# A Selective Encryption Approach to Fine-Grained Access Control for P2P File Sharing


Aditi Gupta
Department of Computer Science
Purdue University
aditi@purdue.edu

Salmin Sultana
Department of Electrical and Computer Engineering
Purdue University
ssultana@purdue.edu

Michael Kirkpatrick
Department of Computer Science
Purdue University
mkirkpat@cs.purdue.edu

Elisa Bertino
Department of Computer Science
Purdue University
bertino@cs.purdue.edu



***Abstract*—As the use of peer-to-peer (P2P) services for distributed file sharing has grown, the need for fine-grained access control (FGAC) has emerged. Existing access control frameworks use an all-or-nothing approach that is inadequate for sensitive content that may be shared by multiple users. In this paper, we propose a FGAC mechanism based on selective encryption techniques. Using this approach, the owner of a file specifies access control policies over various byte ranges in the file. The separate byte ranges are then encrypted and signed with different keys. Users of the file only receive the encryption keys for the ranges they are authorized to read and signing keys for the ranges they are authorized to write. We also propose an optional enhancement of the scheme where a file owner can hide location of the file. Our approach includes a key distribution scheme based on a public key infrastructure (PKI) and access control vectors. We also discuss how policy changes and file modifications are handled in our scheme. We have integrated our FGAC mechanism with the Chord structured P2P network. In this paper, we discuss relevant issues concerning the implementation and integration with Chord and present the performance results for our prototype implementation.**


## I. Introduction

File sharing systems based on peer-to-peer (P2P) networks are very popular because of their ability to distribute the burden of storage across a number of nodes. However, if they are to gain widespread adoption for enterprise systems and in contexts like social networks, proper access control frameworks must be defined. In existing file sharing applications, access control is either absent or enforced at a coarse granularity, such as at the file-level. Such an all-or-nothing mechanism is inappropriate for sharing files containing sensitive data. There is a need for secure and efficient mechanisms to enforce fine-grained access control (FGAC) that protects *portions* of sensitive files while capitalizing on the strengths of P2P systems.

As an example, consider a company with a number of government contracts. The company uses an internal P2P-based distributed database to organize highly sensitive information related to the employees assigned to these contracts. For instance, this database could include contact information, security clearances, and a history of previous projects. The nature of these contracts necessitates that access to information be assigned on a need-to-know basis. As such, employees should have access only to the records of others assigned to the same project. Furthermore, each employee must be allowed to modify portions of his own record (e.g., contact information), but restricted from modifying everything else. Consequently, read and write permissions to portions of the file must be handled carefully.

Existing P2P access control mechanisms operate at a coarse granularity, typically only on the file itself. Proposed FGAC approaches for P2P systems either enforce access control constraints at file block level or assume particular file structures, such as XML-based structures. We aim at a finer level of granularity where access control policies can be specified on arbitrary contiguous byte ranges of the file. We propose such a FGAC scheme based on *selective encryption* that allows content owners to specify fine-grained policies for unstructured files; under this scheme separate portions are encrypted with different keys. The owner can grant a user read access to a specific set of file byte ranges by providing him with the encryption key used for these ranges. Each of these portions is also signed with a different key. This signature is computed using a private key which is given to only to those users who have been granted write access. We use signatures to detect unauthorized modifications and ensure integrity of each file portion. Notice that, in our approach, it is not mandatory to encrypt the entire file. If a file portion can be made public, then the owner can use a special policy to permit all accesses. Thus, we selectively encrypt only the sensitive portions of a file, and leave the unprotected portions in the clear. Our FGAC design is flexible enough that it can be used to implement previous approaches that enforce access control at the level of files, blocks, or components in structured XML documents.

Once encrypted, a file along with its signatures is stored on a node of a P2P system. Our approach requires some cryptographic meta-data (of limited size) which is also stored on a peer node in P2P. This meta-data information is used for derivation of encryption and signature keys by the authorized users. In our prototype implementation, we use Chord [1] as the underlying storage mechanism.

As an additional level of security, we provide the file owner with an option to hide the location of a file. Anonymizing the location of a file defends against various attacks by adding an extra layer of protection. It protects against denial of service attack since the attacker does not know which peer stores a particular file, and hence does not know which peer node to attack. Hiding the file location also reduces the risk

of intentional file corruption by a malicious attacker. This protection is enforced by introducing an indirection step in the look-up phase. The file location is hidden by storing it on a randomly selected peer node. Its location is revealed only to the set of authorized users. The location information is also stored in a secure manner in the meta-data file such that only authorized users can extract it.

An important issue in our approach is represented by the key distribution mechanisms. The keys used to encrypt the portions need to be securely and efficiently distributed to the authorized users. To address this issue, we propose a variation of the access control vector key distribution technique proposed by Shang et al. [2]. Whereas such approach uses an individual's credentials for key distribution, our mechanism is based on public keys. A user must first register his public key with the file publisher to obtain a subscription secret. This secret is then used to derive the keys for the file portions to which the user is granted access. Each user is provided a unique secret, even if the policy grants identical permissions to multiple users. A nice feature of our design is that no further communication between the user and the file owner is required after the initial registration. Even if the file is updated or re-encrypted with a different key, the original subscription secret can still be used for key derivation. We compare our key distribution scheme with conventional light-weight directory access protocol (LDAP [3]) key distribution, and discuss the trade-offs incurred by adopting our scheme over LDAP.

The main contributions of our paper can be summarized as follows:

- We define a novel FGAC mechanism for secure file sharing on P2P that allows content owners to specify access control policies over arbitrary byte ranges in files with no assumptions regarding the file structure.
- We provide a low-level architecture that can be extended to implement previous block- and file-level access control schemes.
- We present details of our implementation and its integration with Chord P2P along with experimental results about the performance of our prototype.

The rest of this paper is organized as follows. Section II discusses previous related research. Section III introduces relevant background material. Section IV details our FGAC mechanism and key management schemes. We discuss issues and challenges encountered during our implementation in Section V, and present our prototype evaluation in Sections VI and VII. We conclude the discussion in Section VIII and propose directions for future work.

## II. Related Work

A number of approaches have been proposed that aim at developing access control mechanisms for file sharing. Sirius [4] is a file system built on top of untrusted P2P system that enforces only file-level access control. Plutus [5] enforces access control by using block-level file encryption. However, Plutus relies on a client-server architecture, rather than P2P, and requires the keys be obtained from file owners on demand. To minimize the number of keys, Plutus employs file groups. Unlike Sirius, our access control mechanism operates at a finer granularity by granting access to portions of a file. In contrast to Plutus, our key distribution scheme does not require users to contact file owners. Furthermore, rather than using file groups, we minimize keys for portions within a file.

