J. Parallel Distrib. Comput. 74 (2014) 2872–2883

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Shield: A stackable secure storage system for file sharing in public storage

Jiwu Shu *, Zhirong Shen, Wei Xue

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China

HIGHLIGHTS

- We propose a new system architecture for secure file sharing in cloud scenario.
- We implement a stackable secure storage system named Shield.
- A hierarchical key organization is designed for convenient keys management.
- Shield adopts lazy revocation to accelerate the revocation process.
- Shield supports concurrent write access by employing a virtual linked list.

ARTICLE INFO

Article history:
Received 23 September 2013
Received in revised form
8 June 2014
Accepted 10 June 2014
Available online 19 June 2014

Keywords:
Storage system
Cryptographic controls
Keys management
Proxy server
Secure sharing
Permission revocation
Concurrent writes

ABSTRACT

With the increasing amount of personal data stored in public storage, users are losing control of their physical data, putting their data information at risk of theft or being compromised. Traditional secure storage systems either require users to completely trust the storage provider or impose the considerable burden of managing files on file owners; such systems are inapplicable in the practical cloud environment. This paper addresses these challenging problems by proposing a new secure system architecture and implementing a stackable secure storage system named Shield, in which a proxy server is introduced to be in charge of authentication and access control. We propose a new variant of the Merkle Hash Tree to support efficient integrity checking and file content update; further, we have designed a hierarchical key organization to achieve convenient keys management and efficient permission revocation. Shield supports concurrent write access by employing a virtual linked list; it also provides secure file sharing without any modification to the underlying file systems. A series of evaluations over various real benchmarks show that Shield causes about 7%∼13% performance degradation when compared with eCryptfs but provides enhanced security for user's data.

1. Introduction

As a new kind of online storage over network, cloud storage delivers elastic storage service and provides virtually unlimited storage capacity without requiring users to perform complex system configurations or buy expensive storage devices. Owing to its convenience and economy, data owners are willing to concentrate their data to the cloud.

Although cloud storage dramatically improves the efficiency of data management, data owners have to sacrifice physical control of their data by handing it over to the cloud server, which may put the data information at risk of theft or being compromised caused by unauthorized access. Derived from this worry, the research reports on data leaks have increasingly emerged in recent years, causing public concern about the security when their sensitive data are stored in the public storage. The report released by IDC [8] further points out that data security have been treated as a top priority in cloud computing.

Many secure storage systems [15,17,13,40,3,14,16,19] have been proposed for protecting data security by using several key technologies such as encrypt-on-disk [17], but most of them are mostly based on the outdated models of either requiring the cloud server to be completely trusted [28,25] to execute access control and key distribution, or needing file owners to actively manage security themselves [17,26] (i.e. only trusting themselves and deal with users' access requests by themselves). However, the two

* Corresponding author at: Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.
E-mail addresses: shujw@tsinghua.edu.cn (J. Shu), zhirong.shen2601@gmail.com, czr10@mails.tsinghua.edu.cn (Z. Shen), xuewei@tsinghua.edu.cn (W. Xue).

http://dx.doi.org/10.1016/j.jpdc.2014.06.003
0743-7315/

extreme trust models, complete trust or no trust, either invite potential security threats or involve significant inconveniences. When the cloud server is given complete trust, users may worry about the status of their sensitive data once the cloud server does not behave neutrally or is fully controlled by hackers, and the cloud server may also fear to get into the risks of economic disputes and reputation losses, once user's data are leaked because of either sudden accidents or careless operations. In another case, if file owners are required to bear the burden of trusting themselves, they will be forced to be always online to handle access requests and execute access control, causing a considerable management burden. There are extra works that make some impressive efforts to alleviate the shortcomings of the above two architectures. For example, Castiglione et al. [5] try to move the management burden to users by utilizing secret sharing mechanism. They require any data access should receive the permissions from at least k users. However, it may also introduce considerable computation and management burden to users. Though another work [7] introduces a generic credential authority to manage keys and enforce access control. Users in this architecture are responsible for keys management for the version signature. This may cause considerable management burden once a user joins many groups. Meanwhile, it also introduces many public-key encryption computations, which are far more expensive than symmetric-key encryption.

To address these problems, we analyze the threat models of file sharing among multiple users in public storage, establish a new trust system, and propose a new system architecture in which *users can store files and share them efficiently under multi-party shared public storage and network environments*. Based on this architecture, we develop a stackable secure storage system named Shield, which strives to provide secure file sharing and free file owners from cumbersome keys management in the cloud scenario. To improve portability, Shield requires no modification to the file system and can be directly deployed on top of existing file systems to provide extended end-to-end security and efficient access control, both of which are independent of cloud storage systems or administrators.

In addition, to prevent the cloud server from accessing plaintext when encrypting/decrypting data, Shield migrates operations of data encryption/decryption and integrity checking to be performed at the client side. This change also benefits the scalability of the whole system. Moreover, Shield concentrates on the mechanisms that provide efficient permission revocation and support write concurrency when data files are shared among multiple users. Our contributions are as follows:

First, we propose a new architecture for secure file sharing that neither places complete trust over the cloud server nor imposes cumbersome management burdens on file owners. In this architecture, we use the *proxy server* (PS) to manage access control and distribute secret keys. To avoid the problem of the PS being a bottleneck, a PS-Group can be easily constructed to decrease the burden and trust over every individual PS.

Second, we develop a hierarchical key organization to lighten the complexity of keys management, design a variant of the *Merkle Hash Tree* (MHT) for integrity checking, adopt lazy revocation to improve the efficiency of revocation operations, and exploit the virtual hash linked list to support concurrent writing to a file.

Third, by employing some representative benchmarks, intensive tests are conducted to evaluate the performance of Shield. The final results show that Shield causes about 7%–13% performance degradation when compared with eCryptfs but provides enhanced security and a single PS can support more than 45,000 users' requests in one second.

The remainder of the paper continues as follows: We review the related work in Section 2, describe the design goals and assumptions in Section 3, and introduce the key techniques in Section 4. Section 5 describes the protocols for file reading, file writing, and permission revocation. Subsequently, we analyze the security of Shield in Section 6, discuss its implementation in Section 7, and evaluate its performance in Section 8. Finally, we conclude our work in Section 9.

2. Related work

CFS [3] is the earliest work of encrypt-on-disk file systems, which uses a single key to encrypt the whole file directory. As its variants, Cryptfs [40], Cepheus [32], and TCFS [6] are proposed later. Cryptfs associates symmetric keys and file groups, allowing group file sharing. Cepheus introduces a lockbox for group management and firstly proposes lazy revocation. TCFS designs transparent cryptographic file systems by integrating the encryption service with file systems. However, all of them miss the considerations of read–write differentiation and lack an efficient keys management mechanism.

To support wide file sharing service, NCryptfs [37] implemented at the kernel level supports multi-user sharing on the same machine. The Swallow [30] implements access control by executing encryption at the client level. However, Swallow neither offers file-sharing nor differentiates read–write operations. NASD [14] provides data security for network-attached storage. It keeps data in the form of plaintext and the security guarantee requires the participation of storage devices.

SFS [25] has to rely on the trusted server to enforce access control. It provides authentication for remote file systems, and the communication channel with the server is encrypted by a session key. CryptosFS [28] also trusts the storage servers to verify user's access and uses public-key encryption instead of existing access control mechanism in NFS to regulate user's access. Derived from CryptosFS, eCryptfs [16] is designed to enforce data confidentiality on secondary storage belonging to a single host, however, it cannot support file sharing because of the absence of keys management and access control.

With the system scales up, the data reliability and durability becomes important design criteria for large storage systems. OceanStore [20] and FARSITE [1] mainly focus on the availability and fault-tolerance while providing security for those distributed file systems. To protect long-term data, POTSHARDS [33] uses secret splitting and approximate pointers to secure data, which may only be cracked after decades, and SafeStore [19] spreads data across autonomous SSPs using informed hierarchical erasure coding to increase data durability.

The Secure Untrusted Data Repository (SUNDR) [24] relies on a storage server to execute access control while providing data confidentiality with per-file key encryption and file integrity with hash trees. PCFS [12] is a file system with proof-carrying authorization that provides access control with policy support by formal proof and capability. Maat [21] is designed for object-based storage and it uses extended capabilities, automatic revocation and secure delegation to secure distributed file systems. SNAD [26] employs a lockbox to protect integrity. However, neither Maat nor PCFS can provide on-disk security, leaving data exposed to adversaries. What is more, all of SUNDR [24], SNAD [26] and Maat [21] require new types of storage servers, while Shield does not demand a new infrastructure and allows users to manage their own access without relying on storage servers.

