc-file), and the encrypted data file as data file (or d-file). The sc-file can be divided into three different parts: the access control block (ACB) which contains critical information relevant to access control (like ACL) and is covered by HMAC, the root hash list which stores root hash values of hash trees composed of file block hash, and the encryption keys to the data file. A user has to read the sc-file corresponding to the d-file from the underlying file system, sends the ACB part of this file together with his operation request (read, write, etc.) and his own identifier to the TDS for authenticating and have the system authorize the accesses, and then obtain the relevant keys to access the content of d-file.

3.2 Trust systems

In Corslet, the trust flow stems from the trusted TDS. When creating a file, the file owner specifies access control information of this file and then authorizes the TDS to execute access control, as shown in Figure 1. Every user contacts the TDS for access grant to a file, without the need of communicating the file owner or sharing any common information with other users, precluding the influence on file access caused by off-line owners or users. This distinctive trust system with the trusted TDS distinguishes Corslet from other secure storage systems.

The trust system of Corslet is composed of two parts: the user identification and the global trust root (i.e., TDS). To prevent non-privileged users' data from being snooped by privileged users such as administrators, Corslet supplies its own user identification mechanism. We exploit the certifications granted by Public Key Infrastructure (PKI [12]) to identify users (as well as the TDS).

The global trust root is offered by the trusted TDS. As the global trusted entrance to Corslet system, the TDS is in charge of guaranteeing confidentiality and integrity of data and enforcing access control. The security of the whole Corslet system relies on the TDS; that is also to say that if this server is kept safe, the whole Corslet is guaranteed safe. The TDS does not store any information of any user, nor maintain any login status of any user, except the simple sessions between the server and users, which could be terminated at any time and created easily. The TDS does not distribute keys itself, but decides who can request those keys by executing some simple symmetric encryption and decryption computation with its two secret keys. This simple logic and structure can reduce the possibility that the TDS becomes

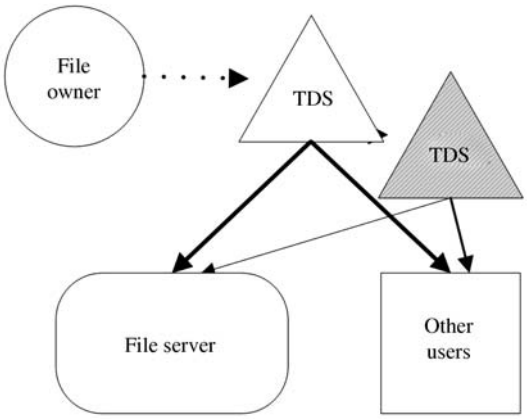

Figure 1 Trust system of Corslet. The solid line represents the control relationship, and the dashed line represents user authorization. The arrow points out the direction. The shaded TDS represents another trust domain.

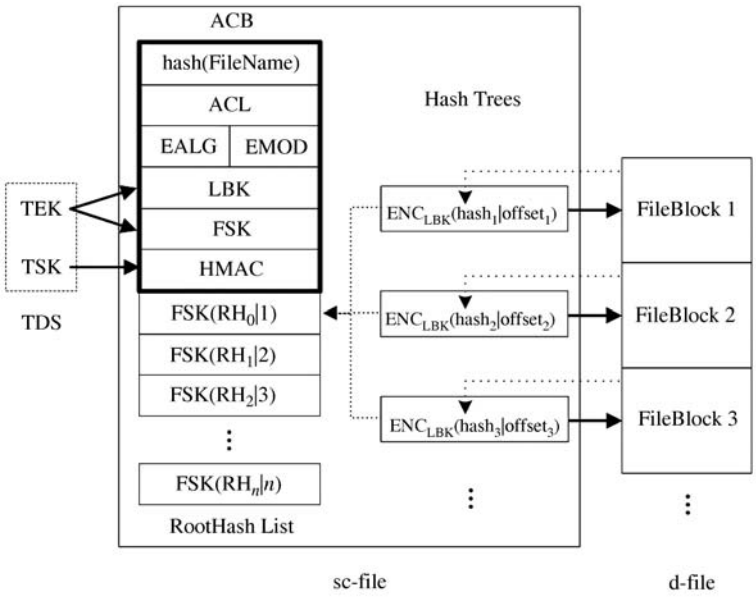

Figure 2 The hierarchical organization of Corslet keys.

the bottleneck of the whole system, and can also prevent the TDS from being the system's single point of failure, since the simple service it supplies can be switched to another server easily and instantly.

