Key Derivation Algorithms for Monotone Access Structures in Large File Systems

Mudhakar Srivatsa and Ling Liu
College of Computing, Georgia Institute of Technology
{mudhakar, lingliu}@cc.gatech.edu

Abstract. Advances in networking technologies have triggered the "storage as a service" (SAS) model. The SAS model allows content providers to leverage hardware and software solutions provided by the storage service providers (SSPs), without having to develop them on their own, thereby freeing them to concentrate on their core business. The SAS model is faced with at least two important security issues: (i) How to maintain the confidentiality and integrity of files stored at the SSPs? (ii) How to efficiently support flexible access control policies on the file system? The former problem is handled using a cryptographic file system, while the later problem is largely unexplored. In this paper, we propose secure, efficient and scalable key management algorithms to support monotone access structures on large file systems. We use key derivation algorithms to ensure that a user who is authorized to access a file, can efficiently derive the file's encryption key. However, it is computationally infeasible for a user to guess the encryption keys for those files that she is not authorized to access. We present concrete algorithms to efficiently and scaleably support a discretionary access control model (DAC) and handle dynamic access control updates & revocations. We also present a prototype implementation of our proposal on a distributed file system. A trace driven evaluation of our prototype shows that our algorithms meet the security requirements while incurring a low performance overhead on the file system.

1 Introduction

The widespread availability of networks, such as the Internet, has prompted a proliferation of both stationary and mobile devices capable of sharing and accessing data across networks spanning multiple administrative domains. Today, efficient data storage is vital for almost every scientific, academic, or business organization. Advances in the networking technologies have triggered the "storage as a service" (SAS) model [17, 15]. The SAS model allows organizations to leverage hardware and software solutions provided by third party storage service providers (SSPs), thereby freeing them to concentrate on their core business. The SAS model decouples physical storage from file management issues such as access control and thus allows the file system to scale to a large number of users, files, and organizations. However, from the perspective of the organization (content owner), the SAS model should address at least two important security issues: (i) How to maintain the confidentiality & integrity of files stored at the SSPs? (ii) How to securely and efficiently support flexible access control policies on the file system?

Cryptographic File Systems. Cryptographic file systems address the first problem. These file systems essentially maintain the confidentiality and integrity of the file data by storing it in an encrypted format at the SSPs. With the advent of high speed hardware for encrypting and decrypting data, the overhead in a cryptographic file system due to file encryption and decryption is affordably small. Examples of cryptographic file systems include CFS [4], TCFS [7], CryptFS [36] and NCryptFS [35]. Examples of wide-area distributed cryptographic file systems include Farsite [2] and cooperative file system [8].

Access Control. Access control in a cryptographic file system translates into a secure key management problem. Cryptographic access control [14] is achieved by distributing a file's encryption key to only those users that are authorized to access that file. A read/write access to the files stored at the SSP is granted to all principals, but only those who know the key are able to decrypt the file data. However, there is an inherent tension between the cost of key management and the flexibility of the access control policies. At one extreme the access control matrix is highly flexible and can thus encode arbitrary access control policies (static). An access control matrix [19] is $(0, 1)$ matrix $M_{U \times F}$, where U is a set of users and F is a set of files and $M_{uf} = 1$ if and only if user u can access file f. Implement-

ing cryptographic access control would require one key for every element M_{uf} such that $M_{uf} = 1$. This makes key management a challenging performance and scalability problem in a large file system wherein, access permissions may be dynamically granted and revoked.

Our Approach. The access control matrix representation of the access control rules does not scale well with the number of users and the number of files in the system. A very common strategy is to impose an access structure on the access control policies. An access structure, as the name indicates, imposes a structure on the access control policies. Given an access structure, can we perform efficient and scalable key management without compromising the access control policies in the file system? We propose to use an access structure to build a key derivation algorithm. The key derivation algorithm uses a *much smaller* set of keys, but gives the same effect as having one key for every $M_{uf} = 1$. The key derivation algorithm guarantees that a user u can use its small set of keys to efficiently derive the key for any file f if and only if $M_{uf} = 1$.

Monotone Access Structure. In this paper we consider access control policies based on monotone access structures. Most large enterprises, academic institutions, and military organizations allow users to be categorized into user groups. For example, let $\{g_1, g_2, g_3\}$ denote a set of three user groups. A user u can be a member of one or more groups denoted by G_u. Access control policies are expressed as monotone Boolean expressions on user groups. For example, a file f may be tagged with a monotone $B_f = g_1 \wedge (g_2 \vee g_3)$. This would imply that a user u can access file f if and only if it belongs to group g_1 and either one of the groups g_2 or g_3. For example, if $G_{u_1} = \{g_1, g_3\}$ and $G_{u_2} = \{g_2, g_3\}$, then user u_1 can access file f, but not user u_2. Monotone access structures are a common place in role-based access control models [28]. In the RBAC model, each role (say, a physician or a pharmacist) is associated with a set of credentials. Files are associated with a monotone Boolean expression B_f on credentials. A role r can access a file f if and only if the credentials for role r satisfies the monotone B_f.

Our Contribution. In this paper, we propose secure, efficient and scalable key management algorithms that support monotone access structures in a cryptographic file system. (i) *Number of Keys:* We ensure that each user needs to maintain only a small number of keys. It suffices for a user to maintain only one key per group that the user belongs to. For example, a user u with $G_u = \{g_1, g_2\}$ needs to maintain keys one key corresponding to group g_1 and group g_2. (ii) *Efficient Key Derivation:* It is computationally easy for a user to derive the keys for all files that she is authorized to access. For example, a user who has the key for groups g_1 and g_2 can easily derive the encryption key for a file f with $B_f = g_1 \wedge (g_2 \vee g_3)$. (iii) *Secure Key Derivation:* It is computationally infeasible for a user to derive the key for any file that she is not authorized to access. For example, for a user who has the keys only for groups g_2 and g_3, it is infeasible to guess the encryption key for a file f with $B_f = g_1 \wedge (g_2 \vee g_3)$. (iv) *Discretionary Access Control (collusion resistance):* It is computationally infeasible for two or more colluding users to guess the encryption key for a file that none of them are independently authorized to access. For example, two colluding users u_1 with $G_{u_1} = \{g_1\}$ and u_2 with $G_{u_2} = \{g_3\}$ should not able to guess the encryption key for a file f with $B_f = g_1 \wedge (g_2 \vee g_3)$. (v) *Revocation:* Our key management algorithms support dynamic revocations of a user's group membership through cryptographic leases. A lease permits a user u to be a member of some group g from time a to time b. Our algorithms allow the lease duration (a, b) to be highly fine grained (say, to a millisecond precision).

Paper Outline. The following sections of this paper are organized as follows. Section 2 describes the SAS model and monotone access structures in detail. Section 3 presents a detailed design and analysis of our key management algorithms for implementing discretionary access control using monotone access structures in cryptographic file systems. Section 4 sketches an implementation of our key management algorithms on a distributed file system followed by trace-driven evaluation in Section 5. Finally, we present related work in Section 6 followed by a conclusion in Section 7.

2 Preliminaries

In this section, we present an overview of the SAS model. We explicitly specify the roles played by the three key players in the SAS architecture: content provider, storage service provider, and users. We also formally describe the notion of user groups and the properties of monotone access structures on user groups.

2.1 SAS Model

The SAS model comprises of three entities: the content provider, the storage service provider and the users.

Storage Service Providers (SSPs). Large SSPs like IBM and HP use high speed storage area networks (SANs) to provide large and fast storage solutions for multiple organizations. The content provider encrypts files before storing them at a SSP. The SSP serves only encrypted data to the users. The content provider does not trust the SSP with the confidentiality and integrity of file data. However, the SSP is trusted to perform read and write operations on the encrypted files. For performance reasons, each file is divided into multiple data blocks that are encrypted separately. An encrypted data block is the smallest granularity of data that can be read or written by a user or a content provider.

Content Provider. The content provider is responsible for secure, efficient and scalable key management. We assume that there is a secure channel between the group key management service and the users. This channel is used by the content provider to distribute keys to the users. We assume that the channel between the content provider & the SSP and that between the users & the SSP could be untrusted. An adversary would be able to eavesdrop or corrupt data sent on these untrusted channels. The content provider also includes a file key server. The users interact with the file key server to derive the encryption keys for the files they are authorized to access. The channel between the user and the file key server may be untrusted. In the following sections of this paper, we present an efficient, scalable and secure design for the file key server.