In the operating systems realm, several cryptographic file systems have been proposed. Some of these systems, DM-Crypt [6], TrueCrypt [7] and loop-AES [8], only support coarse-grained access control. In such systems, only per-mount point based encryption policies are possible which means that all data is encrypted, even if it is not sensitive. The Cryptographic File System [9] has the same shortcoming as the encryption granularity is the directory. In our work, we allow the file owner to specify access policies with very fine granularity. We encrypt only the sensitive data portions and avoid unnecessary encryption of insensitive data.

eCryptFS [10], which is an encrypted stacked file system for Linux, uses a per-file key management which implies that the granularity of access control is the file. In this scheme, the cryptographic meta-data is stored in the file header which allows the file to be moved. Our scheme uses a similar approach for storing the cryptographic information (such as the access vectors for key derivation) together with the file which allows the file to be moved across various nodes of P2P without much overhead. While eCryptFS uses one encryption key per file, we use multiple encryption keys to encrypt different portions of the file and enforce FGAC.

PeerStreaming [11] is P2P media streaming system for distributing media securely. An identical copy of encrypted media is delivered to all clients. The key distribution scheme for PeerStreaming is a PKI based approach which embeds the master key in a license and encrypts it with the public key of user. We use a more efficient key distribution scheme based on access control vector (ACV) approach which generates a unique ACV for distributing keys to all users. PeerStreaming does not define any policy-based FGAC framework. Anyone having a master key can decrypt the entire file. We perform encryption at application level based on the access control policies and the users accessing the file, while in PeerStreaming encryption is at network layer.

FireCoral [12] is a P2P web content distribution service which allows a user to specify his sharing policy by using whitelist and blacklist of domains and/or URLs. It uses signing services by a trusted third party to authenticate data so that its integrity can be verified by other peers. We specify sharing policies for users instead of domains, and support fine-grained resource sharing among peers. We do not employ any third party signing services. Instead, we allow any user with write access to sign the file portion and validate the signing key information by the content owner's signature.

## III. Background

In this section we first introduce some key aspects of Chord and then review the group key management scheme by Shang et al. [2], on which our scheme is based.

### A. Chord

Chord [1] is a structured P2P system based on a distributed hash table (DHT). The fundamental structure of a DHT is that all objects (including keys and nodes) are assigned $m$-bit identifiers which are used to map keys to peer nodes. Chord constructs a logical ring (called Chord ring) consisting of the ordered identifiers from 0 to $2^{m-1}$. A key $k$ is assigned to a node whose identifier is equal to or follows the identifier of $k$. This node is referred to as $successor(k)$. Each node must know the IP address of its successor node to perform a linear search. To improve search efficiency, each node (with identifier say $n$) maintains a finger table which has $m$ entries. The $i^{th}$ entry contains the IP address and Chord identifier of successor $(n + 2^{i-1})$. Thus, in steady state, each node maintains information about $O(\log N)$ other nodes. Using this finger table, a successor for a given key can be determined by contacting $O(\log N)$ nodes.

### B. Group key management

The key management scheme by Shang et al. [2] has been proposed for supporting selective access to documents. In this scheme, the publisher ($Pub$) of a document $D$ specifies access control policies (ACP) against sub-documents of $D$ and a conjunction of identity attribute conditions to be satisfied by a subscriber ($Sub$) for accessing a sub-document.

First, the subscriber $Sub_i$ presents its certified identity attributes to the *Identity Manager*, which is a trusted third party, to obtain identity tokens. $Sub_i$ then registers these identity tokens with $Pub$ in order to receive a *conditional subscription secret* (CSS). To enable token registration, $Pub$ chooses and publishes a $l$-bit prime number $q$, a cryptographic hash function $H(.)$ with output length no shorter than $l$ and a semantically secure symmetric key encryption algorithm with key length $l$-bits. For each ACP that $Sub_i$ wants to satisfy, $Sub_i$ registers an identity token whose id-tag matches the attribute name in condition $cond_j$ in ACP. $Pub$ verifies the credential and generates a $\kappa$-bit random value $r_{i,j} \in \mathbb{F}_q$, where $\mathbb{F}_q$ is a finite field with $q$ elements and $\kappa$ is the security parameter chosen by $Pub$. This $r_{i,j}$ is the conditional subscription secret (CSS). It is sent to $Sub_i$ using an OCBE [13] session. $Pub$ maintains a table $\mathcal{T}$ of the delivered $r_{i,j}$'s along with $nym_i$ (unique pseudonym for $Sub_i$) and policy condition $cond_j$.

All sub-documents which have same set of policies applicable to it are encrypted with same encryption key. Suppose that a sub-document $D_1$ has a policy configuration $\mathcal{PC} = \{\text{acp}_1, \ldots, \text{acp}_\alpha\}$. Let each policy $\text{acp}_k$ be a conjunction of conditions $cond_1 \wedge \ldots \wedge cond_{m_k}$. For each $\text{acp}_k$, $Pub$ constructs a set $\mathcal{U}_k = \{nym_1^{(k)}, \ldots, nym_{n_k}^{(k)}\}$ containing all those $nym_i$'s whose CSS records corresponding to attribute conditions are found in $\mathcal{T}$. Then $Pub$ chooses some

$$N \geq \sum_{k=1}^{\alpha} \#\mathcal{U}_k$$

where $\#\mathcal{U}_k$ is the cardinality of $\mathcal{U}_k$.

For each unique policy configuration, $Pub$ chooses a symmetric encryption key $K$ randomly from $\mathbb{F}_q$ and $N$ random values $z_1, \ldots, z_N \in \mathbb{F}_p$, for some prime $p$. Then $Pub$ constructs matrix $A$ as follows:

$$A = \left(\begin{array}{ccccc} 1 & a_{1,1}^{(1)} & a_{1,2}^{(1)} & \ldots & a_{1,N}^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a_{n_1,1}^{(1)} & a_{n_1,2}^{(1)} & \ldots & a_{n_1,N}^{(1)} \\ \hline \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline 1 & a_{1,1}^{(\alpha)} & a_{1,2}^{(\alpha)} & \ldots & a_{1,N}^{(\alpha)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a_{n_\alpha,1}^{(\alpha)} & a_{n_\alpha,2}^{(\alpha)} & \ldots & a_{n_\alpha,N}^{(\alpha)} \end{array}\right)$$

with $a_{i,j}^{(k)} = H(r_{i,1}^{(k)} || r_{i,2}^{(k)} || \ldots || r_{i,m_k}^{(k)} || z_j)$, where $||$ denotes concatenation.