Besides disordering data by encryption, some representative works protect data security by forbidding the illegal access to the data. I3FS [29] is a file system with build-in integrity checking that uses cryptographic checksums to provide integrity validation for files. Kerberos [27] provides authentication service for clients in insecure network environments. Clients have to interact with AS (Authentication Server) and TGS (Ticket Granting Server) for authentication before applying for this service.

Table 1
Comparison between some related secure storage systems.

Systems	Encryption	Keys	Integrity	Lazy revocation	Concurrent write
FARSITE [1]	Block level	Asymmetric	MHT	No	No
CRUST [13]	Block level	Symmetric	MHT	Yes	No
Horus [23]	Block level	Asymmetric	–	No	No
SNAD [26]	Block level	Asymmetric	Block hash	No	No
SiRiUS [15]	File level	Asymmetric	File hash	No	No
Plutus [17]	Block level	Asymmetric	MHT	Yes	No
Shield	Block level	Symmetric	MHT	Yes	Yes

To securely share data with massive users through network, a versatile secure storage system should include various mechanisms, such as authentication, access control, keys management, integrity protection, and permission revocation. SiRiUS [15] can work over insecure file systems as a cryptographic storage layer to supply storage security. Plutus [17] offers cryptographic group sharing with lazy revocation, random access, and file name encryption. CRUST [13] is a stackable secure file system with completely symmetrical encryption and in-band key distribution. However, Plutus [17], SiRiUS [15] and CRUST [13] all require the file owner to bear the burden of dominating the access control and key distribution (although there are some differences in key distribution: Plutus needs users to get the file key from the file owner, while SiRiUS and CRUST require pre-sharing of some message of key materials among users before file access). Moreover, SiRiUS employs aggressive revocation which requires far more intensive computations than lazy revocation and CRUST relies on some global shared data structures to distribute keys and to retrieve previous states for revocation. Corslet [38] presents a secure storage system to protect data stored in untrusted cloud storage, but it does not take concurrent writing into consideration. As in Shield, the most recent work by Horus [23] introduces an independent key distribution server to generate the range keys for users to enforce the secure range access. Compared to studies by Horus, Shield supports concurrent writes and improves the efficiency of integrity checking (see Table 1).

In the past few years, access control for secure storage system received intensive attention. One interesting work is over-encryption [35,36] which performs two-level encryptions to regulate user's access, so that the re-encryption will be executed without being shipped to the data owner. However, over-encryption may cause the publishing of many public tokens and users in the setting of over-encryption are required to afford the keys management. The management burden will become large once a user is allowed to access many files owned by different owners. Shield frees users from complex keys management by designing the hierarchical keys management. Another new cryptographic primitive called Attribute-Based Encryption (ABE) [39] achieves the flexible access policy and eliminates the complexity of keys management for users. As a typical public-key encryption based on the expensive bilinear pairing operation, ABE may cause considerable performance penalty. Moreover, users in ABE usually possess a set of attribute keys. Once the attribute of a user changes, his access privilege will correspondingly alter. This requirement will make the authority to distribute new attribute keys and force the update of access policy based on the new attribute keys. This procedure is expensive and tedious. To provide a high performance, Shield temporarily adopts ACL to record user's access permission, which is easy to apply in real scenario.

3. System model and design goals

3.1. System model and threat model

In this paper, we mainly consider a cloud service of data sharing that consists of four kinds of entities as illustrated in Fig. 1, i.e., the cloud server, the proxy server group, many file owners, and many file users.

Trust Domain. To efficiently organize the massive number of users who are assigned various attributes in the cloud environment, Shield first defines the concept of *trust domain*, which actually specifies the granularity of file sharing and users in the same domain are treated to have the same security requirements. In fact, the actual use of this concept appears frequently in real life. For example, several departments inside the same organization may share the same storage system but need separate access control mechanisms and have different security requirements; thus, users and files belonging to the same department can be organized in the same trust domain.

The cloud server. On the basis of the following two reasons, Shield excludes the secret keys management from the responsibility of the cloud server. First, due to security concern, users will be uneasy for their privacy once the cloud server grasps both the ciphertext and secret keys. Second, according to the communication with the companies planning to provide public cloud service, they are also afraid of legal dispute and reputation degradation once user's data are either lost or leaked. They are more like to store ciphertext only rather than managing the secret keys with the ciphertext together.

Therefore, our model assumes the cloud server only performs two fundamental tasks: storing the files reliably and utilizing existing methods (e.g., Access Control List) to enforce access control to the ciphertext, where the former task indicates the files stored in cloud storage will not be maliciously deleted or altered while the latter ensures only authorized users are allowed to access the ciphertext.

The proxy server. In Shield, to avoid being always online to process users' access requests and manage a large number of keys, file owners can choose to delegate the execution of key distribution to a group of *proxy server* (PSs) that is in the same trust domain. In this paper, a PS is an independent third party that is responsible for processing users' access requests by distributing the corresponding secret keys according to their access permissions. Furthermore, to prevent a single PS from becoming a bottleneck in the whole system, multiple PSs can be organized as a "PS-Group" by employing a (n, k) secret-sharing mechanism, so that as long as no more than k PSs are invaded, the confidentiality of plaintext will be still under protection.

To lighten the burden on a PS, it is designed to be deployed independently without sharing any information with either the cloud server or the users, except for maintaining a simple session between the PS and the user. The simple design of the PS offers Shield good scalability and alleviates the possibility that the PS will become a bottleneck in the whole system; moreover, the simple service can be easily migrated to another PS immediately if an accident occurs.

The file owners and file users. The file owners are the entities who create the data files. Apart from the ordinary operations (e.g., reading and writing), they can also determine a user's access permission to their files. As the parties who wish to access the file content, file users, including file readers and file writers, should faithfully obey the granted access rights.

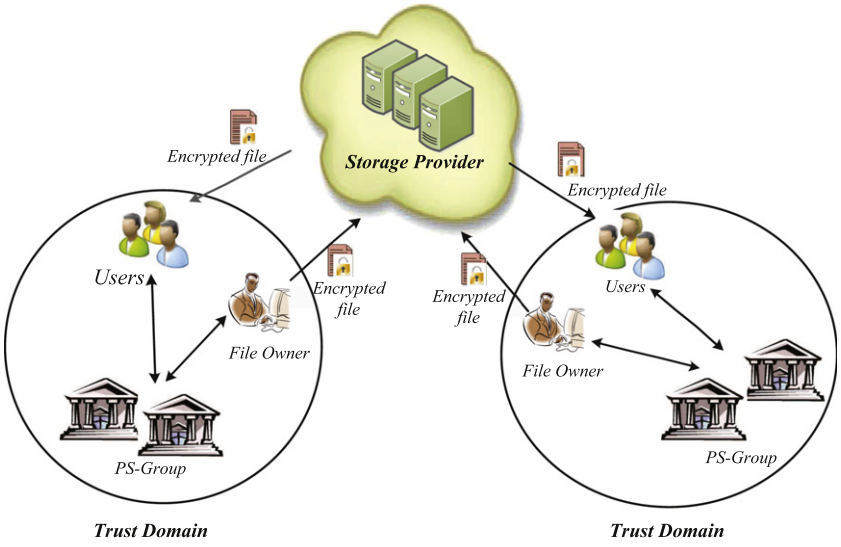


Fig. 1. The system model of Shield.

3.2. Design goals

To realize such cloud-data-sharing service, Shield should achieve the following goals.

Underlying file systems independence. Considering the variety of file systems operated on users' machines, Shield should be designed with no modification to the underlying file systems to ensure good portability.

End-to-end protection for confidentiality and integrity. As a basic guarantee of data security, Shield needs to provide end-to-end confidentiality and integrity, in which unauthorized users cannot learn actual information from the data and any unpermitted modification or accidental corruption to the data will be timely detected.

Keys management and key distribution. With the increasing number of files, the number of keys to be managed expands dramatically and owners will be exhausted to manage the massive numbers of keys, especially when files are encrypted in a smaller unit like file blocks. To free owners from cumbersome keys management, Shield needs to have a delicate mechanism for efficient keys management without the participation of owners. Furthermore, cloud servers should not be able to participate in keys management due to security concerns.