Users. We use an honest-but-curious model for the users. Content providers authorize users to access certain files by securely distributing appropriate keys to them. Let $K(f)$ denote the encryption key used to encrypt file f. If a user u is authorized to access file f, then we assume that the user u would neither distribute the key $K(f)$ nor the contents of the file f to an unauthorized user. However, a user u' who is not authorized to access file f would be curious to know the file's contents. We assume that unauthorized users may collude with one another and with the SSP. Unauthorized users may eavesdrop or corrupt the channel between an authorized user and the SSP. We use a discretionary access control (DAC) model to formally study collusions amongst users. Under the DAC model the set of files that is accessible to two colluding users u_1 and u_2 should be no more than the union of the set of files accessible to the user u_1 and the user u_2. Equivalently, if a file f is accessible neither to user u_1 nor to user u_2 then it should remain inaccessible even when the users u_1 and u_2 collude with one another.

2.2 Monotone Access Structures

In this section, we describe monotone access structures based access control policies. Our access control policies allow files to be tagged with monotone Boolean expressions on user groups. Let $G = \{g_1, g_2, \cdots, g_s\}$ denote a set of s user groups. A user may be a member of one or more user groups. Each file f is associated with a monotone Boolean expression B_f. For example, $B_f = g_1 \wedge (g_2 \vee g_3)$ would mean that the file f is accessible by a user u if and only if u is a member of group g_1 and a member of either group g_2 or group g_3.

We require that the Boolean expression B_f be a *monotone*. This assumption has several consequences: (i) Let G_u denote the set of groups to which user u belongs. Let $B_f(G_u)$ denotes $B_f(g_1, g_2, \cdots, g_s)$ where $g_i = 1$ if the group $g_i \in G_u$ and $g_i = 0$ otherwise. For two users u and v if $G_u \subseteq G_v$ then $B_f(G_u) \Rightarrow B_f(G_v)$. (ii) Let us suppose that a user u is authorized to access a set of files F. If the user u were to obtain membership to additional groups, it does not deny u access to any file $f \in F$ (monotone property). (iii) For all files f, B_f can be expressed using only $\wedge$ and $\vee$ operators (without the NOT ($\sim$) operator) [20]. (iv) Access control policies specified using monotone Boolean expressions are easily tractable. Let G_u denote the set of groups to which user u belongs. Then, one can efficiently determine whether the user u can access a file f by evaluating the Boolean expression $B_f(G_u)$. Note that evaluating a Boolean expression on a given input can be accomplished in $O(|B_f|)$ time, where $|B_f|$ denotes the size of the Boolean expression B_f.

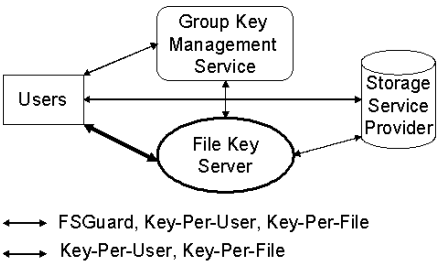

Figure 1: FSGuard Architecture

	File Access	B_f Update	G_u Update	Num keys per User	Storage Overhead
FSGuard	1cpu + 1net	-	-	4	1disk
Key-per-User	-	10^3cpu + 10net	10^6cpu + 10^4net	1	20disk
Key-per-File	-	10^3cpu + 10net	10^6cpu + 10^4net	10^4	1disk

Table 1: Comparison of Key Management Algorithms

3 User Groups

3.1 Overview

Figure 1 shows the entities involved in our design. The core component of our design is the file key server. We use the file key server for securely, efficiently and scaleably managing the file encryption keys. A high level description of our key management algorithm is as follows. Each file f is encrypted with a key $K(f)$. The key $K(f)$ is encrypted with a key encryption key $KEK(f)$. The encrypted file is stored at the SSP. The content owner stores the key encryption keys in the trusted file key server in a *compressed format*. The key server can use the stored information to efficiently derive the key encryption keys on the fly (Section 3.4) and distributes a secure transformation of the $KEKs$ to the users. A transformation on $KEK(f)$ is secure if the transformed version can be made publicly available (to all users and the SSP) without compromising the access control guarantees of the file system (Sections 3.2 and 3.3). We handle dynamic revocations of file accesses to users using a novel authorization key tree (Section 3.5). For comparison purposes, we describe two simple key management algorithms in this section: key-per-user and key-per-file.

Key-per-User. The key-per-user approach associates a secret key $K(u)$ with user u. For any file f, the key server determines the set of users that are permitted to access file f based on the group membership of user u and the monotone B_f. For all users u that can access file f, the key server stores $E_{K(u)}(KEK(f))$ along with the attributes of file f at the SSP. Note that $E_K(x)$ denotes a symmetric key encryption of input x using an encryption algorithm E (like DES [12] or AES [23]) and key K. However, such an implementation does not scale with the number of files and users in the system since the key server has to store and maintain updates on $KEK(f)$ for all f, B_f for all f, $K(u)$ for all u, and G_u for all u. For example, if G_u changes for any u, the key server needs to inspect all the files in the system before determining the set of files to which the user u's access needs to be granted or revoked. For all files f, whose access is either granted to user u, the key server has to add $E_{K(u)}(KEK(f)$ to its attribute. For all files f, whose access is revoked to user u, the key server has to update $KEK(f)$ (to say, $KEK'(f)$); the key server has to add $E_{K(u')}(KEK'(f))$ for all other users u' that are allowed to access file f.

Key-per-File. The second approach is the key-per-file approach. This approach associates a key $K(f)$ with file f. For each user u, the key server determines the set of files that the user is permitted to access based on the group membership of the user u and the monotone B_f. We use the key server to distribute $KEK(f)$ to all users that are permitted to access the file f. We use a group key management protocol [22] to update $KEK(f)$ as the set of users permitted to access file f varies. However, the key-per-file approach also suffers from similar scalability problems as the key-per-user approach.

FSGuard. In this paper, we present our key management algorithms for implementing discretionary access control using monotone access structures in a cryptographic file system. As shown in Figure 1 our approach (FSGuard) does not require any communication between the key server & the group key management service and the key server & the SSP. Table 1 shows a rough cost comparison between our approach and other approaches. Our approach incurs a small processing (cpu) and networking (net) overhead for file accesses. The key-per-user and key-per-file approach incurs several orders of magnitude higher cost for updating a file's access control expression B_f and updating a user's group membership G_u. The average number of keys maintained by one user in key-per-file approach is several orders of magnitude larger than our approach and the key-per-user approach. The storage overhead at the SSP in the key-per-user approach is at least one order of magnitude larger than our approach and the key-per-file approach.

3.2 Basic Construction

In this section, we present a basic construction for building a secure transformation. Recall that a transformation on $KEK(f)$ is secure if the transformed version can be made publicly available (to all users and the SSP) without compromising the access control guarantees of the file system. The basic construction assumes that users do not collude with one another and that the access control policies are static with respect to time. Further, the basic construction incurs a heavy communication cost between the key server and the group key management service. We remove these restrictions in later Sections 3.3, 3.4 and 3.5.

The key idea behind the basic construction is to transform the $KEK(f)$ such that a user u can reconstruct $KEK(f)$ if and only if the user u satisfies the condition B_f. Our construction is based on generalized secret sharing scheme presented in [3]. We assume that all keys are 128-bits long and all integer arithmetic is performed in a 128-bit integer domain (modulo 2^{128}). We use $K(g)$ to denote the group key for group g. When a user u joins group g, it gets the group key $K(g)$ from the group key management service via a secure channel. In this section, we assume a non-collusive setting: a user u knows $K(g)$ if and only if user u is a member of group g. We extend our algorithm to permit collusions in Section 3.3.