$Pub$ solves $AY = 0$ to get a nonzero $(N+1)$ dimensional column vector $Y$, called the *access control vector* (ACV). For broadcasting $D_1$ which is encrypted with symmetric key $K$, $Pub$ constructs a vector $X$ as

$$X = (K, 0, 0, \ldots, 0)^T + Y,$$

where $v^T$ denotes the transpose of vector $v$. $Pub$ then broadcasts the encrypted document $D_1$ along with $X$, $z_1$, ..., $z_n$. To read a document, $Sub_i$ derives the decryption key using the CSS issued to it. It selects a policy $\text{acp}_k$ which it satisfies and derives the decryption key by computing

$$K' = (1, a_{i,1}^{(k)}, a_{i,2}^{(k)}, \ldots, a_{i,N}^{(k)}).X,$$

One of the advantages of this scheme is that updates of keys do not require a secure communication channel.

## IV. System Design

In this section we describe our selective encryption-based FGAC approach for P2P file sharing. The files under consideration are unstructured, UNIX-like files. Based on the access control policies specified by the file owner, a file is divided into read portions which are encrypted with different keys. Access to read a file portion requires possession of corresponding decryption key. Based on the write access policies, each read portion is further divided into write portions which are signed using different keys. The keys for signing are given to users with write access so that they can generate valid signatures after modifying a file portion. Thus, minimization of the number of generated keys and key distribution play a very important role.

### A. Access control policies

An access control policy is specified against the public keys of users and has the form

$$\text{acp} = \langle id, R, \mathcal{K}, P, D \rangle$$

where: $id$ is the policy identifier which uniquely identifies each policy; $D$ is the file on which access control is specified; $R$ is a byte-range $[s, e]$ in $D$, such that $0 \leq s \leq e < L$, where $L$ is length of $D$; $\mathcal{K}$ is the set of public keys of users who can access range $R$; and $P$ is the privilege that is granted and can be one of *read-only (r)*, *read-write (rw)* and *write (w)*.

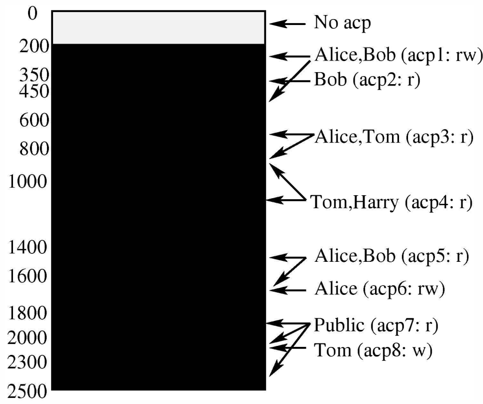



Fig. 1. Access control policies for file F

An access control policy, $\text{acp} = \langle id, R, \mathcal{K}, P, D\rangle$, is said to be satisfied by a user with privilege $P$, if his public key is in $\mathcal{K}$.

A *read-only* privilege indicates that the user has read access to the specified byte range. A *read-write* privilege specifies both read and write accesses to a private file portion. To specify write access for a public portion, the *write (w)* privilege is used. Notice that we do not include a *write-only* privilege since it implies that a user can perform only plain-text insertion which destroys the confidentiality of private portions.

The byte-ranges which have no policies associated with them are accessible only to the owner of the file. A special policy, called *public policy*, indicates which portions of the file are public. This policy is characterized by an empty $\mathcal{K}$ set.

A *policy set* is defined as the set of all policies associated with a file. A *minimal* policy set is a policy set which only contains policies that are not subsumed by other policies. A policy $\text{acp}_1 = \langle id_1, R_1, \mathcal{K}_1, P_1, D\rangle$ subsumes a policy $\text{acp}_2 = \langle id_2, R_2, \mathcal{K}_2, P_2, D\rangle$ if $R_2 \subseteq R_1$, $\mathcal{K}_2 \subseteq \mathcal{K}_1$ and $P_2 \subseteq P_1$.

A *read policy configuration* $\mathcal{PC}_R$ for a byte-range $R$ is defined as the set of all access control policies $\{\text{acp}_1, \ldots, \text{acp}_n\}$ where $\text{acp}_i = \langle id_i, R_i, \mathcal{K}_i, P_i, D\rangle$, $i = 1, \ldots, n$, is such that $R \subseteq R_i$ and $P_i = \{r|rw\}$. In other words, $\mathcal{PC}_R$ for byte-range $R$ includes all policies with read privileges that are specified on byte-ranges that include $R$. Similarly, a *write policy configuration* $\mathcal{PC}_W$ for a byte-range $R$ is the set of all access policies $\{\text{acp}_1, \ldots, \text{acp}_n\}$ where $\text{acp}_i = \langle id_i, R_i, \mathcal{K}_i, P_i, D\rangle$, $i = 1, \ldots, n$, is such that $R \subseteq R_i$ and $P_i = \{rw|w\}$.

A *read user group* $\mathcal{U}_R$ for a byte-range $R$ is defined as the set of all users who satisfy some policy in the read policy configuration of $R$. Similarly, we define a *write user group* $\mathcal{U}_W$ for a byte-range $R$ as the set of all users who satisfy some policy in the write policy configuration of $R$.

For example, consider a file $F$ (Fig. 1) owned by John which has the following policies associated with it:

- $\text{acp}_1 = \langle 1, [200, 600), \{Pu_{\text{Alice}}, Pu_{\text{Bob}}\}, rw, F\rangle$
- $\text{acp}_2 = \langle 2, [350, 450], \{Pu_{\text{Bob}}\}, r, F\rangle$
- $\text{acp}_3 = \langle 3, [600, 1000), \{Pu_{\text{Alice}}, Pu_{\text{Tom}}\}, r, F\rangle$
- $\text{acp}_4 = \langle 4, [800, 1400), \{Pu_{\text{Tom}}, Pu_{\text{Harry}}\}, r, F\rangle$
- $\text{acp}_5 = \langle 5, [1400, 1800), \{Pu_{\text{Alice}}, Pu_{\text{Bob}}\}, r, F\rangle$
- $\text{acp}_6 = \langle 6, [1600, 1800), \{Pu_{\text{Alice}}\}, rw, F\rangle$
- $\text{acp}_7 = \langle 7, [1800, 2500), \{\}, r, F\rangle$
- $\text{acp}_8 = \langle 8, [2000, 2300), \{Pu_{\text{Tom}}\}, w, F\rangle$

Here $\text{acp}_7$ indicates that the range $[1800, 2500)$ is public and need not be encrypted. In order to construct a minimal policy set for $F$, we need to eliminate $\text{acp}_2$ since it is subsumed by $\text{acp}_1$. Thus, the minimal set of policies is as follows:

$$P_{min} = \{\text{acp}_1, \text{acp}_3, \text{acp}_4, \text{acp}_5, \text{acp}_6, \text{acp}_7, \text{acp}_8\}$$

For the file portion $[800, 1000)$, $\mathcal{PC}_R = \{\text{acp}_3, \text{acp}_4\}$ and $\mathcal{U}_R = \{\text{John,Alice,Tom,Harry}\}$. For file portion $[1600, 1800)$, $\mathcal{U}_R = \{\text{Alice,Bob,John}\}$ and $\mathcal{U}_W = \{\text{Alice,John}\}$.