Unlike some previous shared cryptographic file systems (e.g., Plutus [17]), Shield does not rely on out-of-band mechanisms or keys shared with each pair of users to distribute keys. Instead, Shield should exploit a new mechanism to distribute corresponding keys to users according to their access permissions. This mechanism should combine access control and part of the key distribution without needing direct online communication between file owners and users.

Efficient permission revocation. Traditional permission revocation not only introduces a burst re-encryption for the involved files but also requires key re-distribution to the surviving users, resulting in considerable performance degradation. To alleviate this performance loss, Shield should design an efficient and secure permission revocation mechanism.

Concurrent writing support. Most cryptographic file systems handle concurrent reading well, but few of them have good support for concurrent writing, owing to consistency consideration and the restrictions to supply efficient integrity checking. To meet the condition in which multiple users may apply to simultaneously update a file, Shield should support concurrent writes while preserving file consistency and maintaining integrity.

4. Design

4.1. Overview

Shield is designed to *secure data storing and data sharing inside the trust domain (e.g., an organization or department) under the network and storage environments shared among multiple parties, and to save file owners from tedious management without fully trusting the cloud server.*

Aiming at removing the burden of keys management on the client side, the security control information (including user's access permission, integrity information and decryption keys) of each data file will be expressed in the form of a normal file (denoted as ***s**ecurity **c**ontrol **file***, sc-file) and stored with the corresponding encrypted data file (called ***d**ata **file***, d-file) in the cloud storage.

Fig. 2 presents the overview of Shield and Table 2 lists the abbreviations in Fig. 2. Both sc-file and d-file are stored in the cloud storage, while PS grasps two keys, PEK and PSK. The d-file is constituted of many file blocks with constant size that are encrypted by file block keys (the 1st-level keys). The sc-file contains three parts, i.e., the ***a**ccess **c**ontrol **b**lock* (ACB), ***M**erkle **H**ash **T**rees* (MHTs) and a ***r**oot **h**ash **l**ist* (RHL). The ACB includes the hashed file name, an ***a**ccess **c**ontrol **l**ist* (ACL) listing the authorized users and corresponding access permissions, and the 2nd-level keys (LBK and FSK) encrypted by PEK (the 3rd-level keys). Moreover, the integrity of ACB is protected by PSK. The MHT is constructed over a set of both encrypted file block keys and the hash values, in order to provide integrity promise. The RHL stores the hash values of the MHTs' root nodes. As will be discussed later, the root hash values guarantee the integrity of the whole Merkle tree and can only be signed by the writer using FSK. The RHL will also assist the support of concurrent access. Notice that though ACL may be not very scalable, some improvements can be adopted to significantly reduce the latency while PS scans the ACL to verify users' access rights. For example, Shield can construct a symbol-based index,[1] so that the complexity of scanning ACL items can be reduced from $O(n)$ to $O(\log n)$.

[1] The key idea of this construction is that all the items in an ACL sharing a common prefix have a common node.

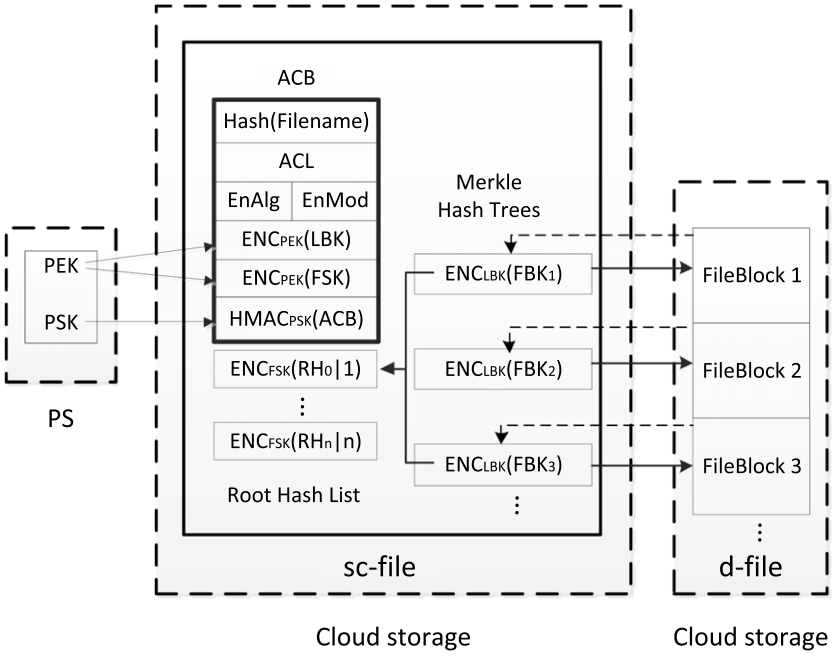


Fig. 2. The overview of Shield.

Table 2
A list for the abbreviations.

Abbreviations	Concepts
ACB, MHT	Access Control Block, Merkle Hash Tree
FBK, LBK, FSK	File Block Key, Lockbox Key, File Signature Key
PEK, PSK	PS Encryption Key, PS Signature Key
ACL, RHL	Access Control List, Root Hash List
EnAlg, EnMod	Encryption Algorithm, Encryption Mode

4.2. Hierarchical symmetric key organization

The scale of managed keys and the efficiency of keys regeneration after revocation become two extremely important concerns when designing a shared secure storage system. The straightforward idea of selecting a random symmetric key for every file suffers from the sequela of the whole-file re-decryption and re-encryption for any modification to the file. Maintaining multiple keys for each file may be another solution, but inefficient key organization would easily result in complicated system management. Because of the problems above, Shield exploits a hierarchical key organization and classifies the symmetric encryption keys into three levels, which can be referred in Fig. 2.

4.2.1. The 1st-level keys: file block keys

To maintain efficiency when dealing with large files, the files in Shield will be cut into file blocks with a constant size (e.g., 4 KB), and each file block will be encrypted by a symmetric key called the ***f**ile **b**lock **k**ey* (FBK), which is a random string.

4.2.2. The 2nd-level keys: security control file keys

The 2nd-level keys are the ***l**ock**b**ox **k**ey* (LBK) and the ***f**ile **s**ignature **k**ey* (FSK), both of which are kept secretly in the sc-file. A random LBK will be generated for every file and used to encrypt that file's FBKs. Therefore, only those who obtain the LBK can unlock the lockbox and recover the FBKs to decrypt the encrypted file blocks. The confidentiality of the root hash is protected by the FSK, which is also used to differentiate read and write permissions (for more details about the access control differentiation one can refer to Section 5.1). Unlike the asymmetric FSKs in Plutus [17], the LBK, FBK and FSK here are all symmetric keys, which will greatly reduce the cryptographic cost when users operate the file content.

4.2.3. The 3rd-level keys: PS keys

Because FSK and LBK are stored in the sc-file and placed in public storage, their confidentiality and integrity should be protected. This protection is achieved by two top-level symmetric keys possessed by the PS, i.e., the ***PS E**ncryption **K**ey* (PEK) and the ***PS S**ignature **K**ey* (PSK). PEK protects the confidentiality of LBK and FSK, while PSK involves in the computation of the HMAC value to guarantee the integrity of ACB. These two keys should be protected secretly by employing either some hardware-assisted measures (e.g., a smart card to manage the cryptographic keys) or proactive cryptographic mechanisms [15,20].

This three-level hierarchical symmetric key system can efficiently organize huge amounts of encryption keys and reduce extra storage cost.

Discussion: Shield places the protection of top-level keys (i.e., PEK, PSK) and ciphertext (i.e., sc-file, d-file) on the PS and the cloud server, respectively. To avoid the single point attack against PS, we can organize multi-PS to be a PS-Group and partition PEK and PSK into many shares. Every PS is placed with a share of PEK and PSK, so that a user can operate the file content only after the admission of a certain number of PSs. Obviously, the PS-Group is more secure and reliable than a single PS. In the PS-Group, every PS possesses symmetric keys $\{PEK_i, PSK_i\}$. LBK and FSK are divided into n shares $\{lbk_1, \ldots, lbk_n\}$ and $\{fsk_1, \ldots, fsk_n\}$ by using a (n, k)-secret-sharing mechanism, and both lbk_i and fsk_i are encrypted by PEK_i. When a user plans to access a file, he is required to issue a request to the PS_i for the decryption for lbk_i, so that any k received responses $\{lbk_i\}_{i=t_1}^{t_k}$ from $\{PS_i\}_{i=t_1}^{t_k}$ can reconstruct LBK successfully. This improved design can obviously remove the key dependence of the whole system over every single PS and promise that the general operations in Shield will not be affected as long as no more than $(n - k)$ PSs are captured. The adversaries will still be unaware of either LBK or FSK as long as no more than k PSs are invaded and moreover, it will be a burdensome task for them to attack multiple well-protected PSs, so that the security of data files will be well protected. We are targeting the design of the PS-Group for our future work.