Given a monotone Boolean expression B_f we mark the literals in the expression as follows. The i^{th} occurrence of a literal g in the expression B_f is marked as g^i. For example, B_f = $(g_1 \vee g_2) \wedge (g_2 \vee g_3) \wedge (g_3 \vee g_4)$ is marked as $(g_1^1 \vee g_2^1) \wedge (g_2^2 \vee g_3^1) \wedge (g_3^2 \vee g_4^1)$. The key server published $T(KEK(f), B_f)$, where the transformation function T is recursively defined as follows:

$$\begin{aligned} T(x, A_1 \wedge A_2) &= T(x_1, A_1) \cup T(x_2, A_2) \text{ such that } x_1 + x_2 = x \\ T(x, A_1 \vee A_2) &= T(x, A_1) \cup T(x, A_2) \\ T(x, g^i) &= x + H_{salt}(K(g), i) \end{aligned}$$

The symbols A_1 and A_2 denote arbitrary monotone Boolean expressions. The $\cup$ denotes the union operator and $+$ denotes the modular addition operator on a 128-bit integer domain. For the Boolean $\wedge$ operator, we chose x_1 and x_2 randomly such that x_1 + x_2 = x. Observe that knowing only x_1 or x_2 does not give any information about x = x_1 + x_2. Note that H denotes a keyed pseudo-random function (PRF) (like HMAC-MD5 or HMAC-SHA1 [18]). The $salt$ value is randomly chosen per file and is stored at the SSP along with the rest of the file f's attributes. The $salt$ value is used as the key for the PRF H. The above construction can be easily extended to cases where the function T takes more than two arguments:

$$\begin{aligned} T(x, \bigwedge_{i=1}^{n} A_i) &= \bigcup_{i=1}^{n} T(x_i, A_i) \text{ such that } \sum_{i=1}^{n} x_i = x \\ T(x, \bigvee_{i=1}^{n} A_i) &= \bigcup_{i=1}^{n} T(x, A_i) \\ T(x, g^i) &= x + H_{salt}(K(g), i) \end{aligned}$$

Theorem 3.1 *The transformation T described in Section 3.2 secure in the absence of collusions amongst malicious users.*

Proof Recall that a transformation T is secure if $T(KEK(f), B_f)$ can be made publicly available (to all users and the SSP) without compromising the access control guarantees of the file system. It is easy to see that a user u whose group membership satisfies the Boolean expression B_f can easily recover $KEK(f)$ from $T(KEK(f), B_f)$.

We now show that a user whose group membership does not satisfy B_f cannot obtain any information about $KEK(f)$. We prove this using an induction on the number of operators in B_f. Our base case consists of a formula that contains no operators: $T(KEK(f), g^i)$ = $KEK(f)$ + $H_{salt}(K(g), i)$. Note that in a non-collusive setting, a user u knows $K(g)$ if and only if the user u belongs to group g. Using the standard properties of the PRF H, a user u can guess $H_{salt}(K(g))$ if and only if user u knows $K(g)$. Hence, given $T(KEK(f), g^i)$ = $KEK(f)$ + $H_{salt}(K(g), i)$, a user u can guess $KEK(f)$ if and only if user u knows $H_{salt}(K(g), i)$ $\rightsquigarrow$ user u knows $K(g)$ $\rightsquigarrow$ user u is a member

of group g. By the properties of an ideal PRF H, for all $i \neq j$, $H_{salt}(K(g), i)$ and $H_{salt}(K(g), j)$ appear random (and independent) to a user who does not know $K(g)$.

Let us assume that the induction hypothesis holds for d operators, for some $d \geq 0$. Let B_f be a monotone Boolean expression with $d+1$ operators. Then B_f can be expressed either as $B_f = A_1 \wedge A_2$ or $B_f = A_1 \vee A_2$, such that A_1 and A_2 have no more than d operators. If B_f is expressed as $A_1 \vee A_2$, $T(KEK(f), B_f) = T(KEK(f), A_1) \cup T(KEK(f), A_2)$. By our induction hypothesis, $T(KEK(f), A_1)$ and $T(KEK(f), A_2)$ are secure transformations. Hence, an unauthorized user cannot derive any useful information about $KEK(f)$ from either $T(KEK(f), A_1)$ or $T(KEK(f), A_2)$. Since $T(KEK(f), A_1)$ and $T(KEK(f), A_2)$ are computed independently, there is no joint information that can derived from $T(KEK(f), A_1)$ and $T(KEK(f), A_2)$.

If B_f is expressed as $A_1 \wedge A_2$, $T(KEK(f), B_f) = T(x_1, A_1) \cup T(x_2, A_2)$ such that $x_1 + x_2 = KEK(f)$. An unauthorized user either has no information about x_1 or x_2 or both (based on the induction hypothesis). Since $T(x_1, A_1)$ and $T(x_2, A_2)$ are computed independently, there is no joint information on x_1 or x_2 that can derived from $T(x_1, A_1)$ and $T(x_2, A_2)$. Since, $KEK(f) = x_1 + x_2$, an unauthorized user that does not know any information about x_1 or x_2 or both cannot guess any information about $KEK(f)$. Hence, for a monotone function B_f with $d+1$ operators, $T(KEK(f), B_f)$ is secure. By induction on the number of operators in B_f, T is a secure transformation for all monotone Boolean expressions B_f. ∎

Drawbacks. While the basic construction presents a secure transformation T, it has several drawbacks. First, the basic construction does not tolerate collusions among users. A collusion between two users u_1 and u_2 may result in *unauthorized privilege escalation*. For example, let us say that u_1 is a member of group g_1 and u_2 is a member of group g_2. By colluding with one another, users u_1 and u_2 would be able to access a file f with $B_f = g_1 \wedge g_2$, thereby violating the discretionary access control (DAC) model. Recall that in a DAC model, the set of files that is accessible to two colluding users u_1 and u_2 should be no more than the union of the set of files accessible to the user u_1 and the user u_2. Second, the key server needs to know $KEK(f)$ and B_f for all files in the system. In a static setting, wherein $KEK(f)$ and B_f do not change with time, this incurs heavy storage costs at the key server. In a dynamic setting, this incurs heavy communication, synchronization and consistency maintenance costs in addition to the storage cost. Note that in a dynamic setting, the key server has to maintain up to date information on $KEK(f)$ and B_f for all files in the system.

3.3 Collusion Resistant Construction

In this section, we present techniques to tolerate malicious collusions between users. The key problem with our basic construction (Section 3.2) is that the authorization information given to a user u_3 that belongs to both groups g_1 and g_2 (namely, $K(g_1)$ and $K(g_2)$) is simply the union of the authorization information given to a user u_1 that belongs to group g_1 (namely, $K(g_1)$) and to a user u_2 that belongs to group g_2 (namely, $K(g_2)$). We propose that when an user u joins a group g, it gets two pieces of authorization information $K(g)$ and $K(u, g)$. The key $K(u, g)$ binds user u to group g. However, using randomly chosen values for $K(u, g)$ does not scale with the number of users, since the group key management service and our key server would have to maintain potentially $|U| * |G|$ keys, where $|U|$ is the number of users and $|G|$ is the number of groups. We propose to mitigate this problem by choosing $K(u, g)$ pseudo-randomly. We derive $K(u, g)$ as $K(u, g) = H_{MK}(u, g)$, where MK is the master key shared between the group management service and the key server. For notational simplicity, we overload u and g to denote the u's user identifier and g's group identifier respectively.

Now, we modify the recursive definition of the transformation T described in Section 3.2 as follows:

$$
\begin{aligned}
T(x, u, \bigwedge_{i=1}^{n} A_i) &= \bigcup_{i=1}^{n} T(x_i, u, A_i) \text{ such that } \sum_{i=1}^{n} x_i = x \\
T(x, u, \bigvee_{i=1}^{n} A_i) &= \bigcup_{i=1}^{n} T(x, u, A_i) \\
T(x, u, g^i) &= x + H_{salt}(K(u, g), i)
\end{aligned}
$$

Theorem 3.2 *The transformation T described in Section 3.3 is secure and collusion resistant.*

Proof We first show that the transformation T is resilient to collusion amongst malicious users. We prove collusion resistance by showing that two users u_1 a member of group g_1 and u_2 a member of group g_2 cannot access a file f with $B_f = g_1 \wedge g_2$. Assuming an ideal PRF H, it is hard to guess any information about $K(u, g)$ from $K(u', g')$ if $u \neq u'$ or $g \neq g'$. The authorization information known to a user u_1 that is a member of group g_1 is $K(u_1, g_1)$ and that known to a user u_2 that is a member of group g_2 is $K(u_2, g_2)$. Hence, even when user u_1 colludes with user u_2, u_1 cannot access a file f with $B_f = g_1 \wedge g_2$ since it does not have any information about $K(u_1, g_2)$.