3.3 Hierarchical symmetric key organization

In shared secure file systems, there are two key problems of key management: reducing the number of keys to be maintained and dealing with key update happening in permission revocation, which are critical to making the system easy and safe to use.

Corslet organizes symmetric encryption keys into the following three levels, as depicted in Figure 2.

i) Data file keys. The first level is d-file keys. To deal with big files safely and efficiently, Corslet encrypts data files in blocks. We refer to this block as file block to differentiate it from the underlying file system block, and each file block is encrypted by a unique symmetric key, called file block key (FBK). Using different FBKs for different file blocks can reduce vulnerability to known plaintext and known ciphertext attacks, but leads to large numbers of keys. To improve the storage efficiency of FBKs, Corslet concatenates the hash value and the offset in the data file of a file block as the file block's own encryption key. We call this self-encryption since the material used for generating the FBK for each file block is partly from the hash value of that file block: FBK_i=hash(file block_i)|| offset_i.

ii) Security control file keys. The keys kept in the sc-file are the second level. These keys include a lockbox key (LBK) and a file signature key (FSK). A lockbox, originating from Cepheus [11], holds the FBKs for all the file blocks of a file and is encrypted by the symmetric LBK. Only those who get the LBK can decrypt the lockbox, and then get FBKs to decrypt file blocks further. The FSK is used to encrypt and decrypt the roots of the hash trees constituted by file block hash values, and Corslet differentiates read and write permissions by this symmetric key and the FSK will only be granted to writers to allow them to update file blocks and re-sign the roots of hash trees. Note that the FSK here is different from those in [9, 13]: The former is symmetric, while the latter is asymmetric.

iii) TDS keys. On the top level lie the TDS keys. There are merely two symmetric keys maintained by TDS. One is the TDS Encryption Key (TEK) and the other is the TDS Signature Key (TSK). The former is used to encrypt the LBK and the FSK of an sc-file to implement access control and to differentiate between read and write operations, while the latter is used to compute HMAC of the ACB in the sc-file to ensure its integrity. The TDS guarantees the privacy of these two secret keys, and should not divulge them to any other at any time. This could be achieved easily by some hardware-assisted measures in practice.

3.4 Integrity protection

Corslet computes hash for each file block of a file to ensure its content integrity. Then these hash values are further organized into several Merkle Hash Trees [14] to efficiently guarantee their own integrity. Different from CRUST [15], the Merkle Tree here stores file block hash values both in leaf nodes and in internal nodes (Figures 2 and 3). Each node contains two parts: the file block hash value of that node, which guarantees the integrity of the file block, named bHash, and the hash value computed on the concatenation of file block hash values in all children of that node, which guarantees the integrity of the values stored in the children nodes of that node, named cHash. Then the root of this Merkle Tree is encrypted by FSK stored in the ACB of sc-file to authenticate the integrity of the whole file. This structure can reduce the modification path of the Merkle Tree when a node is modified: Only the modified node and those above this node in the path are re-computated.

As a side effect of generating the FBKs from the corresponding file block hash values, the Merkle Tree also guarantees the integrity of FBKs, which we believe is valued by few existing systems. In addition, since Corslet first computes the hash value of a data block and then encrypts this data block with the FBK generated from that hash, these file block hash values guarantee the integrity of the real content of the data block, not that of the data block in encrypted form. We believe this is very necessary, as otherwise even if a user can verify the hash of the encrypted data blocks, he cannot make sure if the data he reads are what he wants, given the fact that the decryption keys may have been tampered due to lack of integrity protection, leaving the content unintelligible after decryption.

With this Merkle Tree structure, a legal modification to the content of the file block should include the following steps: i) modify the file block; ii) re-compute the hash of that file block; iii) modify the relevant nodes in the Merkle Tree; iv) re-sign the root node of the Merkle Tree with FSK.

Before any access to a file block, Corslet will first check the integrity of the Merkle Tree (thus guaranteeing the integrity of both the file block hash and the FBK) by the following steps: i) re-compute the hash value of the file block and compare it with the original bHash value stored in the current node corresponding to the file block; ii) locate the path from the father of current node to the root node; iii) with the father of current node as the start node, follow the path to the root node, re-compute the hash value of the concatenation value of all the children and compare it with the cHash value stored in the node.

According to the integrity validate steps, any illegal interpolation to the original data, whether to the file block or to the Merkle Tree node (thus the hash and the FBK of the file block), would be detected at once.