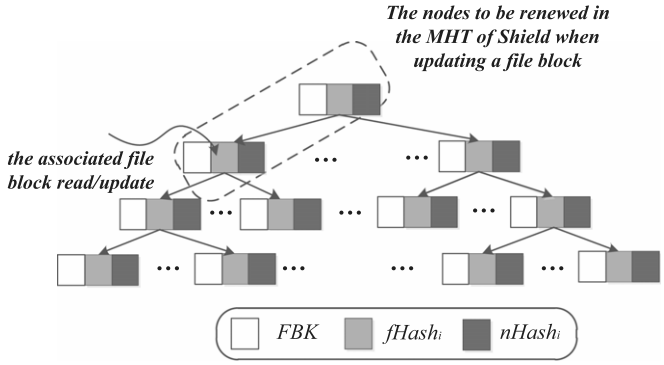


Fig. 3. The MHT in Shield. Each node in MHT has m children at most. The dashed line shows the re-calculation of relevant node when a file block is updated or read.

4.3. Integrity protection

To improve the access performance, Shield proposes a new design of MHT as shown in Fig. 3. It keeps a mapping from the file blocks to the nodes of MHT (including internal nodes), for example, Shield maps the sequential file blocks to the nodes of MHT from top (i.e., the root node) to down (i.e., the leaves nodes). Specifically, every node in Shield's MHT includes FBK_i, fHash_i, and nHash_i, where FBK_i is the file block key of the ith file block, fHash_i denotes the hash value of that file block, and nHash_i is the hash value of the concatenation of its children. FBK_i is then encrypted by the LBK, while fHash_i and nHash_i need no encryption. Finally, the root nodes are signed by FSK and stored in the *Root Hash List* (RHL) of the sc-file to authenticate the integrity of the whole file and all the FBKs. Obviously, the root hash value of an MHT aggregates the information of all the FBKs, fHashes and nHashes in this MHT.

With this MHT structure, a legal modification will trigger the following steps: (1) update the corresponding file block; (2) re-compute the corresponding fHash; (3) renew the nHash of the involved nodes following the path from the node associated to the file block to the MHT's root; (4) re-sign the root node of the MHT using FSK. Before accessing the file content, the user should first check the integrity by re-computing the root hash value following the path from the nodes assigned to the target file blocks to the root node, and comparing it with the pre-computed root hash value (an example is shown in Fig. 3).

4.4. Support for concurrent access

For MHT and other forms of hash trees in existing works [15,13], since only one modifying process of thread is allowed each time (otherwise the root hash will be confused), concurrent writes are forbidden even to different parts of a file. To provide writing concurrency, Shield divides a file into different parts, each of which is covered by a separate MHT. Thus, concurrent writes are allowed as long as they are issued to the parts covered by different MHTs.

However, supporting concurrent writing over multiple MHTs is a non-trivial issue. The straightforward way of computing another hash for the concatenation of all these root hash values would lead again to poor writing concurrency. To solve this problem, Shield proposes a new structure called *root hash virtual linked list*. As shown in Fig. 4, the root hashes of the MHTs belonging to the same file are concatenated with an index. For each root hash node except the last one in the linked list, its index will point to the next neighbor node (i.e., $\text{index}_i := i + 1$, where $1 \leq i < \text{end}$) while the index of the last node will point to itself (i.e., $\text{index}_{end} := \text{end}$).

This design can be used to validate the integrity and the order of the MHTs. Whenever a root hash node is accessed, Shield will first check whether the node is at the correct position by using its index information (for non-end nodes, $\text{index}_i = i + 1$; for the last node, $\text{index}_{end} = \text{end}$). This necessary checking makes sure the root hash node in the linked list has not been illegally modified or removed.

In addition, since each root hash node is encrypted by the FSK, any violent change to the node without FSK would make the corresponding index unintelligible when the node is decrypted by authorized users, and thereby, the interpolation will be detected. The linked list is virtual because the index relation is established by reading the root hash values sequentially from the sc-file without any pointers.

Another significant problem is how to maintain the correct version of data files when multiple different versions are generated at the same time. To address this problem, the data files are divided into many sub-files while each sub-file is covered by an MHT. When authorized users plan to update the files, they can fetch the targeted sub-files, update the content and the associated MHTs, and submit them to the cloud server. It should be noticed that when a sub-file is updated, the associated MHT will be renewed by either adding or removing leaf nodes, just according to the changed size of sub-files. Thus, only the structure of the MHT that associates with the updated sub-file will be influenced, while that of other MHTs will be reserved. In addition, to keep the consistency of each sub-file, when a sub-file is being written, the cloud server will lock the associated root in RHL until the operation finishes.

Since every subfile is covered by an MHT where each MHT has the constant configuration, such as the number of children nodes that an internal node has and the height of an MHT. Therefore, every subfile has the same size. For the access control, we currently assume the granularity of access control is file-level. Extending the access control to the subfile-level is feasible, but the subsequently caused cost is considerable. For example, we have to design the ACB for each subfile, which will significantly increase the storage cost.

4.5. Permission revocation

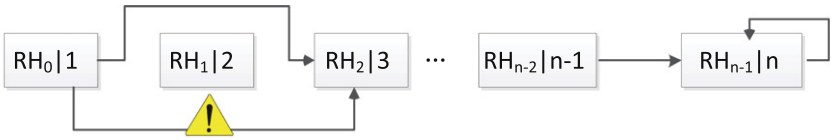
In many encrypt-on-disk systems [15], permission revocation usually demands the immediate re-encryption of the targeted file and the re-distribution of new keys to the surviving data users to that file. This behavior incurs a performance penalty caused by burst operations and potentially breaks the system stability. Shield considers the revocation when the file owner revokes a user's access permission to a certain file and this operation will not affect other files that the revoked user can access; this is different from the revocation banishing a user from a user-group.

Shield adopts *lazy revocation* [32], which defers the re-encryption until the first time when the file is updated after the revocation. Lazy revocation only renews the involved file's security information, but it usually requires complex key organization. When permission revocation occurs in Shield, PS will generate a new LBK and a new FSK into the ACB and re-encrypt all the FBKs and the root hash values of this file (not the whole file set), the size of which is far smaller than that of the whole file. Compared to aggressive revocation, lazy revocation can significantly reduce the amount of data to be suddenly re-encrypted; for instance, if the file block size is 64 KB and the adopted hash algorithm is SHA-1, then a revocation for a 1 GB file will cause re-encryption for the FBKs and root hashes, whose size is only 320 KB.[2]

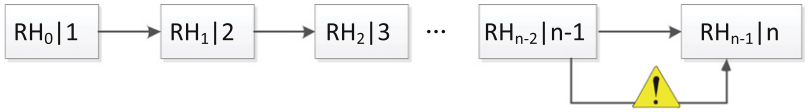
When some file blocks are modified after the revocation, the writer will unlock the original FBKs to decrypt the data, update the content, select other random strings to serve as the new FBKs and produce the new ciphertext using these newer FBKs. This step neither requires complicated mechanisms, nor demands extra space

[2] Compared to FBKs, the number of root hash values is less by orders of magnitude, thus their cost can be ignored.

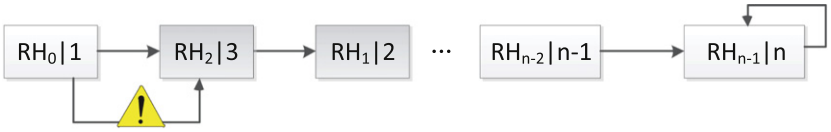
(a) An intact root hash virtual linked list.



(b) Node RH_1 has been illegally removed from the list. This can be detected by verifying the index of RH_0 which should point to RH_1.



(c) Node RH_{n-1} (the tail of the list) has been illegally removed from the list. This can be detected by verifying the index of RH_{n-2} which should point to itself (if it was the genuine tail).



(d) Node RH_1 and Node RH_2 have been illegally swapped in the list. This can be detected by verifying the index of RH_0 which should point to RH_1.

Fig. 4. The root hash virtual linked list.

to keep the historical statuses of FBKs when compared to other lazy revocation mechanisms [17], reducing considerable storage and computation overhead.