Second, we show that the $T(KEK(f), u, B_f)$ can be made publicly available (to all users and the SSP) without compromising the access control guarantees of the file system. Assuming an ideal PRF H, it is hard to guess any information about $K(u, g)$ from $K(u', g')$ if $u \neq u'$ or $g \neq g'$. Hence, this construction satisfies the base case of our inductive proof for theorem 3.1. The remaining arguments for the inductive proof remains the same as in theorem 3.1. Therefore, the key server can publicly disseminate $T(KEK(f), u, B_f)$ without verifying the identity of u and without verifying if u's membership indeed satisfies the monotone B_f. ∎

3.4 Key Encryption Keys

We have so far described techniques to securely transform and distribute key encryption keys. However, a major scalability bottleneck still remains in the system. The key server needs to know $KEK(f)$ and B_f for all files in the file system. This incurs not only heavy storage costs, but also incurs heavy communication costs to maintain the consistency (up to date) of $KEK(f)$ and B_f. In this section, we propose to circumvent this problem as follows. We propose to derive $KEK(f)$ as a function of B_f. Hence, when a user u requests for $T(KEK(f), u, B_f)$, the key server first computes $KEK(f)$ as a function of B_f. Then, it uses the derived value for $KEK(f)$ to construct the $T(KEK(f), u, B_f)$ as described in Section 3.3. In the following portions of this section, we present a technique to derive $KEK(f)$ from B_f. Our technique maintains the semantic equivalence of monotone Boolean expressions, that is, for any two equivalent but non-identical representations of a monotone Boolean function B_f and B'_f, $KEK(B_f)$ = $KEK(B'_f)$.

Preprocessing. Given a monotone Boolean expression B_f we normalize it as follows. We express B_f in a minimal conjunctive normal form (CNF) as $B_f = C_1 \wedge C_2 \cdots \wedge C_n$. $C_1 \wedge C_2 \cdots \wedge C_n$ is a minimal expression of B_f if for no $1 \leq i, j \leq n$ and $i \neq j$, $C_i \Rightarrow C_j$. Note that a monotone Boolean expression in its minimal form is unique up to a permutation on the clauses and permutation of literals within a clause. If not, let us suppose that $B_f = C_1 \wedge C_2 \cdots \wedge C_n = C'_1 \wedge C'_2 \cdots \wedge C'_{n'}$ be two distinct minimal CNF representations of B_f. Then, there exists C_i such that $C_i \neq C'_j$ for all $1 \leq j \leq n'$. Setting all the literals in C_i to `false` sets the expression B_f to `false`. Hence, for $C'_1 \wedge C'_2 \cdots \wedge C'_{n'}$ to be an equivalent representation, there has to exist C'_j such that the literals in C'_j is a proper subset of the literals in C_i. Then, setting all the literals in C'_j to `false` sets B_f to `false`. Hence, for $C_1 \wedge C_2 \cdots \wedge C_n$ to be an equivalent representation, there has to exist $C_{i'}$ ($i \neq i'$) such that the literals in $C_{i'}$ is a proper subset of the literals in C'_j. Hence, the literals in $C_{i'}$ is a proper subset of the literals in C_i, that is, $C_{i'} \Rightarrow C_i$ ($i \neq i'$). This contradicts the fact that $C_1 \wedge C_2 \cdots \wedge C_n$ is a minimal CNF representation of the monotone Boolean expression B_f. We normalize the representation of each clause C_i as $g_{i_1} \vee g_{i_2} \vee \cdots \vee g_{i_m}$ such that $i_j < i_{j+1}$ for all $1 \leq j < m$.

Deriving $KEK(f)$. We compute $KEK(f)$ recursively as follows:

$$\begin{aligned} KC(C_i) &= H_{MK}(i_1, i_2, \cdots, i_m) \text{ where } C_i = g_{i_1} \vee g_{i_2} \vee \cdots \vee g_{i_m} \text{ and } i_1 < i_2 < \cdots < i_m \\ KEK(B_f) &= H_{MK}(KC(C_1) \oplus KC(C_2) \oplus \cdots \oplus KC(C_n)) \text{ where } B_f = C_1 \wedge C_2 \wedge \cdots \wedge C_n \\ KEK(f) &= H_{MK}(KEK(B_f), salt) \end{aligned}$$

Note that MK is a master key used by the key server. The $salt$ value is an auxiliary attribute associated with the file f. The PRF H is neither commutative nor associative; hence, we impose an arbitrary total order on groups using their group number. The $\oplus$ operator is both commutative and associative; hence, the order of the clauses in B_f does not affect $KEK(B_f)$. Hence, given any two equivalent representations of a monotone Boolean function $B_f = B'_f$, our algorithm computes the same key encryption key.

Security Analysis. It is easy to see that a user u who is authorized to access file f can easily recover $KEK(f)$ from $T(KEK(f), u, B_f)$. Let us suppose that a user u is not authorized to access file f. The user can present incorrect inputs since, the inputs are not authenticated by the key server. Recall that the key server accepts three inputs $salt$, u and B_f. Let us suppose that a user u sends an incorrect input u'. By the property of the secure transformation function T (Section 3.3), the user u cannot guess $KEK(f)$ from $T(KEK(f), u', B_f)$. Even if the users u and u' were to collude, we have shown in Section 3.3 that they can obtain $KEK(f)$ if and only if either u or u' is indeed authorized to access the file f. Let us suppose that a user u sends an incorrect input B'_f. By the description of our key derivation algorithm in this Section, using an incorrect B'_f results in an incorrect $KEK'(f)$. Indeed the properties of the PRF H ensures that the user u cannot guess $KEK(f)$ from $KEK'(f)$. The same argument also applies if the user u were to send an incorrect input $salt'$. Hence, given one or more outputs from the key server, a user u can construct $KEK(f)$ if and only if the user u is authorized to access the file f, that is, $B_f(G_u)$ = `true`.

The key server exports only one interface that accepts the file's $salt$, u and B_f as inputs and returns a secure transformation of $KEK(f)$, namely, $T(KEK(f), u, B_f)$ as output. The key server does not have to interact with either the group key management service to maintain $KEK(f)$ and B_f for all files f or G_u and $K(u, g)$ for all users u and groups g. This large minimizes the storage costs, communication costs, synchronization and consistency management costs in a dynamic setting and largely improves the scalability of the key server.

3.5 Dynamic Group Membership

So far, we have explored a static setting, wherein, the monotone Boolean expression (B_f) used to specify access control on the file is static and the group membership of users are static. The former is easier to deal with. Let us suppose that a content owner changes the access control on a file f from B_f to B'_f. The key server would compute $KEK'(f)$ and sends $T(KEK'(f), u, B'_f)$ to the content owner u. The content owner u can now extract $KEK'(f)$ from its secure transformation $T(KEK'(f), u, B'_f)$. Note that any user $u' \neq u$ cannot obtain any useful information from $T(KEK'(f), u, B'_f)$. Now, the content owner u, encrypts the file key $K(f)$ using its new key encryption key $KEK'(f)$. Note that we do not change $K(f)$ (or re-encrypt the file with a new file encryption key) when B_f changes. We use a lazy revocation mechanism wherein, the next write operation on the file chooses a new file encryption key $K'(f)$ to encrypt the file. It also encrypts $K'(f)$ with the new key encryption key $KEK'(f)$ and stores it along with other attributes for file f. While a user u' whose access to file f has been revoked, can determine whether the contents of a file f has changed or not, it cannot decrypt updates (new writes) to the file f.

In this section, we present techniques to flexibly handle dynamic group membership. To illustrate the flexibility, we permit temporal group memberships of the form: User u is a member of group g from 10 AM, January 12^{th} 2006 to 4 PM, March 18^{th} 2006. We permit arbitrary time granularity at the expense of higher key management costs. We use a novel key tree based algorithm to efficiently support dynamic group membership up to any arbitrary time granularity. Let δt seconds denote the smallest time granularity of interest. Let time equal to t denote the t^{th} time unit in an year, where one unit time = δt seconds. We associate a key $K^t(u, g)$ as the authorization key that denotes the fact that user u belongs to group g at time t. Hence, to authorize a user for T time units we require T authorization keys. Using, δt = 1 second an authorization for one year would involve $3.15 * 10^7$ authorization keys. We reduce the authorization cost by building a key tree. The key tree reduces the number of authorization keys to at most $2 \log_2 T - 2$ (for all $T > 2$). Hence, with no more than 48 authorization keys one can handle dynamism at the granularity of one second.