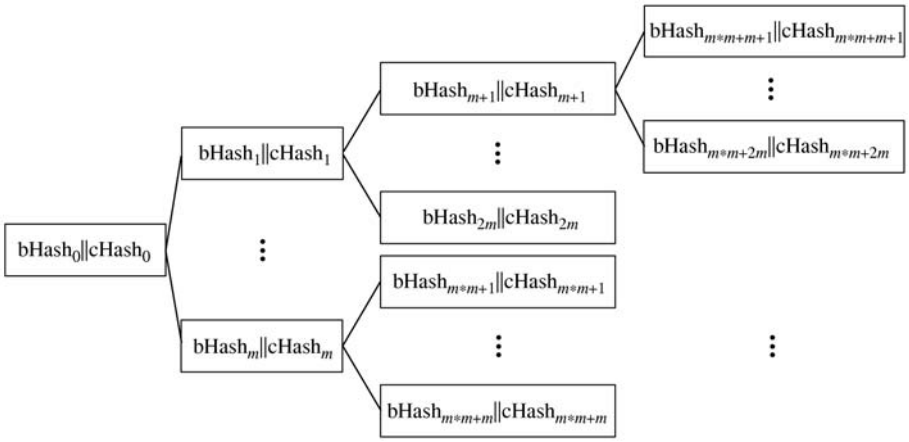

Figure 3 Merkle Tree of Corslet.

3.5 Permission revocation

User permission revocation is a frequent operation in large distributed file systems [9]. Traditionally in common systems, the permission revocation has to rely on servers, needing heavy trust on the servers. In encrypt-on-disk systems, permission revocation introduces extra overhead by re-encrypting the related files and re-distributing new file keys to survival users to prevent further illegal file access by revoked users. This simple instant re-encryption would incur file access blocking and system-wide performance penalty, which may be intolerable when considering the large numbers of users and files in a distributed shared file system.

To reduce the overhead of permission revocation while preserving the security of the system, Corslet adopts the lazy revocation [11] that postpones the re-encryption to each affected file until the file is updated for the first time after revocation, while the security information should still be renewed immediately once revocation happens. Lazy revocation is less expensive, but may be more complicated because there would be multiple versions of keys for a file to be maintained. In Corslet, however, this process will be simple attributing to its hierarchical symmetric key organization. When permission revocation occurs, TDS will generate a new LBK and a new FSK into the ACB of the sc-file, and then re-encrypt all the FBKs and root hash virtual linked list of this file; besides, the HMAC of ACB also needs to be re-computed. Note that in this step, the re-encrypted data size is much smaller than the whole file. Then when parts of this file are modified, the relevant file blocks would be re-encrypted with the newer FBK automatically as the hash values of these file blocks will have been changed; thus this process has no difference with common file block update in Corslet. This scheme does not need any complicated mechanism or extra space to store and retrieve the historical keys or historical statuses that are required in other lazy systems [9, 15], saving considerable storage and computation cost which is needed when comparing to different versions of the keys.

4 Implementation

4.1 Overview

Corslet was implemented on Linux using the FUSE (Filesystem in Userspace) framework [16] and OpenSSL [17]. OpenSSL is well-known for its good implementation and complete interfaces, and thus is widely used in many systems. In Corslet we use SHA-1 [18] as the cryptographic hash function, HMAC [19] based on SHA-1 as the MAC algorithm and AES-256 [20] as the block cipher by default. These can all be configured by users. In addition, OpenSSL also provides good implementation of PKI, which is used in Corslet to authenticate system roles and set up security channel between TDS and clients.

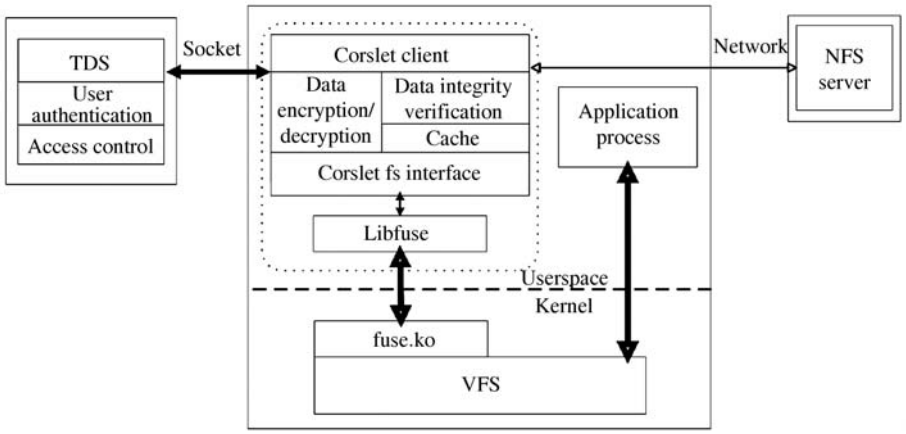

Figure 4 The architecture of Corslet.