### B. Read and Write Permissions

A user can access portions of a file with *read-only*, *read-write* or *write* privileges. These privileges are enforced by using separate keys for encrypting and signing file portions and are referred to as the *read key* and the *write key* respectively. Each write key has a corresponding *verify key* used to verify the signatures on file portions signed using this write key.

The read key is used to encrypt the file portions and guarantees confidentiality. The read key for a file portion is given to only those users who have *read* privilege for that portion so that they can decrypt and read it. Also, in case of *read-write* privilege, the read key is used to encrypt the file portions after modification. The write key is used to sign the file portions and guarantees integrity. Only users with *read-write* or *write* privilege are given the write key for the file portion. Thus, only users with write access are able to generate valid signatures after file modification. The verify keys are public and can be obtained by any user from the cryptographic meta-data. Note that a symmetric key based integrity solution will not work in this case since it would imply that only a user with write access would be able to verify integrity, while a user with only read access would be unable to do so.

### C. File partitioning and key generation

To enable selective encryption of a file, we first need to partition the file into non-overlapping file portions based on access control policies. These partitions are generated based on *read* and *read-write* policies. Each of these file portions is associated with a read-user group, which is the set of users with read access to that file portion, and need to be encrypted with a read key, except if it is a public portion. These file portions are called *read partitions*. Once all the read partitions have been generated, each read partition is further associated with a set of *write partitions* based on write access policies.

We aim at minimizing the total number of read and write keys per file. To do this, we encrypt all read partitions which have the same associated read user group with the same read key. Similarly, we use the same write key for signing the write partitions associated with same write user group. Thus, while generating the keys for a file, we assign each unique read user group a different symmetric read key and each unique write user group a different pair of write-verify keys.

The algorithm for file partitioning and minimal key generation (see Fig. 4) takes a minimal set of policies as input. If the policy set associated with $D$ is not minimal, a minimal policy set $\mathcal{P}_{min}$ is first constructed. The algorithm returns a set of

TABLE I
READ PARTITIONS

| Byte Range | User Group | Read key | Write Partitions |
|---|---|---|---|
| [0,200) | John | $\text{eKey}_1$ | $W_1$ |
| [200,600) | John,Alice,Bob | $\text{eKey}_2$ | $W_2$ |
| [600,800) | John,Alice,Tom | $\text{eKey}_3$ | $W_3$ |
| [800,1000) | John,Alice,Tom,Harry | $\text{eKey}_4$ | $W_4$ |
| [1000,1400) | John,Tom,Harry | $\text{eKey}_5$ | $W_5$ |
| [1400,1800) | John,Alice,Bob | $\text{eKey}_2$ | $W_6$,$W_7$ |
| [1800,2500) | Public | - | $W_8$,$W_9$,$W_{10}$ |

TABLE II
WRITE PARTITIONS

| Write Partition | Byte Range | User Group | Write key |
|---|---|---|---|
| $W_1$ | [0,200) | John | $\text{sKey}_1$ |
| $W_2$ | [200,600) | John,Alice,Bob | $\text{sKey}_2$ |
| $W_3$ | [600,800) | John | $\text{sKey}_1$ |
| $W_4$ | [800,1000) | John | $\text{sKey}_1$ |
| $W_5$ | [1000,1400) | John | $\text{sKey}_1$ |
| $W_6$ | [1400,1600) | John | $\text{sKey}_1$ |
| $W_7$ | [1600,1800) | John,Alice | $\text{sKey}_3$ |
| $W_8$ | [1800,2000) | John | $\text{sKey}_1$ |
| $W_9$ | [2000,2300) | John,Tom | $\text{sKey}_4$ |
| $W_{10}$ | [2300,2500) | John | $\text{sKey}_1$ |

non-overlapping read partitions, and for each read partition, its associated user group and set of write partitions.

*Time complexity analysis:* The file partitioning algorithm has two main parts - (1) generation of read partitions, and (2) generation of write partitions. Let $n$ be the number of policies in the policy set which is given as input to this algorithm. Let $k_1$ and $k_2$ be the total number of read and write partitions generated respectively as output. The first part of the algorithm requires processing all read policies one at a time. For each policy, it requires finding the first and last overlapping read partitions (which has logarithmic time) and process all the intervals lying between the first and last overlaps. These are the intervals which overlap with current ACP's byte range and can be $O(k_1)$ in worst case. Thus the worst case running time of first part of the algorithm is $O(k_1 n)$. The second part of the algorithm requires processing all write policies one at a time. For each policy, it finds the overlapping read portions (which is logarithmic time). For each of these read portions, it updates the set of write portions associated with it. The worst case amortized cost for updating the write portions is $O(k_2)$ for each policy. Thus, the worst case running time for the second part of the algorithm is $O((k_2 + \log k_1)n)$. The worst case running time of the entire algorithm is $O((k_1 + k_2)n)$.

Executing the partitioning algorithm on the example in section IV-A would generate the partitions as shown in Tables I and II respectively. Note that the file owner, John, has access to the keys for all file portions. The file portions $[200, 600)$ and $[1400, 1800)$ are encrypted with same key $\text{eKey}_2$ since they have the same read user group {John,Alice,Bob} associated with them. Also, the file portions with no write policy specified for them are signed using the file owner's key.

### D. Key management

The keys for the file portions need to be securely and efficiently distributed to the authorized users. Key distribution is accomplished by using a variation of the ACV-based group key management scheme (see Section III-B). In describing our approach, we use the term *user* instead of *subscriber* and *file owner* instead of *publisher*. Also, public keys are used instead of credentials, which is equivalent to each user ($Sub$) having exactly one credential of type *Public-key*. We now highlight the differences between Shang et al. [2]'s key management scheme and our approach.