The key idea behind our technique is to associate an authorization key $K^{a,b}(u, g)$ with the time duration (a, b). The authorization key tree satisfies the following properties:

- Given $K^{a,b}(u, g)$ it computationally easy to derive a key $K^{a',b'}(u, g)$, if $a \leq a' \leq b' \leq b$.
- Given $K^{a,b}(u, g)$ it should be computationally infeasible to derive a key $K^{a',b'}(u, g)$, if $a' < a \vee b' > b$.

At any given time t, the key server uses $K^{t,t}(u, g)$ as the authorization key $K(u, g)$ in its transformation function (Section 3.3). Note that the time interval (t, t) denotes the time instant t. The key server includes the timestamp t along with the secure transformation of the key encryption key in its response to the user. A user u that has authorization for time interval (a, b) and thus possesses $K^{a,b}(u, g)$ can derive $K^{t,t}(u, g)$ if and only if $a \leq t \leq b$. Hence, the user u can extract $KEK(f)$ from $T(KEK(f), u, B_f)$ at time t if and only if G_u satisfies B_f at time t.

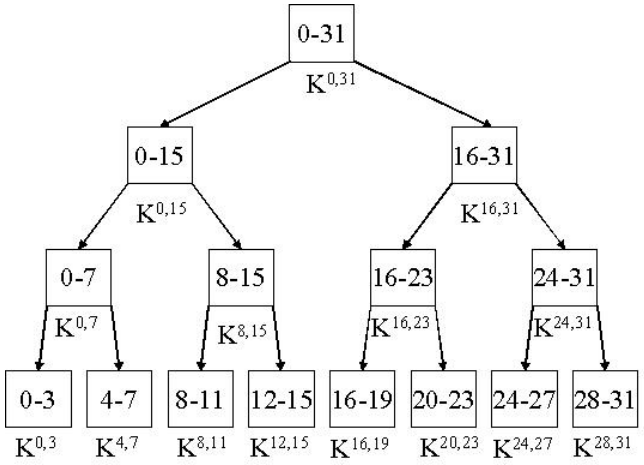

Figure 2: Authorization Key Tree

δt	Avg / Max Num Keys	Avg / Max Compute Time (μs)
one month	2 / 6	1.82 / 5.46
one week	3 / 10	2.73 / 9.10
one day	5 / 16	4.55 / 14.56
one hour	7 / 26	6.37 / 23.66
one minute	10 / 38	9.10 / 34.58
one second	13 / 48	11.83 / 43.68
one millisecond	18 / 68	16.38 / 61.88

Table 2: Number of Keys and Computation Time Vs Time Granularity

Authorization Key Tree. We construct the authorization key tree as follows. Each element in the key tree is associated with a time interval. At the root of the key tree is $(0, T_{max})$, where T_{max} denote the number of time units in one year. Each element (a, b) in the key tree has two children $(a, \frac{a+b}{2})$ and $(\frac{a+b}{2} + 1, b)$. We associate a key $K^{a,b}(u, g)$ with every element (a, b) in the key tree. The keys associated with the elements of the key tree are derived recursively as follows: $K^{a,\frac{a+b}{2}}(u, g) = H(K^{a,b}(u, g), 0)$ and $K^{\frac{a+b}{2}+1,b}(u, g) = H(K^{a,b}(u, g), 1)$, where H is a one-way hash function like MD5 [26] or SHA1 [10]. The root of the key tree has a key $K^{0,T_{max}}(u, g) = H_{MK}(u, g)$, where MK is the key server's master key. Observe, that given $K^{a,b}(u, g)$ one can derive all keys $\{K^{t,t}(u, g)$: $a \leq t \leq b\}$. Also, deriving the key $K^{t,t}(u, g)$ for any $a \leq t \leq b$ from $K^{a,b}(u, g)$ requires no more than $\log_2(b - a)$ applications of the hash function H. Figure 2 illustrates the construction of our key tree assuming T_{max} = 31 time units. We derive $K^{0,31}(u, g) = H_{MK}(u, g)$. Then, we compute $K^{0,15}(u, g) = H(K^{0,31}(u, g), 0)$ and $K^{16,31}(u, g) = H(K^{0,31}(u, g), 1)$. One can recursively extend this definition to any arbitrarily small time granularity. Given any arbitrary time range (beg, end), one can show that the range can be partitioned into no more than $2 \log_2 T_{max} - 2$ elements in the key tree. For example, given a time interval $(8, 19)$, we partition the time interval into two sub intervals $(8, 15)$ and $(16, 19)$ (see Figure 2). We authorize user u to be a member of group g from the 8^{th} to the 19^{th} time unit by issuing keys $K^{8,15}(u, g)$ and $K^{16,19}(u, g)$.

Cost Analysis. In general, if one uses a a-ary key tree ($a \geq 2$), any range can always be subdivided into no more than a $(\log_a(T_{max}) - 1)$ sub-ranges ($T_{max} > a$). One can show that this is a monotonically increasing function in a (for $a \geq 2$) and thus has a minimum value when $a = 2$. Hence, a binary key tree is optimal and it requires no more than $2 \log_2(T_{max}) - 2$ sub-ranges for any given range. In addition, one can also show that a = 2 minimizes the average number of ranges into which any randomly chosen range need to be subdivided $\frac{1}{2}$*$(a - 1)\log_a(T_{max})$. However, as a increases the height of the key tree ($\log_a(T_{max})$) decreases, that is, the cost of key derivation decreases monotonically with a. Our experiments show that the cost of key derivation is very small. Each application of the one-way hash function H takes less than 1μs on a 900 MHz Intel Pentium III processor. Therefore, we chose to use a binary key tree (a = 2) that minimizes the key management cost. Table 2 shows the average & the maximum number of keys and computation time required for different values of δt for a time interval of one year using a binary authorization key tree ($a = 2$).

4 Implementation

In this section, we present a sketch of our implementation of the SAS model using a distributed cryptographic write-once versioning file system.

Distributed File System. We have implemented our storage service provider (SSP) on a publicly available Java code for a distributed file system [30] operating atop of the Chord lookup protocol [32]. Given the file name f_{name}, the Chord lookup protocol returns the IP-address of the node that hosts the file f. Additionally, if the file is replicated, the lookup protocol can be used to identify a set of nodes that host the replicas of file f.

Notation	Algorithm
H	SHA1
H_K	HMAC-SHA1
E	AES-128-CBC

Table 3: Cryptographic Primitives

Parameter	Default	Description
nf	10^7	number of files
nu	1000	number of users
ng	32	number of groups
nug	zipf(1, 10)	number of groups per user
nc	zipf(2, 4)	number of clauses in B_f
nl	zipf(2, 4)	number of literals per clause
δt	1	time granularity (seconds)

Table 4: Parameters

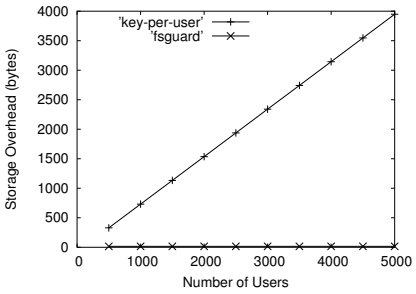

Figure 3: Storage Overhead

Cryptographic File System. All files are encrypted and authenticated using message authentication codes (MAC) before they are stored at the SSP. For performance reasons, files are divided into data blocks of size 8 KB. We use AES-128 bits in CBC mode for encryption and HMAC-SHA1 for computing MACs.

Write Once Versioning File System. A write once file system [29] does not allow a file to be changed after it is written the first time. The file system never overwrites a file; instead it makes a new copy (version) of the file each time the file is written. This property of the file system ensures that the file system can recover from illegal writes to a file by unauthorized users with the aim of corrupting the file.

Lazy Re-encryption. When a file key is changed, we do not immediately re-encrypt the file. The next time an authorized user writes to the file, it uses the new file key to encrypt the file. Note that even after the access to a user is revoked, the user could still access older versions of the file. This does not affect file confidentiality, since the user could have very well made a copy of the file before its access was revoked. However, the newer updates to the file would be unintelligible to the revoked user.