4.2 Architecture

Corslet system consists of three parts: TDS, Corslet clients, and the underlying storage system, as shown in Figure 4. TDS plays as the global trusted root of the whole system.

Corslet offers the POSIX file system interfaces to the users. Most cryptographic computation and data integrity verification are carried out on Corslet clients, and are transparent to users. Applications can directly run on Corslet without any awareness of extra security operations, while enjoying end-to-end security protection. It would also be convenient for users to use Corslet since our system does not require users to keep or manage any keys locally. When a user wants to access a file, he has to contact with the TDS for keys, and the communication between clients and TDS is done by SSL/TSL [21]. To deal with big files safely and efficiently, Corslet encrypts and decrypts data files in file blocks. The size of the file block, typically ranging from 4 to 64 KB, can be configured by users when mounting Corslet to tune for the performance.

The underlying storage system is regarded by Corslet as infrastructure which provides reliable storage service. Both remote and local file systems can work well with Corslet as long as they have the VFS structure, though we mainly focus on remote shared storage systems.

We also developed an ACL utility which allows owners to administrate the access control permission of their files.

5 Performance evaluation

We evaluated Corslet by performing a series of tests to assess its performance and overhead.

We first used Bonnie++ 1.03e to compare the performance of Corslet over NFS to native NFS. We used three Sun SunFire V20z servers, each had two 1.8 GHz AMD CPUs and 4 GB memory. The first one server was set up as TDS, the second one was NFS server, the last one was set up as the NFS client, as well as the Corslet client (mounted on NFS mounting point). The servers were connected with GbNet. The operating system was Debian Linux (version 2.6.30). We used the default 64 KB as the size of file block, 256-bit AES in CFB mode for data encryption, SHA-1 as the cryptographic hash function, and HMAC based on SHA-1 as the MAC algorithm. We ran the 8 GB file Bonnie++ test on the NFS mounting point, and then the Corslet mounting point. The results are shown in Figure 5.

From the results we can observe that the throughput of Corslet over NFS can achieve above 90% of that of raw NFS except "write per char" test. Corslet would verify the integrity of file blocks and Merkle Tree before any access. Even to read/write a single byte, the whole file block, whose size is 64 KB, would be checked (recalculate its hash and compare it with the hash value stored in the FBK). So the integrity verification brought a 30% performance drop in "write per char" test. But in "read per char" test, the performance decrease brought by integrity verification was much smaller due to the longer access path of NFS.

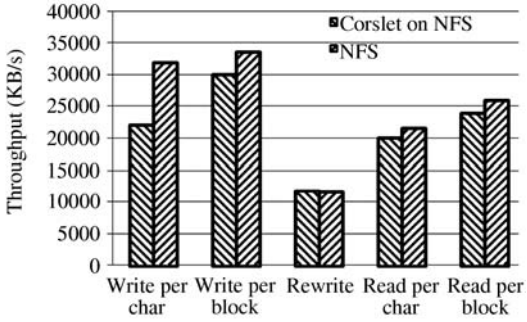

Figure 5 Bonnie++ results on one node (Corslet vs. NFS).

Table 1 Corslet permission operation cost

Operation	Time (ms)
Permission granting	1.86
Permission update	1.86
Revocation	21.74

We then evaluated the performance of the operations that modified file access permissions. These operations included granting a new user, revoking a user, and updating the permission of an existing user. The test set included a TDS server and a Corslet client machine, whose configurations were the same as those in the above test. We first granted r− permissions to 1000 separate users, updated those permissions into rw−, and then revoked all these permissions on a 1 GB file. The operation time for each user was measured from the ACL utility developed by us on the client and the times were averaged, as shown in Table 1.

From Table 1 we can conclude that the cost of permission operations is acceptable, and users will rarely be aware of the extra overhead of cryptographic computation and network communication.