In our approach, each $user_i$ registers with *file owner* using its public key $Pu_i$. The *file owner* verifies the certificate corresponding to this public key and generates exactly one CSS $r_i$ per user. This CSS is sent to $user_i$ encrypted with $Pu_i$ and can be retrieved by $user_i$ using its private key $Pr_i$. The file owner maintains a table $\mathcal{T}$ that contains $nym_i$ (unique pseudonym for $user_i$) along with the corresponding delivered CSS $r_i$ for each registered user.

The file portions which have the same associated read-user group will be encrypted with the same symmetric read key. This differs from the ACV approach where each sub-document with the same policy configuration is encrypted with the same key. To illustrate this, consider the example introduced in Section IV-A. The file portions $[200, 600)$ and $[1400, 1800)$ both have the same read user group $\{John, Alice, Bob\}$ but different read policy configurations, namely:

$$\mathcal{PC}_{R,[200,600)} = \{\text{acp}_1\} \text{ and } \mathcal{PC}_{R,[1400,1800)} = \{\text{acp}_5\}$$

Since we aim at minimizing the total number of read keys for a file, it is beneficial to use same read key for file portions with the same read user-groups. We use a similar approach for write keys wherein a unique write-verify key pair is generated for each write user group.

Consider the key distribution for a file portion $D_1$. Let $D_1$'s associated read-user group be $\mathcal{U}_1 = \{nym_1, \ldots, nym_\alpha\}$. The *file owner* searches table $\mathcal{T}$ for each of these $nym_i$'s and constructs a set $\mathcal{K}_1 = \{nym_{i_1}, nym_{i_2}, \ldots, nym_{i_{n_1}}\}$ containing $nym_i$'s whose corresponding CSS is found in $\mathcal{T}$. The *file owner* chooses a suitable value $N \geq \#\mathcal{K}_1$ where $\#\mathcal{K}_1$ denotes the cardinality of set $\mathcal{K}_1$.

For $\mathcal{U}_1$ (or equivalently $D_1$), the *file owner* chooses a symmetric read key $K$ randomly from $\mathbb{F}_q$ and $N$ random values $z_1, \ldots, z_N \in \mathbb{F}_p$, for some prime $p$. The matrix $A$ constructed by owner will have only one row per registered user in the user group of $D_1$:

$$A = \begin{pmatrix} 1 & a_{i_1,1} & a_{i_1,2} & \ldots & a_{i_1,N} \\ 1 & a_{i_2,1} & a_{i_2,2} & \ldots & a_{i_2,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a_{i_{n_1},1} & a_{i_{n_1},2} & \ldots & a_{i_{n_1},N} \end{pmatrix}$$

with $a_{i,j} = H(r_i || z_j)$, where $r_i$ denotes the CSS for user with pseudonym $nym_i$ and $||$ denotes concatenation. The remaining steps are the same as the steps of the ACV scheme (see section III-B). The write keys are also distributed in a similar manner using ACVs for each unique write user group.

### E. Managing Updates to Policies

Modifying an access control policy can be considered as a two step process - deleting the policy followed by adding a new policy. We provide four operations for modifying the

polices - *Add policy*, *Delete Policy*, *Add User* and *Delete User*. Although the last two operations are redundant, we include them for efficiency reasons. While *add policy* and *add user* grant access to file portions, *delete policy* and *delete user* revoke access. All of these may require re-partitioning and re-encryption for the file portions that overlap with the policy's byte range. In some special cases, re-partitioning can be avoided. For example, when Tom is added to policy $acp_5$ (see Fig. 1), we only need to re-encrypt byte range $[1400, 1800]$ with a new key. In a scenario where policies change frequently, the file may become more and more fragmented over time. This requires that a merging effort be applied wherein the adjacent file portions with same associated user groups are merged into one portion.

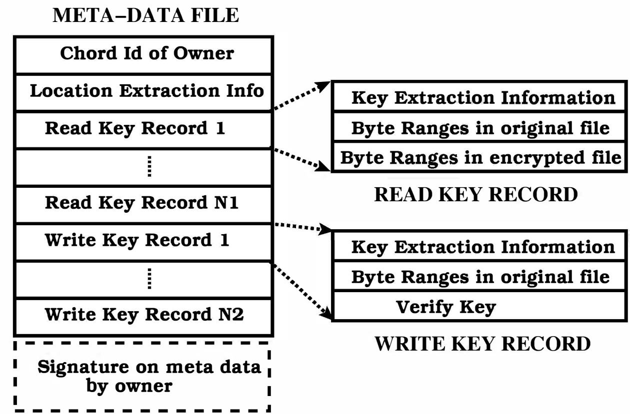



Fig. 2. Structure of the Meta-Data File

### *F. Integration with Chord P2P*

Our FGAC approach can be integrated with any structured P2P system. However, we chose Chord P2P as it provides a working implementation. Note that *user* is not tied to a particular node, as the policies are specified against public keys, not Chord node identifiers. This design choice makes our approach easily adaptable to alternative P2P systems. The shared file is selectively encrypted and stored at a peer node depending on its key identifier. We refer to this selectively encrypted file as the *data file*. The key-identifier for the data file is computed by taking a hash of its complete file name. As discussed later, if the file location is to be hidden, a random file name is generated for the data file which results in a random key-identifier that is revealed only to authorized users.

The cryptographic data (see section IV-D) required to derive the keys and file location is stored in a separate meta-data file (refer Fig. 2). This file stores the following information:

- Chord node-identifier of the node corresponding to the file owner.
- Information for extracting the Chord key of the data file to determine its location (see section IV-G).
- For each read key, the key extraction information (vector $X$ and random $z$ values (see section III-B)), and a mapping between the original byte-ranges and the byte-ranges in the encrypted file for all the portions encrypted with this key.
- For each write key, the key extraction information (vector $X$ and random $z$ values as described in Section III-B), the byte-ranges in the original file for all the portions which are signed with this key and the corresponding verify key.

The Chord node-identifier of the file owner is used when a user wants to register or needs to know his access rights. It is file owner's responsibility to keep this identifier correctly updated if it uses different P2P nodes at different times.