Key Server. We have implemented the key server as a stand alone entity. The key server exports only one interface that accepts the file's $salt$, u and B_f as inputs and returns a secure transformation of $KEK(f)$, namely, $T(KEK(f), u, B_f)$ as output. Being a central component in our file system, the key server could become a performance bottleneck. However, our experiments show that the key server operations are computationally inexpensive, thereby, allowing it to scale to a large number of users, files and groups.

File Client. We modified the basic file client to support interactions with the key server. We used AspectJ [11] to modify the functionality of `read`, `write`, and `create` calls to the file system. A file read operation by the client proceeds as follows: (i) The user fetches the encrypted file block from the SSP along with attributes $salt$ and $E_{KEK(f)}(K(f))$. (ii) The user checks if the file's key $K(f)$ is available in the local cache. If yes, the user uses the MAC on the file block to check if cached value of $K(f)$ is indeed its most recent value. If yes, the user updates the last access time for the cached entry corresponding to $K(f)$ in its local cache and proceeds to step (v). If $K(f)$ is not available in the cache or the cached value has become stale, the user u sends the file's $salt$, u and B_f to the key server. (iii) The key server computes $KEK(f)$ from B_f and $salt$ (see Section 3.4) and sends a secure transformation of $KEK(f)$, namely, $T(KEK(f), u, B_f)$ and the timestamp t to the user u (see Section 3.3). (iv) The user extracts $KEK(f)$ from $T(KEK(f), u, B_f)$ and the timestamp t (see Section 3.5). The user now uses $KEK(f)$ to obtain $K(f)$ from $E_{KEK(f)}(K(f))$. The user adds $K(f)$ into its local file cache. We use a simple LRU based cache replacement strategy to maintain the key cache. (v) Now, the user can use $K(f)$ to read the file block.

The `write` and `create` calls to the file system proceed analogously. They involve an additional step wherein the user gets a new key $KEK'(f)$. For a write operation, the key server chooses a new random $salt'$ and derives $KEK'(f)$ from B_f and $salt'$. The user chooses a new random file encryption key $K'(f)$ to encrypt the updated file blocks and stores $E_{KEK'(f)}(K(f))$ as an attribute with the file block. Using a new file encryption key $K'(f)$ ensures that the updates to file f is unintelligible to users whose access to file f has been revoked.

5 Evaluation

In this section, we present a concrete evaluation of our prototype implementation. We ran our prototype implementation on eight machines (550 MHz Intel Pentium III Xeon processor running RedHat Linux 9.0) connected via a high

speed 100 Mbps LAN. We used six machines to operate as the file servers, one machines to operate as the client, and one machine operates as the key server.

We compare our approach with two other approaches: key-per-user and key-per-file approach (see Section 3.1). We evaluate the performance of our proposal using four performance metrics: number of keys per user, storage cost at SSP, communication cost for various file system operations (file access, file's access control expression update, user's group membership update), and computation cost for various file system operations (file access, file's access control expression update, user's group membership update). We perform trace driven evaluations using the SPECsfs workload generator [1] of our approach to study the scalability of the key server and the performance overhead of our approach on a cryptographic file system. We used a synthetic file system with 10 million files, 1000 users, and 32 user groups. We assume that the group popularity follows a Zipf distribution [27], that is, the number of users that are a member of group i ($1 \leq i \leq 32$) is proportional to $\frac{1}{i}$. We assume that the number of clauses in any monotone B_f follows a Zipf distribution between 2 to 4 and the number of literals per clause follows a Zipf distribution between 2 to 4. Table 4 summarizes our main file system parameters.

5.1 Storage, Computation and Communication Costs

5.1.1 Number of Keys per User

In our first experiment, we measure the average number of keys maintained by a user using the three approaches. As the number of keys per user increases, so does the cost of managing those keys. Also, requiring a user to maintain a large number of keys increases the risk of one more keys being lost or accidentally leaked to an adversary. The key-per-user approach requires the user to store only one key. Our approach requires the user to store one key per group; we found that the average number of keys per user was 3.78.

The key-per-file approach requires the user to store one key per file that it is permitted to access. The average number of keys per user in this case is about $4.2 * 10^5$. [22, 24] propose techniques to cluster files (termed file groups) based on their similarity. One can cluster files based on their access control expression B_f: all files in a cluster have identical (equivalent) B_f. We found that amongst 10 million files, there were $1.3 * 10^5$ unique monotones B_f: hence, we had $1.3 * 10^5$ file clusters with 1-337 files per cluster. We found that even with the clustering mechanism, the number of keys per user was about $2.3 * 10^4$. Because of the practical infeasibility of the key-per-file approach, the rest of our experiments focus exclusively on the key-per-user approach and our proposal.

5.1.2 Storage Cost as SSP

In our second experiment, we study the storage overhead at the SSP for storing additional file attributes. The key-per-user approach requires that we store $E_{K(u)}(K(f))$ per file block for all users u that is permitted to access the file f. Our approach stores only attribute $E_{KEK(f)}(K(f))$ (16 Bytes). Under the default settings described in Table 4, we found that the average number of users that can access a file was 45.7. Hence, each file block (8 KB) stored on the SSP the key-per-user approach incurs about 45.7 * 16 Bytes = 731.2 Bytes overhead (8.9%), while our approach (`fsguard`) incurs only a 16 Byte overhead (0.2%). As the number of users increase, the size of attributes stored with a file increases. Figure 3 shows the average size of a file's attribute as the number of users varies. Observe that as the numbers of users become 5000, the attribute size is about 4 KB. Using 8 KB file blocks, at least 50% of the storage space on the SSP would be expended on storing file attributes.

5.1.3 Communication Cost

In our third experiment, we measure the communication cost for three important operations: file access (read/write), update on a file's access control expression, and update on a user's group memberships.

File Access (read/write). A file access in the key-per-user approach does not involve any interaction between the user and the key server. The user fetches the file block and $E_{K(u)}(K(f))$ from the SSP and performs read/write operations on the block. On the other hand, file access in our approach requires the user to interact with the key server if the file encryption key $K(f)$ is not available in the user's local key cache. Observe that the communication cost between the user and the key server is $O(|B_f|)$, where $|B_f|$ denotes the number of literals in the monotone expression B_f. For example, $|(g_1 \vee g_2) \wedge (g_1 \vee g_3)|$ = 4. Figure 4 shows the communication cost between the user and the key server

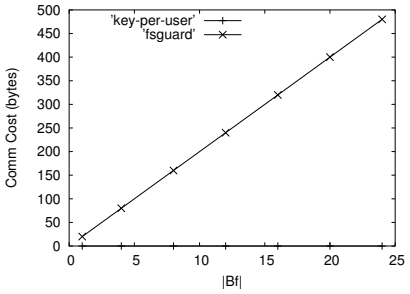

Figure 4: Communication Cost: File Access

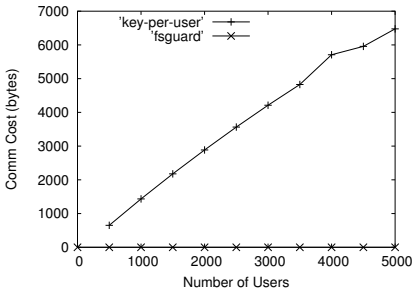

Figure 5: Communication Cost: File Access Control Expression Update

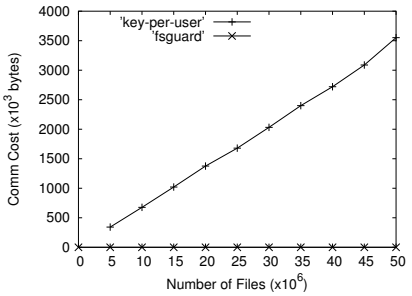

Figure 6: Communication Cost: User Group Membership Update

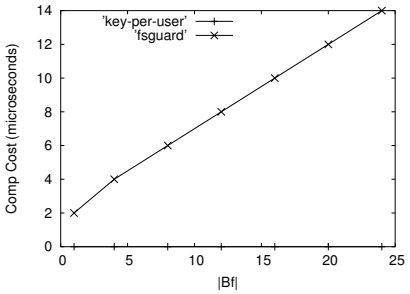

Figure 7: Computation Cost: File Access

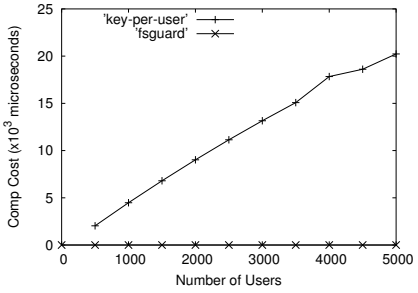

Figure 8: Computation Cost: File Access Control Expression Update

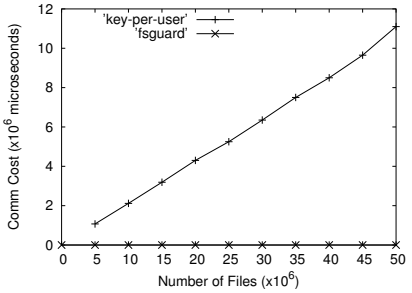

Figure 9: Computation Cost: User Group Membership Update

for different values of $|B_f|$. Observe that even for complex (large) monotones, the communication cost is about a few hundred bytes.