Finally, we used IOzone to test the performance of Corslet over a cluster. This test was set up on a Dell PowerEdge M605 cluster with seven nodes, including one TDS server, one NFS (NFSv4) server and five client machines, which were connected by GbNet. The TDS server and the NFS server both had two 2.4 GHz quad-core CPUs and 16 GB memory, while each client machine had two 2.4 GHz quad-core CPUs and 8 GB memory. The operating systems of the servers and the clients were Fedora Core 10 Linux (kernel version 2.6.32).

We still chose AES-256 as the default cipher, SHA-1 as the cryptographic hash function, and SHA-1 based HMAC as the MAC algorithm. To be close to practical scenarios, we adopted 64 KB as the default size of file block, and CFB as the tested encryption mode. In fact, the safer CFB mode could only provide mediocre performance; thus our test results would be the lower bound of Corslet's real performance. To eliminate the effect of file system cache, we set the tested file size at 16 GB (which was twice as large as the memory size). The results were the aggregated throughput of five nodes, as shown in Figure 6.

From the results we can observe that the aggregated throughput of Corslet over NFS can achieve above 90% of that of raw NFS. Specifically, the percentages for sequential write, sequential re-write, sequential read and sequential re-read are 95.78%, 95.62%, 90.45% and 90.71%, respectively. Furthermore, we can also observe that the overall utilization of the eight cores of two CPU was less than 1/16, which will not interfere the applications running on the clients much.

6 Related work

CFS [22] is among the earliest work on encrypt-on-disk file systems. It is a virtual cryptographic file system, and pathnames and file data are encrypted before written to disks. CFS uses a single key to encrypt the whole file directory; access control is implemented by granting the key, and file sharing will require exporting the key to others. These determine that CFS can only allow coarse file sharing on a single machine, without read-write differentiation.

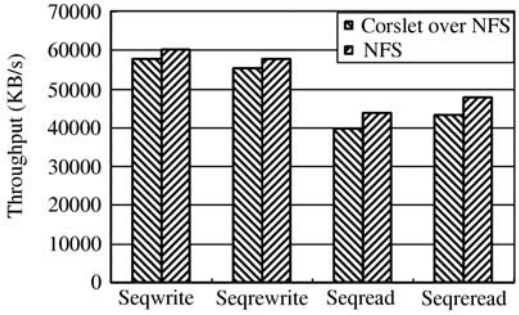

Figure 6 IOzone results on a five-node cluster.

Cryptfs [23], Extended Cryptographic File System (ECFS) [24], Cepheus [11], and TCFS [25] are famous variants of CFS. Cryptfs associates symmetric keys with file groups, allowing group file sharing. Cepheus introduces lockboxes for group management, trusts a centralized key server that stores information of group members to authenticate users and relys on file server to implement access control; it is also the first system to employ lazy revocation. eCryptfs [26] is a kernel-native stackable cryptographic file system derived from Cryptfs. It is designed to enforce data confidentiality on secondary storage devices of a single host, and thus cannot be applied to support file sharing due to lack of efficient key management and access control.

CryptosFS [27] replaces access control mechanism with public-key encryption, and trusts the file servers to verify user access. A user has to get the asymmetric key to access the enciphered file, and further the symmetric key to decrypt that file to get the data. This is a kind of out-band manner. OceanStore [28] and FARSITE [29] provide some security for large distributed file systems, while availability and fault-tolerance are the focus of such systems.

SNAD [30] also employs lockboxes, but its access control has to depend on remote file servers, thus needing strong trust in the servers. SiRiUS [13] is designed to work over insecure file systems as a cryptographic storage layer to supply storage security, and needs secure public key servers. It employs in-band key distribution and instant revocation. Although claimed to support any existing file systems, SiRiUS was merely implemented on NFS. Plutus [9] offers cryptographic group sharing with lazy revocation, random access and file name encryption. Plutus, SiRiUS and SNAD provide end-to-end secure file sharing on untrusted file servers, at a cost of performance. Plutus and SNAD have to rely on file servers for access control. All of the three systems employ public-key encryption.

CRUST [15] is a stackable secure file system with complete symmetric encryption and in-band key distribution. However, CRUST relies on some global shared data structures to distribute keys and to retrieve previous states for revocation, and has to maintain several keys in local client machines, while Corslet does not need to do any of these.

7 Conclusions

This paper has introduced a novel system architecture for secure file sharing. We design and implement Corslet, a secure and efficient stackable file system based on the above architecture. Corslet can guarantee the confidentiality and integrity of user data over untrusted shared storage, while saving users from complexity of key management. For users' convenience, Corslet allows users to choose which files are to be stored in encrypted form and which are to be kept in plaintext.