In the basic approach where the file location is not hidden, the meta-data file is assigned the same key-identifier as the corresponding data file. This ensures that both the data and the meta-data file are stored on the same node and can be retrieved together. A trivial approach to storing meta-data and data together would be to combine them in one file. However, this would require the entire file to be changed every time a policy change occurs. Storing the meta-data as a separate file has the advantage that in some cases only the meta-data file has to be updated without affecting the data file. To illustrate this point, consider the example introduced in section IV-A. Suppose a new user Joe has to be added to policy $acp_3$. This change can be done by simply adding Joe to user groups for file portions $[600, 800)$ and $[800, 1000)$ and giving Joe keys $eKey_3$ and $eKey_4$. This policy change does not require re-encryption of the data file since keys for these portions are not used to encrypt any other file portion to which Joe does not have access. Thus, only the meta-data file needs to be updated. Clearly, in these scenarios, storing the data file and meta-data file separately is better approach. If the file location is to be hidden, then clearly the meta-data file and the data file cannot be stored on same peer node (see section IV-G).

Only the file owner can modify the cryptographic meta-data associated with a file. If this was not the case, then anyone (even an authorized user) could modify or corrupt the meta-data and make it unusable for other users. The contents of the meta-data file are signed using the private key of file owner. This signature is stored with the meta-data file and can be verified by any user. Although this does not prevent anyone from corrupting the meta-data, it does help in detecting the corruption and seeking the correct file from alternative sources like the file owner node.

While preparing the encrypted data file, the signatures are computed on the plain text file portions which are then encrypted. Another approach would be to first encrypt the file portions and then sign them. The latter approach is not desirable because it generates a higher number of partitions and if the adjacent byte ranges have the same read user group but different write user group, they will be encrypted separately. To illustrate this point, consider the byte range $[1400, 1800)$. While the first approach encrypts this range as a single file portion, the latter approach will require the range to be split into two portions, $[1400, 1600)$ and $[1600, 1800)$, since they have different write user groups and have to be encrypted and signed as two different portions. These signatures are computed using the write-key for the write user group associated with a file portion and can be verified using the corresponding verify key.

To read a file portion, a user first needs to retrieve the

corresponding meta-data file and verify its signature to ensure its integrity and authenticity. To retrieve the data file, its Chord key must be known which is the same as the meta-data in basic approach. In the case where file location is hidden, this information is securely stored in the meta-data file and can be successfully extracted only by authorized users. The user then extracts the required cryptographic information and derives the read key and write key (in case of write-access) for the file portion that he wants to read/modify. A successful derivation of these keys is possible only if the user is authorized with these privileges. The user also obtains the verify key for this file portion from the meta-data file. This verify key is public and can be obtained by anyone. The file portion to be read is first decrypted using the read key and then verified for correctness by using the verify key for signature verification. If a user has write access, he can modify the file (refer to section IV-H), re-compute the signature and encrypt the file portion again. The modified file is then inserted into P2P using the same Chord-key as earlier.

The derivation of the keys and the decryption of content are performed on the user's node and not on any other node in the P2P system since the communication channel and the other P2P nodes cannot be trusted to be secure.

The access control policies associated with a file are stored in a separate file, called *policy store*, which is fully encrypted with a symmetric key available only to the owner of the file. Therefore, their confidentiality and integrity are assured. As the policy store is an encrypted file it can be stored either at file owner node or on any other node of the P2P system, just like any other encrypted file.

### *G. Hiding file location in Chord P2P*

We propose an optional enhancement over the basic scheme by which we allow the file owner to hide the file location. This is enforced by introducing an extra indirection step in the look-up phase. The data file is assigned a randomly generated file name. The Chord insertion key is computed as a hash of this random file name. This ensures that the file location is decided at random and cannot be guessed by an attacker.

This random file name is hidden inside an access control vector (section IV-D) which is similar to the access vectors used to hide the read and write keys. This vector is constructed using the subscription secrets of all those users who are authorized to know file location. This access vector is also stored in the meta-data file (refer Fig. 2) and the file location can be extracted from it only by authorized users. The location of the meta-data file is not hidden, since it is required to extract the information of the actual data file. Also, the meta-data does not reveal the location or contents of the protected file in public. It can only reveal the information on how a file is partitioned into portions.

This approach is secure against a brute force search where an adversary attempts to retrieve files from all P2P nodes to determine the location of a particular file. This search will not succeed because as the data file is selectively encrypted and its file name is random, an attacker will be unable to determine if the file retrieved by him is correct. However, this is true only in the absence of public portions, since they may reveal some information about the file.

### *H. File Modification*

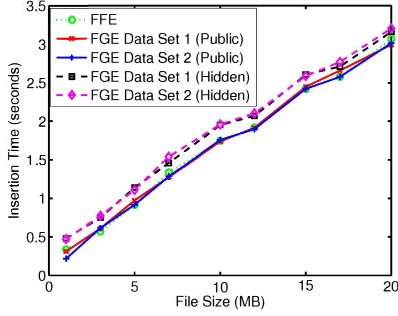
Modifications to a file affect not only the file portions but also the access policies and the meta-data associated with it. File operations which change the length of a file portion cause other byte-ranges to shift, thus affecting other access policies and meta-data information. As discussed earlier, only file owner can modify the meta-data file. This restricts other users from performing file operations which modify the file-portion length since that would require changes in the meta-data file. We allow following modifications by users to a file:

- *Update:* We refer to *update* as a write operation that overwrites existing bytes with new bytes without increasing length of a portion. This is handled by re-encrypting the modified portion with the same key as earlier. We do not support updates that increase the length of a file portion.
- *Append:* Append operations add content to the end of a file. An append operation does not affect any existing file portions or policies. A new file portion is created and since no access policy applies to it, it is encrypted and signed with the owner's keys. Append operations can only be performed by the file owner.
- *Delete:* Delete operations are handled differently depending on whether they are issued by the file owner or other users. Deleting a range of bytes causes other byte ranges to be shifted and thus affects the access control policies and the meta-data. A file owner performs a delete operation by deleting the required bytes and adjusting the access control policies by a constant factor. There is no need of re-partitioning or re-encrypting other partitions except for the partition being modified. Also, the byte ranges information in the meta-data file will need to be adjusted. For other users, the delete operation is basically an overwrite operation where the deleted bytes are overwritten by zero's and marked as deleted. These zeroed bytes can be actually deleted by file owner later in a lazy manner.

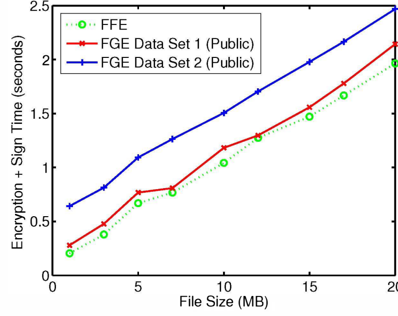
## V. Implementation Issues

The implementation of our approach required addressing several challenging implementation issues, some of which also required extending the functionality of the underlying Chord P2P. We used the Open Chord [14] implementation as a basis for our prototype.