File Access Control Expression Update. Let B_f and B'_f denote the old the new access control expression for file f. In the key-per-user approach, the key server has to determine the set of users U and U' whose group membership satisfies the expression B_f and B'_f respectively. For all $u \in U' - U$, the key server has to add $E_{K(u)}(K(f))$ to the file f's attribute. For all $u \in U - U'$, the key server has to remove $E_{K(u)}(K(f))$ from the file f's attribute. On the next write operation on file f, the key server needs to update $K(f)$ to a new key $K'(f)$ and consequently add $E_{K(u)}(K'(f))$ for all $u \in U'$ as attributes of file f. Note that the old attributes $E_{K(u)}(K(f))$ for all $u \in U$ can be deleted by the SSP. Using our approach, an update to the file's access control expression does not incur any communication cost. Recall that the interface exported by the key server operates on B_f rather than f itself. Figure 5 shows the communication cost between the key server and the SSP as nu, the number of users vary. Observe that as the number of users increase, the communication cost on the key server increases. This largely limits the scalability of the key server with the number of users in the file system. Observe from Figures 4 and 5 that an update on a file's access control expression costs about 1000 times the cost of a file access incurred by our approach.

User Group Membership Update. Let us suppose that a user u's group membership changed from G to G'. In the key-per-user approach, the key server has to determine the set of files F and F' whose access control expression is satisfied by group membership G and G' respectively. For all files $f \in F' - F$, the key server has to add $E_{K(u)}(K(f))$ to the file f's attribute. For all files $f \in F - F'$, the key server has to remove $E_{K(u)}(K(f))$ from the file f's attribute. On the next write operation on any file $f \in F - F'$, the key server needs to update $K(f)$ to a new key $K'(f)$. Consequently the key server has to add $E_{K(u')}(K'(f))$ as an attribute for the file f for all users u' that can access file f. Using our approach, an addition to a user's group membership requires an interaction with the group key management service. Revocation of a group membership does not require any communication using our algorithm in Section 3.5. Figure 6 shows the communication cost as nf, the number of files vary. Using the key-per-user approach, the communication cost incurred in updating one user's group membership grows linearly with the number of files in the system and is of the order of several megabytes. This largely limits the scalability of the key server with the number of files in the system. Observe from Figures 4 and 6 that an update on a user's group membership costs about million times the cost of a file access incurred by our approach.

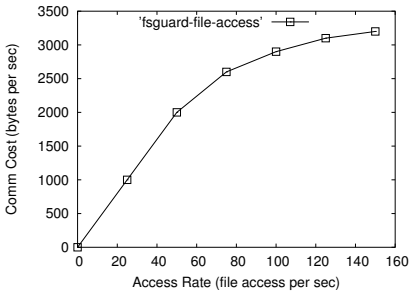

Figure 10: Key Server Communication Cost

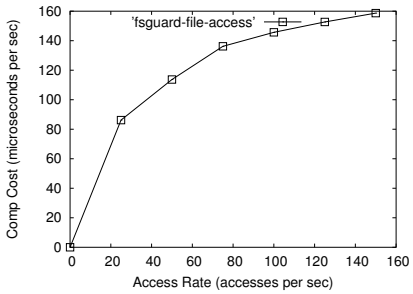

Figure 11: Key Server Computation Cost

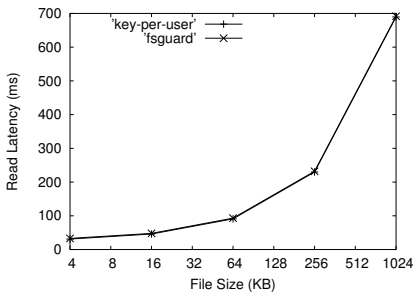

Figure 12: File Read/Write Latency

5.1.4 Computation Cost

In our fourth experiment, we measure the computation cost for three important operations: file access (read/write), update on a file's access control expression, and update on a user's group memberships. The computation cost is divided between the key server and the user. We computation cost is expressed in seconds as measured using a 550 MHz Intel Pentium III Xeon processor running RedHat Linux 9.0. Figures 7, 8 and 9 shows the computation cost at the client and the key server for the three operations listed above. Similar to the communication cost, our approach incurs computation cost only for file read/write operations. Further, this computation cost is incurred only if the file's key is not available in the user's cache. The key-per-user approach imposes heavy computation cost when a file's access control expression is updated or when a user's group membership is updated. An update on a file's access control expression costs about 1000 times the cost of a file access incurred by our approach; an update on a user's group membership costs about million times the cost of a file access incurred by our approach.

5.2 Trace based Evaluation

5.2.1 Key Server Scalability

We have so far compared the computation and communication cost incurred by our approach and the key-per-user approach. In this section, we use a trace based evaluation using SPECsfs [1] workload generator to compare the scalability of the key server between our approach and the key-per-user approach. In our approach, the load on the key server depends on the rate of file accesses, and the actual file access pattern. The actual access pattern determines the efficacy of user's local key cache. In the key-per-user approach, the load on the key server depends on the file access expression update rate, the user group membership update rate, the number of files and the number of users. Dependence on the number of files and users reduces the scalability of the key-per-user approach. In addition, the key-per-user approach needs to interact with the group key management service to maintain up to date information on file access control expressions and user group memberships.

On the other hand, our approach does not require any active state to be maintained by the key server, other than the 16 Byte master key MK. The key server exports only one interface that accepts $salt$, u and B_f as inputs and returns a secure transformation of $KEK(f)$, namely, $T(KEK(f), u, B_f)$ as output. The computation of $KEK(f)$ as described in Section 3.4 does not require any state maintenance at the server. The derivation of $KEK(f)$ and its secure transformation $T(KEK(f), u, B_f)$ being computationally inexpensive can be computed on the fly by the key server. Our approach does not require any synchronization or consistency update messages between the key server and the group key management service. Further, our design permits the key server to be replicated on demand. Observe that any two key server replicas can operate without interacting with one another or the group key management service as long as they share the master key MK. Using the key-per-user approach, the key servers cannot be replicated easily. This is because the key server has to operate in synch with one another and the group key management service to maintain the consistent and up to date information on all user's group membership and all file's access control expressions.

Figures 10 and 11 show the communication and computation cost on the key server as the file access rate is varied using the SPECsfs [1] workload generator. We use a 16 KB key cache at the user to cache file encryptions keys using a LRU based cache replacement strategy. Note that a 16 KB key cache can hold about one thousand file encryption keys.

As the Figures 10 and 11 indicate, the communication and computation cost at the key server grows sub-linearly with the file access rate. Temporal locality in the set of files accessed by a user ensures that most of the time, the required file encryption keys are available in the local user's key cache. From Figures 4-11 one can conclude that our approach incurs lower load on the key server even if the rate of file access control expression update is one-thousandth of the file access rate and if the rate of user group membership update rate is one-millionth of the file access rate.

5.2.2 User Latency

In this experiment, we compare the latency of file read/write operations with the key-per-user approach. The key-per-user approach does not require any interaction with the key server to read/write a file in a static setting. Our approach requires interaction with the key server for read/write operations on a file only if the updated value of the file's key is not available in the user's local key cache. For a more detailed description of the file client's read/write operation see Section 4. However, this may increase the latency for read/write operations as perceived by the user. Fortunately, one can overlap the key server operations with the file read/write operation at the storage service provider (SSP). Figure 12 shows the average latency of a read operation on file blocks of different sizes using our approach and the key-per-user approach using the SPECsfs [1] workload generator. Observe that the operational latency on a file read/write operation is very small; about 4.3% for an 8 KB file block. Hence, the operational latency experienced by a user is only marginally higher than the key-per-user approach in a static setting. In a dynamic setting, our approach incurs no additional overhead on the key server; however, the key-per-user approach incurs heavy overheads on the key server as the file access control expression update rate or the user group membership update rate increases.