Lazy revocation seems to be a trade-off between efficiency and security; however, the security of the whole system is not actually weakened. In fact, even if the re-encryption of the involved files is executed once the revocation happens, the revoked user can still recover the original data information either by reading them from the cache of local machines or by copying them before revocation. Moreover, if we try to cover the security about the non-changed content by monitoring the status for all the file content, the complexity will significantly increase. Thus, it is much more practical to prevent revoked users from accessing the updated file content after the revocation.

4.6. Key distribution and read–write differentiation

Shield uses an sc-file to store all the relevant keys of the data file without requiring users to manage them. Users can get the sc-file from the cloud server and decrypt the FBKs after passing the permission checking executed by the PS. During the key distribution, file owners and users do not have to be online at the same time to communicate with each other and need not keep any keys at local machines. The PS and each user should share a security channel for sending a small amount of security information data; this channel is established with PKI (*public key infrastructure*) when a new user joins the system and thus will not slow down the file sharing.

The access rights of readers and writers to files should be differentiated. An alternative solution of employing a public-key system [17,26,15] (i.e., send the public keys and private keys to readers and writers, respectively) will lead to a considerable computation burden. Shield exploits a symmetric method with the help of FSK, i.e., the writer obtains LBK, root hash values and FSK to access the file content and re-signs the MHT's root hash after the update, while the reader only gets LBK and root hash values to read the file content.

5. Access protocol

5.1. File writing and file reading

The steps for writing a file are shown in Fig. 5. The writer first sends the requests to the cloud server for the retrieval of the wanted d-file and sc-file. The cloud server will then check if the writer has the access permission to the files and returns them if his access checking is passed. The writer then takes the ACB and the RHL from sc-file and sends them with the write request to the PS. After receiving the request, the PS first calculates the HMAC value of the ACB by using PSK to ensure the integrity of ACB, and searches through the ACL to validate whether his access permission confirms the request. After that, the PS decrypts the LBK and FSK using PEK and recovers root hash values using FSK, and returns all of them to the writer (step 3, 4, 5, 6). The writer decrypts FBKs using LBK, re-computes the MHTs' root hash values and compares them with those returned from the PS to verify the integrity of the MHT. Then he writes the new encrypted data to the d-file and updates the MHTs including the nodes and the roots to the sc-file (steps 7, 8, 9). Finally, the writer submits the updated sub-files and corresponding MHTs to the cloud server (step 10).

The steps for reading a file are similar as file writing, except that the reader will only obtain LBK and the decrypted root hash values from the PS. The reader will also check the data integrity before accessing the content.

5.2. User revocation

Fig. 6 shows the procedures of permission revocation. When revocation happens, (1) the file owner first fetches the needed files from the cloud server and sends the ACB and a list of revoked users to the PS (steps 1, 2); (2) the PS confirms the identity, deletes the revoked users from the original ACL, and generates a new lockbox key LBK' and a new file signature key FSK' for the file (steps 3, 4, 5). Then the PS re-encrypts LBK' and FSK' with the PEK, replaces the old

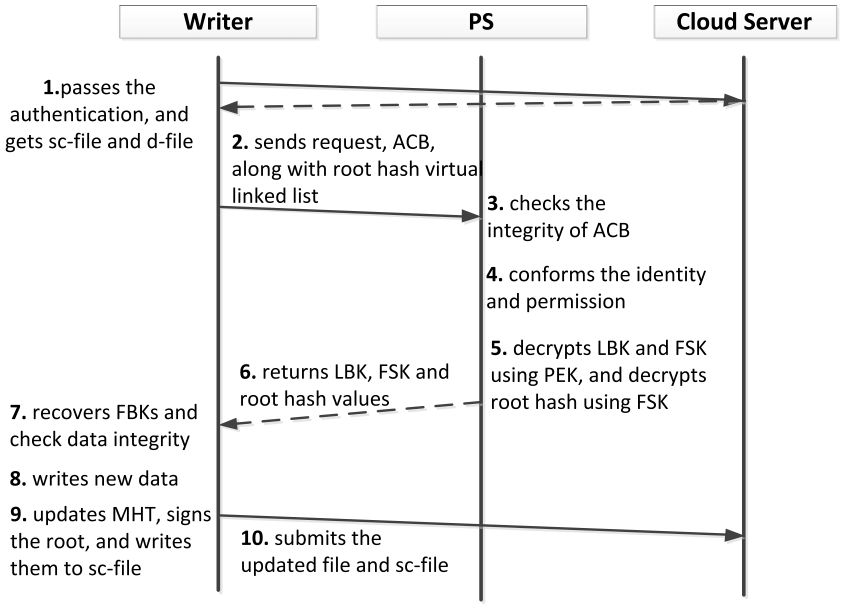


Fig. 5. The write procedure of Shield.

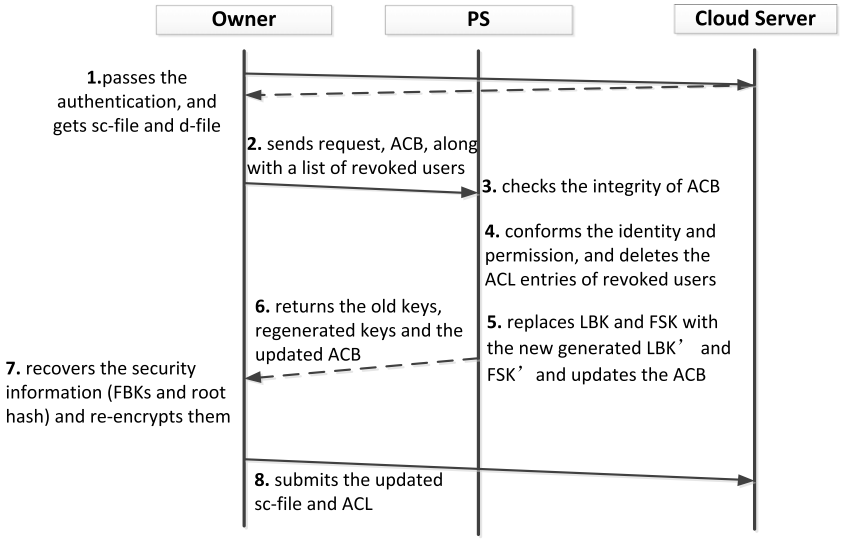


Fig. 6. The revocation procedure of Shield.

ACB with the new ACB' (including the modified ACL, LBK' and FSK'), and then re-computes the HMAC' of ACB' using PSK (step 5). Next, the PS returns ACB', FSK, FSK', LBK and LBK' to the file owner (step 6); (3) the owner then decrypts all FBKs (resp. root hash values) using the LBK (resp. FSK) and re-encrypts these FBKs (resp. root hash values) using LBK' (resp. FSK') and updates the sc-file (step 7) ; (4) finally, the owner sends the updated sc-file and ACL to the cloud server (step 8).

6. Security analysis

In this section, we mainly discuss the security protection offered by Shield and show how Shield can resist these attacks.

Shield keeps encrypted data on the disk and for a user without hosting the proper secret key (i.e., FBK) it will be extremely hard to recover the plaintext. FBK is then locked by LBK. Therefore, to read the data, one must obtain LBK first to free FBK from the lockbox.

Shield uses cryptographic hash values to concentrate the data information and constructs MHT to guarantee the integrity of FBKs and file blocks, therefore any unauthorized change to FBKs and the file blocks will be sensed. In addition, Shield pays more attention to protect the security of updated data when a permission revocation happens. The excluded user cannot obtain the updated data by using the expired keys.

The permission differentiation between readers and writers is achieved by FSK, which is used to prove the legitimate update operation by signing the root hash values. Without the knowledge of the FSK, the reader cannot produce the valid signature even if he tries to update the file content. This incorrect signature will be detected by other authorized writers when they check the data integrity before writing. Shield cannot prevent the readers from infusing garbage into the files, since unauthorized writes prevention is out of the research scope of this paper.