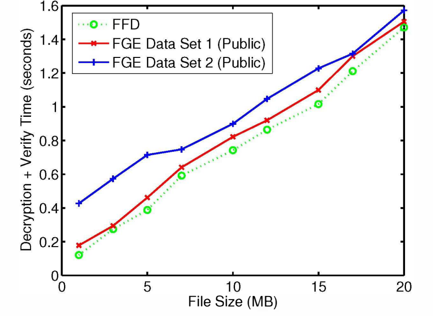
The first extension was to introduce in Open Chord a *register* command which enables a user to register with the owner node to obtain the CSS. This functionality was not implemented at application level since Open Chord does not provide primitives for exchanging messages over which a registration protocol could be built. Also, this would require establishing a separate communication channel between the owner and user node. A low level implementation of the register functionality allowed us to make use of underlying P2P to carry out this protocol.



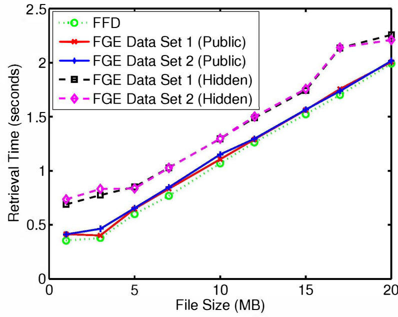

(a) Experiment 1: Insertion time with varying file sizes



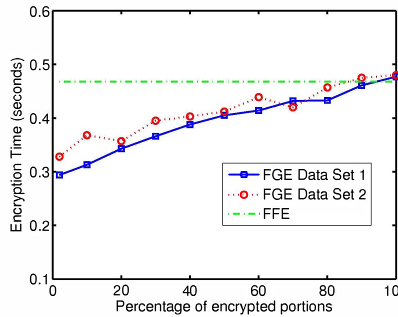

(b) Experiment 1: Encryption and Signing times with varying file sizes



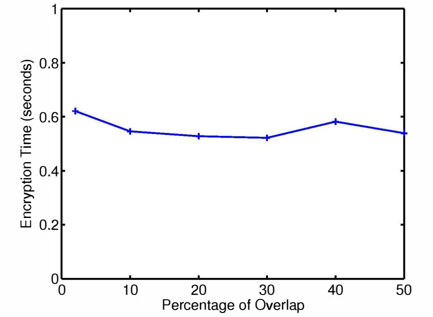

(c) Experiment 1: Decryption and Verify times with varying file sizes




(d) Experiment 1: Retrieval time with varying file sizes




(e) Experiment 2: Encryption times for FFE and FGE with varying size of public portions




(f) Experiment 3: Encryption time for FGE with increase in policy overlap

Fig. 3. Experiment results

The second extension was to modify the command to *insert* a file in Open Chord. Our scheme requires insertion of two files - the data file and meta-data file. The conventional approach which we implemented initially made the call to the Chord *insert* command twice, which required the search for responsible peer node to be executed twice. Since our basic scheme requires storing both these files on the same node, we modified the lower level *insert* command to insert both these files by determining the responsible node just once.

The third extension was to add support for hiding the file location in Open Chord. This functionality could be implemented either at application level or at a low level within Open Chord. We decided to incorporate this at low level to make this process transparent to the user. This required modification of *insert* and *retrieve* commands which are now overloaded to perform insertion and retrieval operations with and without hiding file location. The *Retrieve* command now involves retrieving the meta-data, extracting the hidden location and then performing retrieval of the actual data file.

Lastly, we extended Open Chord to support the *modify* command which allows users to update an existing copy of a file on a peer node instead of inserting a new file every time there is a change.

## VI. Experimental Evaluation

We performed three experiments to measure the time required for insertion, retrieval and various cryptographic operations. They were performed on two different machines[1]. In our tests, the insertion and retrieval times include the time for inserting and retrieving both data and meta-data files in Chord P2P. The encryption time includes the time required to generate file partitions, keys for each user group, meta-data file and to perform selective encryption of the data file.

### A. Experiment 1

The goal of this experiment is to compare the performance of our approach, referred to as fine-grained encryption (FGE) with a conventional approach, referred to as full file encryption (FFE), which consists of signing the entire file and then encrypting it using a single key. We compare the times for insertion, retrieval and cryptographic operations (i.e., encryption and signature) for files of varying sizes. We also study the impact of hiding the file location on the insertion and retrieval times. In these measurements, we do not include any public policy for the FGE approach. Graphs in Fig. 3a - 3d report the results of this experiment performed on two data sets.

These two data sets are for encryption and signing using one and ten keys (and ten partitions) respectively. We perform measurements on these data sets with and without location hiding. FFE forms the baseline for our comparison. We observe that our approach does not add any significant

[1]Machine configurations: (1) Windows XP SP3 32-bit (Intel Core 2 Duo CPU, T5750, 2.0GHz, 3GB RAM); (2) Windows Vista SP2 64-bit (Intel Core 2 Duo CPU, T5750, 2GHz, 4GB RAM)

overhead to the conventional approach. The times for partition generation and creation of meta-data file are insignificant when compared to the actual encryption time. Fig. 3a and 3d report the insertion and retrieval times. Our approach introduces some overhead in insertions because of the insertion of the meta-data file. However, the modified insert command (see section V) significantly reduces this overhead.

As shown by these graphs, location hiding is very efficient, especially for larger file sizes. It includes the extra overhead of doing the look-up twice, but as the file size increases, the insertion and retrieval times are dominated by the data transfer time. We observed that the time measurements for decryption and verification operations follow a similar trend as encryption and signature operations and have an insignificant overhead.

### *B. Experiment 2*

In this experiment, we evaluate the advantage of the FGE scheme over the FFE scheme. As our approach supports the specification of a special policy to indicate the public portions of a file, we encrypt only the sensitive portions and leave the public portions as they are. The FGE scheme is more efficient when only a small portion of the file is sensitive as it reduces the amount of encryption. We compare the encryption and insertion times for FGE and FFE by varying the percentage of file portions which are encrypted. These measurements do not include the time taken for signing operations.

We consider two data sets, one with two ACPs and another with twenty-one ACPs. The results of this experiment, performed on a 5MB file, is reported in Fig. 3e. We observe that our approach performs significantly better than FFE when only a small portion of the file has to be encrypted. The insertion times for FGE and FFE were observed to be comparable.