By measuring its performance on several tests and benchmarks, we show that Corslet has acceptable overhead while providing strong security. In addition, almost all cryptography of Corslet is executed on local clients, while the trusted TDS merely performs simple symmetric encryption/decryption computation and stores a small amount of information or status of users.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (Grant No. 60925006), the National Basic Research Program of China (Grant No. 2010CB951903), the National High Technology Research & Development Program of China (Grant No. 2009AA01A403) and Intel.

References

1 Sandberg R, Goldberg D, Kleiman S, et al. Design and implementation of the SUN network filesystem. In: Proceedings of the Summer USENIX Conference, Portland, USA, 1985. 119–130
2 Callaghan B, Pawlowski B, Staubach P. NFS version protocol specification. RFC 1813, 1995
3 Braam P J. The Lustre storage architecture. http://www.lustre.org/documentation.html
4 Braam P J. The Lustre storage architecture. Cluster File Systems, Inc., Aug. 2004. http://www.lustre.org/documentation.html
5 Amazon.com. Amazon simple storage service (Amazon S3). http://aws.amazon.com/s3
6 Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system. In: Proceedings of OSDI, Seattle, USA, 2006. 22
7 Hasan R, Myagmar S, Lee A J, et al. Toward a threat model for storage systems. In: Proceedings of StorageSS, Fairfax, USA, 2005. 94–102
8 Data Breach Investigation Report, Verizon, 2010. http://www.verizonbusiness.com/resources/reports/rp-2010-data-breach-report-en-xg.pdf
9 Kallahalla M, Riedel E, Swaminathan R, et al. Plutus-scalable secure file sharing on untrusted storage. In: Proceedings of the 2nd USENIX File and Storage Technologies, San Francisco, USA, 2003
10 Riedel E, Kallahalla M, Swaminathan R. A framework for evaluating storage system security. In: Proceedings of FAST, Monterey, USA, 2002. 15–30
11 Fu K. Group sharing and random access in cryptographic storage file systems. Dissertation for Master's Degree. Cambridge: Massachusetts Institute of Technology, 1999
12 PKI. http://datatracker.ietf.org/wg/pkix/charter/
13 Goh E, Shacham H, Modadugu N, et al. SiRiUS: Securing remote untrusted storage. In: Proceedings of the 10th Network and Distributed Systems Security Symposium, San Diego, USA, 2003. 131–145
14 Merkle R C. A digital signature based on a conventional encryption function. In: Proceedings of CRYPTO'87, Santa Barbara, USA, 1987. 369–378
15 Geron E, Wool A. CRUST: Cryptographic remote untrusted storage without public keys. In: Proceedings of the 4th International IEEE Security in Storage Workshop, San Diego, USA, 2007. 357–377
16 Szeredi M. Filesystem in userspace. http://fuse.sourceforge.net
17 OpenSSL Project. http://www.openssl.org/
18 NIST. Secure hash standard. Federal Information Processing Standards, FIPS PUB 180-2, 2004
19 Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-hashing for message authentication. RFC 2104, 1997
20 NIST. Advanced encryption standard. Federal Information Processing Standards, FIPS PUB 197, 2001
21 SSL/TLS. http://tools.ietf.org/html/rfc5246
22 Blaze M. A cryptographic file system for Unix. In: Proceedings of the ACM Conference on Computer and Communications Security, Fairfax, USA, 1993. 9–16
23 Zadok E, Badulescu I, Shender A. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98. 1998
24 Bindel D, Chew M, Wells C. Extended cryptographic file system. Unpublished manuscript, 1999
25 Cattaneo G, Catuogno L, Sorbo A D, et al. The design and implementation of a transparent cryptographic filesystem for Unix. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, Berkeley, USA, 2001. 199-212
26 Halcrow M. eCryptfs: A stacked cryptographic filesystem. Linux J, 2007, 156: 2
27 O'Shanahan D P. CryptosFS: Fast cryptographic secure NFS. Dissertation for Master's Degree. Dublin: University of Dublin, 2000
28 Kubiatowicz J, Bindel D, Chen Y, et al. Oceanstore: An architecture for global-scale persistent storage. In: Proceedings of ASPLOS, Cambridge, USA, 2000. 190–201
29 Adya A, Bolosky W, Castro M, et al. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of OSDI, Boston, USA, 2002. 1–14
30 Miller E, Long D, Freeman W, et al. Strong security for network-attached storage. In: Proceedings of FAST, Monterey, USA, 2002. 1–13