As an independent party, the PS may suffer many malicious attacks aiming at PEK and PSK. To stably grasp these two keys, the following practical methods can be adopted. First, Shield can use Tamper-Resistant Hardware to store PEK and PSK to prevent the leakage of PS keys once PS is invaded, for example, utilize a Hardware Security Module (HSM) or a smart card that is employed for

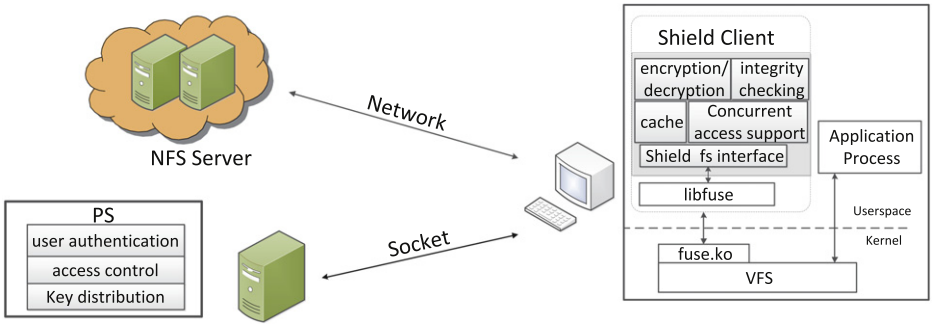


Fig. 7. The architecture of Shield.

the management of cryptographic keys [4]. Second, Shield can employ existing proactive cryptographic methods to protect user's information against PS, such as proxy re-encryption [2]. Third, special security hardware can also be used to execute the encryption and decryption to LBK and FSK without revealing PEK and PSK. A more practical way is to design a PS-Group constituted by multiple PSs, as we mentioned in Section 4.2.3, to reduce the trust over each PS, so that even if a number of PSs (no more than the threshold in the adopted secret sharing scheme) are captured, the data security is still preserved. Even if PEK and PSK are obtained by the malicious user, he cannot learn the information exceeding his access privilege either, since the access control performed by the cloud server will forbid the illegal attempt to fetch the sc-file and d-file exceeding his access permission.

The attackers may even launch DoS (*Denial of Service*) attack against PS by aggressively requesting the service of PS, making it too exhausted to handle other users' requests. Due to the PS's simple logic and statelessness, Shield can mitigate this attack with ease by applying a PS-Group with a load balance mechanism to render services.

A malicious actor can perform rollback attacks by misleading users into accessing stale data. A typical scenario happens when a revoked user replaces the current sc-file, the ACL of which does not include this user, with an obsolete version he kept before the revocation was executed. Then this revoked user will pass the verification of PS and still have file access permission which should have been invalidated, since the PS cannot distinguish the freshness of the sc-file. SiRiUS resists this attack by periodically calculating the timestamp of the owner's file hash hierarchy, organizing them to be a hash tree, and signing the root; this method can also work in Shield just by introducing a symmetric key to re-sign the root.

Another possible attack is the replay attack, where a pretended user may eavesdrop on conversations between valid users and the PS, and submit stolen information to pass the PS's authentication. This kind of attack can be mitigated by applying some assisted techniques, such as the application of one-time session tokens.

PS may suffer the attack when the malicious user tries to invade the PS and read the key information from the memory, such as FSKs and LBKs. This is a common method of attack against software encryption systems, and many ways have been developed to prevent such attacks, such as implementing encryption using hardware, changing the server's configuration, and blocking the ports. Hardware key-deletion techniques can also be employed to erase the keys when the PS is idle.

There is also a potential collusion attack that an attacker controlling the PS tries to collude with the cloud server. As mentioned above, due to the worry of both economical loss and reputation degradation, the cloud server is actually not willing to passively leak user's information. Moreover, we can extend the single PS to the PS-Group consisting of multiple semi-trusted PSs to increase attacker's efforts to invade these PSs. In reality, it is extremely hard to steal PEK_i and PSK_i when these keys are well protected.

7. Implementation

Shield is implemented on Linux using the FUSE [10] (Filesystem in User space) framework and OpenSSL [34], where FUSE is independent of specific underlying file systems and offers good portability. Moreover, the OpenSSL is used to carry out the cryptographic operations.

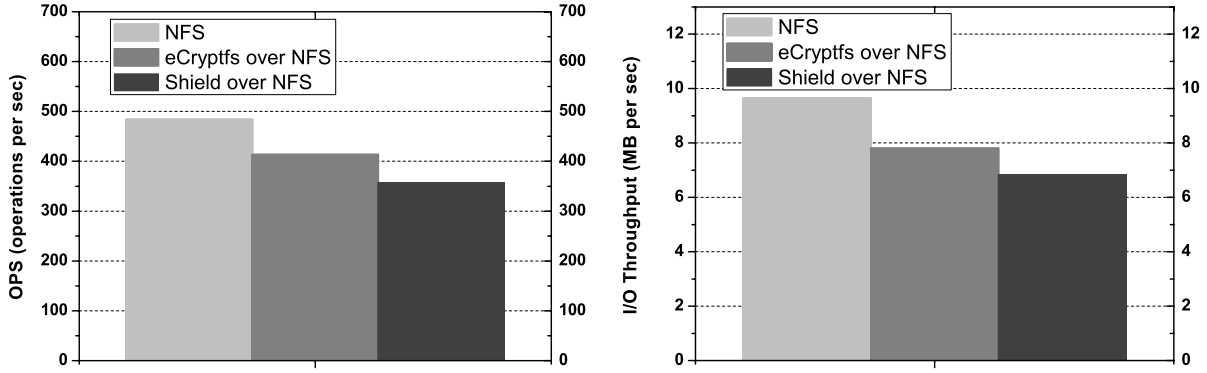
Shield has three components: the PS, Shield clients, and the underlying file system, as shown in Fig. 7. As we mentioned above, only a few simply symmetric cryptographic computations (e.g., the decryption of LBK and FSK, and HMAC verification of ACB) are implemented at the PS and it does not need to keep user's status except for maintaining a cryptographic communication channel between each user and the PS. This simple logic brings three advantages: (1) low workload. The lightweight workload on the PS greatly improves its support capability; (2) simplicity for achieving system reliability and availability. Any PS with the same PEK and PSK can immediately take over the service of a crashed PS; (3) good scalability. The number of PSs can be easily extended.

Shield takes PKI [11] to act as proof of the identities of the composed parties and will not be involved in any cryptographic computation. A new user will apply for certification from an RA (Registration Authority) before using Shield. Although PKI is based on asymmetric encryption, it is a one-time cost only when a new user joins Shield; after that, only symmetric encryption needs to be carried out during the file access, avoiding too many performance penalties. Notice that the purpose of using PKI here is different from that of SiRiUS: Shield uses PKI merely to identify users while SiRiUS [15] and SNAD's Scheme 2 [26] use it to distribute keys and perform asymmetric cryptographic operations.

Shield moves most cryptographic computations (including encryption, decryption, and integrity checking) at clients, which improve the system scalability. The communication between clients and the PS is done by SSL/TSL [31]. In addition, Shield supports multiple requests by utilizing the different threads in FUSE. To guarantee semantic correctness and data consistency, Shield exploits POSIX pthread library to deal with synchronization and mutual exclusion between different threads that share some common data structures and status.

8. Performance evaluation

In this section, we present a series of evaluations for frequent file operations in real-world scenarios. We first analyze the latency of permission operation. To illustrate the generic file operations in daily usage, we deploy Shield on top of NFS in the cluster, conduct two benchmarks (i.e., PostMark [18] and Filebench [9]) that are widely employed to portray the file operations in different real applications, and compare the performance of Shield with that of eCryptfs [16]. Finally, we evaluate the PS's support capability through a pressure test.



(a) The comparison on OPS.

(b) The comparison on I/O throughput.

Fig. 8. Filebench results over cluster. The larger OPS (Operations Per Second) and throughput generally indicate better performance.

Table 3
Comparison of permission operation cost.

Operation	Time of Shield (s)	Time of aggressive revocation (s)
Permission granting	0.033	–
Permission update	0.035	–
Revocation	0.234	19.32

The reasons for choosing eCryptfs as a reference of Shield are as follows. First, eCryptfs has some design criteria in common with Shield, such as good portability and confidentiality protection. Second, eCryptfs offers less security protection compared to Shield, such as integrity protection and efficient revocation, thus the performance comparison can to some extent indicate the feasibility of deploying Shield in real application.

The performance evaluation is conducted on several nodes connected by a 1000 Mbps Ethernet link. The node that acts as the client machine and the PS is equipped with a quad-core Intel(R) Xeon X5472 processor running at 3 GHz and 1 GB RAM, while the node serving as the file server is configured with a quad-core Intel(R) Xeon X5472 processor running at 3 GHz and 8 GB RAM. The operating system of every node is Ubuntu 10.04.3 LTS and the kernel is the 2.6.32-33-server. We choose AES-256 as the cipher for both Shield and eCryptfs, employ SHA-1 as the hash function, select SHA-1 based HMAC as the MAC algorithm, and use 4 KB as the size of the file block.