6 Discussion

6.1 Issues

We have proposed secure, efficient and scalable key management algorithms to enforce discretionary access control using monotone access structures on a cryptographic file system. However, our proposal has a few drawbacks. The key server and the group key management services being centralized units are a target for host compromise and denial of service attacks (DoS). A host compromise attack on the key server can reveal file encryption keys to an unauthorized user. By design, our key server and the group key management service share the master key MK. Compromising the key server, allows an attacker to issue users group membership authorizations. Recall that $K^{a,b}(u, g)$ is derived from the master key MK, the user ID u, the group ID g, and the time interval (a, b). Fortunately, our design of the key server is very simple: the key server exports one simple and stateless interface that accepts $salt$, u and B_f as inputs and returns $T(KEK(f), u, B_f)$ as output. Also, the key server does not have to communicate either with the SSP or the group key management service. Having a simple key server permits one to formally verify its correctness, thereby, making it hard for an attacker to exploit any vulnerability in its implementation.

A DoS attack on the key server can hamper the file access rate to legitimate users. Fortunately, our key server can be easily replicated (on demand). As long as the attacker does not bring down all machines hosting the key server, the legitimate users would still be able to perform file access. Further, the interface exported by the key server is stateless and incurs very low computation & communication costs. Hence, it would be hard to launch a DoS attack on the key server with the goal of exhausting its main memory resources, processing and network resources. In comparison, a SYN flooding attack [31] exploits the fact that a server maintains an active state for open TCP connections; a DoS attack on SSL (secure socket layer) exploits the fact that a server has to perform a computationally intensive task (verify a digital signature) [33]. Even if the key server were resilient to direct DoS attacks, it could be vulnerable to attacks from "the below". For instance, if the key server operates on TCP, any DoS attack on TCP can harm the key server. Given the stateless and connectionless design of our key server, one could host the key server on a light-weight UDP server, thereby, making it even more resilient to DoS attacks.

6.2 Related Work

Advances in the networking technologies have triggered several networking services such as: 'software as a service' also referred to as the application service provider (ASP) model [16], 'database as a service' (DAS) [13] that permits organizations to outsource their DBMS requirements, and 'storage as service' (SAS) model. The SAS model inherits

all the advantages of the ASP model, indeed even more, given that a large number of organizations have their own storage systems. This model allows organizations to leverage hardware and software solutions provided by the service providers, without having to develop them on their own, thereby freeing them to concentrate on their core businesses. However, implementing flexible access control mechanisms and protecting the confidentiality from a storage service provider (SSP) has been a critical problem in the SAS model.

Cryptographic file systems like CFS [4], TCFS [7], CryptFS [36], NCryptFS [35], Farsite [2], StegFS [21], cryptographic disk driver [9] and cooperative file system [8] permit the file data to be kept confidential from the SSP. These file systems were designed with the goal of data confidentiality, while balancing scalability, performance and convenience. However, these systems were not designed with the goal of supporting flexible access control policies.

Cryptographic access control [14] make it possible for one to rely exclusively on cryptography to ensure confidentiality and integrity of data stored in the system. Data are encrypted as the applications store them on a server, which means that the storage system only manages encrypted data. Read/Write access to the physical storage device is granted to all principals (only those who know the key are able to decrypt the data). Cryptographic access control has been deployed to maintain secrecy in group key management protocols [34, 5, 6, 25]. However, the access control policies that could be specified using cryptographic access control mechanisms were naive and inflexible. In this paper we have proposed techniques to implement monotone structure based access control policies in a cryptographic file system.

7 Conclusion

In this paper we have presented − secure, efficient and scalable mechanisms to enforce discretionary access control using monotone access structures on a cryptographic file system. We have presented key derivation algorithms that guarantee that a user who is authorized to access a file, can efficiently derive the file's encryption key; while, it is computationally infeasible for a user to guess the encryption keys associated with the files that she is not authorized to access. We have also presented concrete algorithms to support dynamic access control updates & revocations. We have also presented a prototype implementation of our proposal on a distributed file system. A cost based evaluation of our system showed that our approach incurs lower key management, storage, communication and computation cost when compared to the key-per-user and key-per-file approach. A trace driven evaluation of our prototype showed that our algorithms meet the security requirements while preserving the performance and scalability of the file system.

References

[1] SPEC SFS (system file server) benchmark. http://www.spec.org/osg/sfs97r1.

[2] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th International Symposium on OSDI*, 2002.

[3] J. Benaloh and J. Leichter. Generalized secret sharing and monotone functions. In *Proceedings of CRYPTO*, 1988.

[4] M. Blaze. A cryptographic file system for unix. In *Proceedings of ACM CCS*, 1993.

[5] R. Canetti, J. Garay, G. Itkis, and D. Micciancio. Multicast security: A taxonomy and some efficient constructions. In *Proceedings of the IEEE INFOCOM, Vol. 2, 708-716*, 1999.

[6] R. Canetti, T. Malkin, and K. Nissim. Efficient communication-storage tradeoffs for multicast encryption. In *Advances in Cryptology - EUROCRYPT. J. Stem, Ed. Lecture Notes in Computer Science, vol. 1599, Springer Verlag, pp: 459-474*, 1999.

[7] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of transparent cryptographic file system for unix. In *Proceedings of Annual USENIX Technical Conference*, 2001.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM SOSP*, October 2001.

[9] R. Dowdeswell and J. Ioannidis. The cryptographic disk driver. In *Proceedings of Annual USENIX Technical Conference*, 2003.

[10] E. Eastlake. US secure hash algorithm I. http://www.ietf.org/rfc/rfc3174.txt, 2001.

[11] Eclipse. Aspectj compiler. http://eclipse.org/aspectj.

[12] FIPS. Data encryption standard (DES). http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[13] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proceedings of 18th IEEE ICDE*, 2002.

[14] A. Harrington and C. Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the 8th ACM SACMAT*, 2003.
[15] HP. Data center services. http://h20219.www2.hp.com/services/cache/114078-0-0-225-121.aspx.
[16] IBM. Application service provider business model. http://www.redbooks.ibm.com/abstracts/sg246053.html.
[17] IBM. IBM datacenter scalable offering. http://www-03.ibm.com/servers/eserver/xseries/windows/datacenter/scalable.html.
[18] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. http://www.faqs.org/rfcs/rfc2104.html.
[19] B. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sceinces and Systems, pp: 437-443*, 1971.
[20] Mathpages. Generating monotone boolean functions. http://www.mathpages.com/home/kmath094.htm.
[21] A. D. McDonald and M. G. Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding, pp: 462-477*, 1999.
[22] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM*, 1997.
[23] NIST. AES: Advanced encryption standard. http://csrc.nist.gov/CryptoToolkit/aes/.
[24] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe system. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
[25] A. Perrig, D. Song, and J. D. Tygar. ELK: A new protocol for efficient large group key distribution. In *Proceedings of IEEE Symposium on Security and Privacy*, 2001.
[26] R. Rivest. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, 1992.
[27] C. roadknight, I. Marshall, and D. Vearer. File popularity characterization. In *Proceedings of the 2nd Workshop on Internet Server Performance*, 1999.
[28] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. In *IEEE Computer, Vol. 29, No. 2*, 1996.
[29] D. S. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Deciding when to forget in the elephant file system. In *Proceedings of 17th ACM SOSP*, 1999.
[30] A. Singh and M. Srivatsa. Apoidea: Decentralized P2P web crawling. http://disl.cc.gatech.edu/Apoidea/.
[31] V. A. Siris and F. Papagalou. Application of anomaly detection algorithms for detecting SYN flooding attacks. In *Proceedings of IEEE Globecom*, 2004.
[32] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, August 2001.
[33] A. Stubblefield and D. Dean. Using client puzzles to protect tls. In *Proceedings of 10th USENIX Security Symposium*, 2001.
[34] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. In *IEEE/ACM Transactions on Networking: 8, 1(Feb), 16-30*, 2000.
[35] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convinient cryptographic file system. In *Proceedings of Annual USENIX Technical Conference*, 2003.
[36] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.