8.1. Permission operation cost

We first evaluate the time overhead of modifying user's access permissions, including granting read permission to a new user, updating a user's permission, and revoking a user's privilege. The test sets include a PS and a client machine and the size of the targeted file is specified to 1 GB. Meanwhile, we also define the size of subfile as 512 MB. We first grant the read permissions of the file to 1000 users, update those permissions to write permissions, and then separately revoke all these permissions. We also test the performance of aggressive revocation by recoding the time of decryption and re-encryption for the specified file. The operation time for each user is averaged in Table 3.

When granting the permission to a new user, Shield simply inserts the user's name and permission into the ACL and re-computes the HMAC of the ACB, so the latency is very short (about 0.033 s). Updating an existing user's permission will cause the update of the corresponding item in ACL and the re-computation of HMAC for the updated ACB, leading to overhead similar (about 0.035 s) to that of permission granting. When revoking an existing user, Shield has to re-sign all root hashes of the file and re-encrypt the FBKs; thus, it takes more time (about 0.234 s).

Aggressive revocation needs to re-decrypt the targeted file by using the original secret key and re-encrypt it by using the new generated key, to forbid the excluded user from accessing the file content after the revocation. This intensive operation will cause high latency (about 19.32 s) at the client node. Compared to the size of targeted file, the size of its FBKs to be re-encrypted in Shield is much smaller, resulting in the efficiency improvement of several orders of magnitude brought by the adoption of lazy revocation.

8.2. Filebench over cluster

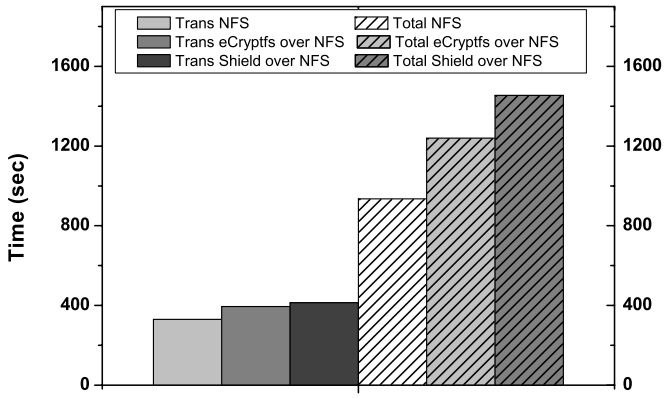
To evaluate the performance of ordinary file operations at Shield clients, we conduct the benchmark using Filebench [9]. We choose the "fileserver" workload Filebench to simulate generic file operations on file servers, such as creating/deleting/closing files, and reading/writing/appending data.

This test is set up on a set of clusters with seven nodes, including a PS, a file server and five client machines. The version of Filebench is 1.4.9.1. Every client machine mounts Shield and eCryptfs over NFS respectively and invokes the "fileserver" workload at the same time. The number of threads is limited to 1 at every client side. We set the mean file size to be 6 MB (the size of the subfile equals that of the file in this evaluation) and specify the number of files to be 1000, so that the size of file set can mitigate the disturbance caused by the memory cache. We perform the test ten times and show the averaged performance of a client machine in Fig. 8.

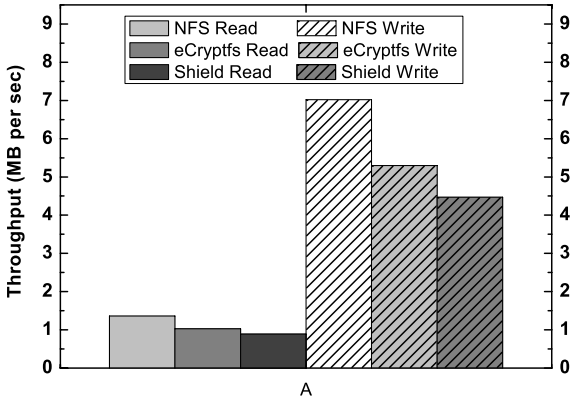
Given the necessary cryptographic operations performed at the client machines to protect data confidentiality, the OPS of both eCryptfs and Shield inevitably suffers a loss when compared with that of raw NFS. In addition, compared with eCryptfs, Shield offers extra security protection, such as integrity protection and secret keys management. These extra-secure protocols require users to contact the PS for authentication, perform integrity checking while reading files, and maintain the MHT when updating the file content. These additional security protection measures will cause about 13% performance degradation on the client machines when compared with eCryptfs, as proved by the test results of Filebench in Fig. 8. Given that Shield supplies many extra security guarantees, we argue that the performance of the daily operations of Shield in the network environment is reasonable and acceptable.

8.3. PostMark over cluster

In this test, we use PostMark [18] to give a picture of performance in ephemeral file operations. In the design of PostMark, a large pool of continually changing files whose sizes can be narrowed among a pre-configured range is built first and a configured number of transactions is executed later. The transactions consist of the following two types of operations: (1) Create File or Delete File, and (2) Read File or Append File.



(a) The comparison on total elapsed time and transaction time.



(a) The comparison on read/write throughput.

Fig. 9. PostMark results over cluster.

This test is performed on a set of seven nodes, including five client machines, a PS and a file server. The version of PostMark is v1.5. In this test, the file size is allowed to vary in a range from 5 MB to 6 MB and the number of files is set to 1000, making the total size of test suits multiple times larger than the system memory to reduce the effect of memory cache. We also set the size of subfile as the file size. The transactions are performed 5000 times and the elapsed time is accumulated. We deploy Shield and eCryptfs on top of NFS respectively and invoke PostMark at the client nodes simultaneously. 1242 files are created during the test, of which 984 files are deleted alone while the remaining files are deleted in the transactions. The averaged results are presented in Fig. 9.

In Fig. 9(a), both Shield and eCryptfs require a bit more time to finish the whole test and transaction, due to the supplementation of security methods. Meanwhile, both the total elapsed time and the transaction time of Shield are about 6% and 13% larger than those of eCryptfs respectively, which seem reasonable when considering the different design criteria between Shield and eCryptfs. From Fig. 9(b), we can observe that the read (resp. write) performance of eCryptfs slightly outperformed that of Shield, being about 11% (resp. 14%) quicker; this is because Shield needs users to check (resp. update) the hash values of some involved nodes in MHT when reading (resp. writing) the file content.

8.4. A pressure test for proxy servers

To evaluate the PS's support capacity, we design and implement a benchmark by using one server to act as the PS in this test.

As mentioned above, PS is responsible for authenticating the user's privilege and distributing keys when the user issues an open file request along with the ACB. PS does not involve in the later operations, such as reading, writing, and closing files which are actually done by the users at the client nodes, so we can measure the PS's support capacity by testing how many open requests (i.e., the processing of ACB and encrypted root hash values) could be accommodated during one second.

Our benchmark first initiates N threads on the PS to deal with the file-open requests with read/write permission and launches the "pthread_exit" command after it finishes a response (i.e., the secret keys, such as LBK). Our benchmark waits until all N threads finish. We record the running time from creating N threads to when they all finish as *Request Handle Time* (RHT).

We perform four tests for $N = 1000, 2000, 4000$, and 8000 respectively and show the results in Fig. 10, from which we conclude three points. First, there is a linear relationship between the RHT and the thread number N. Second, a PS can handle more than 5000 file-open requests in one second. According to previous research [22], the proportion of open operation in the desktop

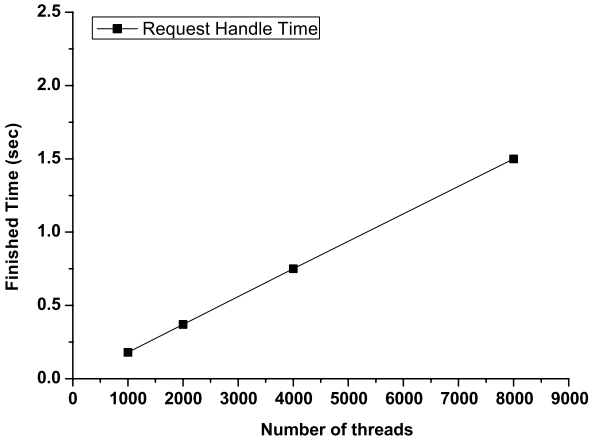


Fig. 10. Pressure test for the support capacity of a proxy server.

and enterprise workloads is about 12%, which indirectly means that one PS can support near 46,296 users online at the same time. Hence, only a small number of PSs are required in a real-life scenario even with massive number of users, proving that the PS is suitable for large-application usage.

9. Conclusion

This paper presents a new system architecture for secure file sharing in public storage and designs a secure and efficient stackable storage system named Shield. In Shield, we construct a hierarchical key organization to efficiently manage secret keys, and use a variant of MHT to accelerate integrity checking. Shield adopts lazy revocation to improve the efficiency for user's permission revocation. In addition, Shield supplies concurrent writing with the data structure of the root hash virtual linked list. A series of evaluations demonstrate that Shield causes about 7%–13% performance degradation when compared with eCryptfs but provide enhanced security and a single PS can support more than 45,000 user's requests in one second.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and Tsinghua University Initiative Scientific Research Program.

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, R. Wattenhofer, Farsite: federated, available, and reliable storage for an incompletely trusted environment, in: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, OSDI'02, ACM, New York, 2002, pp. 1–14.

[2] G. Ateniese, K. Fu, M. Green, S. Hohenberger, Improved proxy re-encryption schemes with applications to secure distributed storage, in: Proceeding of Network and Distributed Systems Security (NDSS) Symposium 2005, The Internet Society, 2005.

[3] M. Blaze, A cryptographic file system for unix, in: Proceedings of the 1st ACM Conference on Computer and Communications Security, CCS, ACM, New York, 1993, pp. 9–16.

[4] L. Buttyán, J. Hubaux, Enforcing service availability in mobile ad-hoc wans, in: Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing, IEEE Press, 2000, pp. 87–96.

[5] A. Castiglione, L. Catuogno, A. Del Sorbo, U. Fiore, F. Palmieri, A Secure file sharing service for distributed computing environments, J. Supercomput. (2013) Springer.

[6] G. Cattaneo, L. Catuogno, A.D. Sorbo, P. Persiano, The design and implementation of a transparent cryptographic file system for unix, in: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, 2001, pp. 199–212.

[7] L. Catuogno, H. Loehr, M. Winandy, A. Sadeghi, A trusted versioning file system for passive mobile storage devices, J. Netw. Comput. Appl. (2014) Elsevier.

[8] T. Dillon, C. Wu, E. Chang, Cloud computing: issues and challenges, in: Proceeding of the 24th IEEE International Conference on Advanced Information Networking and Applications, AINA, IEEE, 2010, pp. 27–33.

[9] Filebench: File system microbenchmarks, 2006. http://www.opensolaris.org/os/community/performance/filebench.

[10] Filesystem in userspace, 2005. http://fuse.sourceforge.net.

[11] S. Galperin, S. Santesson, M. Myers, A. Malpani, C. Adams, X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP (2013).

[12] D. Garg, F. Pfenning, A proof-carrying file system, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, 2010, pp. 349–364.

[13] E. Geron, A. Wool, Crust: cryptographic remote untrusted storage without public keys, Internat. J. Inf. Secur. 8 (5) (2009) 357–377.

[14] H. Gobioff, G. Gibson, D. Tygar, Security for Network Attached Storage Devices, Tech. Rep. CMU-CS-97-185, Carnegie Mellon University, 1997.

[15] E. Goh, H. Shacham, N. Modadugu, D. Boneh, Sirius: securing remote untrusted storage, in: Proceeding of Network and Distributed Systems Security (NDSS) Symposium 2003, The Internet Society, 2003.

[16] M. Halcrow, ecryptfs: an enterprise-class encrypted filesystem for linux, in: Proceedings of the 2005 Linux Symposium, vol. 1, 2005, pp. 201–218.

[17] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, Plutus-scalable secure file sharing on untrusted storage, in: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST'03, USENIX Association, Berkeley, CA, 2003, pp. 29–42.

[18] J. Katcher, Postmark: A new file system benchmark, Tech. rep., Technical Report TR3022, Network Appliance, 1997, www.netapp.com/tech_library/3022.html.

[19] R. Kotla, L. Alvisi, M. Dahlin, Safestore: a durable and practical storage system, in: Proceeding of the 2007 USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, 2007, pp. 129–142.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, Oceanstore: an architecture for global-scale persistent storage, in: Proceeding of ASPLOS, ACM, New York, 2000, pp. 190–201.

[21] A. Leung, E. Miller, S. Jones, Scalable security for petascale parallel file systems, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM, New York, 2007.

[22] A.W. Leung, S. Pasupathy, G. Goodson, E.L. Miller, Measurement and analysis of large-scale network file system workloads, in: Proceedings of 2008 USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, 2008, pp. 213–226.

[23] Y. Li, N.S. Dhotre, Y. Ohara, Horus: fine-grained encryption-based security for large-scale storage, in: Proceedings of the 12nd USENIX Conference on File and Storage Technologies, FAST'13, USENIX Association, Berkeley, CA, 2013, pp. 147–160.

[24] J. Li, M.N. Krohn, D. Mazières, D. Shasha, Secure untrusted data repository (sundr), in: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation, OSDI, USENIX Association, Berkeley, CA, 2004, pp. 9–9.

[25] D. Mazieres, M. Kaminsky, M.F. Kaashoek, E. Witchel, Separating key management from file system security, ACM SIGOPS Oper. Syst. Rev. 33 (5) (1999) 124–139.

[26] E. Miller, D. Long, W. Freeman, B. Reed, Strong security for network-attached storage, in: Proceedings of the 2002 Conference on File and Storage Technologies, FAST'02, USENIX Association, Berkeley, CA, 2002, pp. 1–13.

[27] B. Neuman, T. Ts'o, Kerberos: an authentication service for computer networks, IEEE Commun. Mag. 32 (9) (1994) 33–38.

[28] D.P. O'Shanahan, Cryptosfs: Fast cryptographic secure nfs, Master's Thesis, University of Dublin, 2000.

[29] S. Patil, A. Kashyap, G. Sivathanu, E. Zadok, I3fs: an in-kernel integrity checker and intrusion detection file system, in: Proceedings of the 18th USENIX Conference on System administration, USENIX Association, Berkeley, CA, 2004, pp. 67–78.

[30] D. Reed, L. Svobodova, Swallow: a distributed data storage system for a local network, in: Proceeding of International Workshop on LocalNetworks, 1980.

[31] E. Rescorla, SSL and TLS: Designing and Building Secure Systems, Vol. 1, Addison-Wesley, Reading, Massachusetts, 2001.

[32] R.L. Rivest, K. Fu, K.E. Fu, Group sharing and random access in cryptographic storage file systems, Master's Thesis, MIT, Citeseer, 1999.

[33] M.W. Storer, K.M. Greenan, E.L. Miller, K. Voruganti, Potshards: secure long-term storage without encryption, in: Proceedings of the 2007 USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, 2007, pp. 143–156.

[34] J. Viega, M. Messier, P. Chandra, Network Security with OpenSSL: Cryptography for Secure Communications, O'Reilly Media, Incorporated, 2002.

[35] S. Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, A data outsourcing architecture combining cryptography and access control, in: Proc. of the 1st Computer Security Architecture Workshop, CSAW 2007, Fairfax, Virginia, USA, November 2, 2007.

[36] S. Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Over-encryption: management of access control evolution on outsourced data. in Proc. of the 33rd International Conference on Very Large Data Bases, VLDB 2007, Vienna, Austria, September 23–28, 2007.

[37] C. Wright, M. Martino, E. Zadok, Ncryptfs: a secure and convenient cryptographic file system, in: Proceedings of the 2003 Annual USENIX Technical Conference, USENIX Association, Berkeley, CA, 2003, pp. 197–210.

[38] W. Xue, J. Shu, Y. Liu, M. Xue, Corslet: a shared storage system keeping your data private, Sci. China 54 (6) (2011) 1119–1128.

[39] S. Yu, C. Wang, K. Ren, W. Lou, Achieving secure, scalable, and fine-grained data access control in cloud computing. in Proceeding of IEEE INFOCOM, 2010.

[40] E. Zadok, I. Badulescu, A. Shender, Tech. Rep. CUCS-021-98, CS Department, Columbia University, 2000.

Jiwu Shu, born in 1968. Ph.D., professor, Ph.D. supervisor and senior member of China Computer Federation. His main research interests include network storage and cloud storage, storage security, parallel process technologies, and so on.

Zhirong Shen, born in 1987, Ph.D. candidate. He received his Bachelor Degree from University of Electronic Science and Technology of China. His current research interests include storage reliability and storage security.

Wei Xue, born in 1974, Ph.D., associate professor. His research interests include parallel algorithm design, network storage and cluster computing.