### *C. Experiment 3*

In this experiment, we analyze the effect of various kinds of policy on the performance of our scheme. If the policies have overlapping ranges, then such policies may require splitting the policy ranges and generating many small partitions. In contrast, if the policy ranges are disjoint, then creating partitions does not require splitting ranges into smaller portions. We define the percentage of policy overlap as

$$\text{Overlap} = \frac{\text{Number of overlapping policies}}{\text{Total number of policies}}$$

. We consider different percentages of overlaps ranging from 2% overlap to 50%. This experiment is performed on a 5MB file with 100 ACPs and 10 users. Fig. 3f shows that the encryption time changes only slightly with increasing overlaps in policies. This indicates that the actual encryption time is the dominant factor in the total processing time, and the extra processing required by our scheme has a very slight overhead.

## VII. Analysis of the Key Distribution Scheme

In this section, we compare our key distribution approach with an LDAP-based approach and analyze the security of our approach.

### *A. Comparison of key distribution scheme with a conventional LDAP approach*

An alternative approach for key distribution is based on LDAP directory. The LDAP directory stores one entry for each user, and contains all the keys for the user encrypted with the user's public key. PKI-based encryption and decryption have to be performed by the LDAP server for insertion and by the user for retrieval. These PKI-based operations are computationally expensive. Furthermore, they have to be executed every time a key is changed, and for each user.

In contrast, in our key distribution scheme, only the user registration phase requires PKI-based operations for transmitting the CSS. This is a one-time process.The generation of key extraction information and retrieval of keys from the ACV vector requires only matrix-based operations which are much faster. Thus, our key distribution scheme is more efficient to use when a large number of users are involved and the access control policies are dynamic. However, our scheme incurs more storage overhead as compared to LDAP when the number of keys per file is large since a separate key record has to be created for each key which will contain the key extraction information. Thus, if the system has few users, then LDAP would be a better option.

### *B. Security Analysis*

In this section, we discuss how the basic security requirements of confidentiality, availability, and integrity are addressed by our approach.

*1) Confidentiality:* One possible attack to confidentiality is when an adversary attempts to derive a key from the information in the meta-data to access a file portion for which he is not authorized. To derive the key for a file portion, a user must possess a valid CSS for accessing such file portion.

There are two kinds of adversary in this system. The first kind is an attacker who has not registered with the owner and attempts to read a file portion. Since this user does not have a valid CSS, he cannot extract any of the read keys. Also, he cannot obtain the CSS of some other legitimate user by eavesdropping since the CSSs are sent encrypted over the communication channel. A second kind of attacker is a user who possesses a valid CSS and attempts to access a file portion for which he has not been granted access. This user can only construct a row of matrix $A$ (see section IV-D) for matrices corresponding to keys he has access to. Thus, he will be unable to derive any key for which he has not been granted access. Thus our scheme ensures that only users who have been granted access by the file owner can read a file portion.

*2) Availability:* In our scheme, an attacker can disrupt availability by corrupting or deleting the meta-data file so that legitimate users are unable to obtain the correct keys. Many extensions to Chord support an underlying replication mechanism to increase availability. In our scheme, meta-data is replicated together with the data file and therefore it would automatically benefit from any replication strategy that is implemented by Chord. Moreover, our experiments show that the size of the meta-data file is not significant.

**GeneratePartition**($\mathcal{P}_{min}$,$L$)

Input: $\mathcal{P}_{min}$ : a minimal set of ACP
$L$ : File size in bytes
$\mathcal{P}_{min}$ is generated from the given set of ACPs $\mathcal{P}$

Output: $\mathbb{D}$: a set of read partitions $D_i$
Partition $D_i = \langle [s_i, e_i], \mathcal{U}_i, wP_i \rangle$ where
- $[s_i, e_i]$: byte-range of a read partition
- $\mathcal{U}_i$: associated read user group
- $wP_i$: list of associated write partitions $wp_j$

Write Partition $wp_j = \langle [s_j, e_j], \mathcal{U}_j \rangle$ where
- $[s_j, e_j]$: byte-range of a write partition
- $\mathcal{U}_j$: associated write user group

Assumption:
$\mathcal{P}_{min}$ does not contain conflicting policies.
(That is, a byte range cannot be both public
and access controlled at same time.)
**insert**$\langle [s,e], \mathcal{U} \rangle$ executes only if $s \leq e$.

```
    /* Generate all read partitions */
1.  insert ⟨[0, L−1], {Owner}⟩ in 𝔻
2.  for each p = ⟨[s_p, e_p], U_p, P, D⟩ ∈ P_min s.t. P = {r|rw} do
3.    if (U_p = φ) then
4.      /* Special public policy */
5.      insert ⟨[s_p, e_p], φ⟩ into 𝔻
6.      continue for
7.    end if

8.    D_f ← findFirstOverlap(p, 𝔻)
9.    D_l ← findLastOverlap(p, 𝔻)
10.   if (D_f = D_l) then
11.     remove D_f from 𝔻
12.     insert ⟨[s_f, s_p − 1], U_f⟩ into 𝔻
13.     insert ⟨[s_p, e_p], U_f ∪ U_p⟩ into 𝔻
14.     insert ⟨[e_p + 1, e_f], U_f⟩ into 𝔻
15.   else
16.     if (U_p ⊈ U_f) then
17.       /* Split D_f */
18.       remove D_f from 𝔻
19.       insert ⟨[s_f, s_p − 1], U_f⟩ into 𝔻
20.       insert ⟨[s_p, e_f], U_f ∪ U_p⟩ into 𝔻
21.     end if

22.     if (U_p ⊈ U_l) then
23.       /* Split D_l */
24.       remove D_l from 𝔻
25.       insert ⟨[s_l, e_p], U_l ∪ U_p⟩ into 𝔻
26.       insert ⟨[e_p + 1, e_l], U_l⟩ into 𝔻
27.     end if

28.     for each D = ⟨[s, e], U⟩ ∈ 𝔻 s.t. (D_f < D < D_l) do
29.       U ← U ∪ U_p
30.     end for
31.   end if
32. end for
    /* Generate all write partitions */
33. for each p = ⟨[s_p, e_p], U_p, P, D⟩ ∈ P_min s.t. P = {rw|w} do
34.   D_f ← findFirstOverlap(p, 𝔻)
35.   D_l ← findLastOverlap(p, 𝔻)
36.   insertWritePartition (⟨[s_p, e_f], U_p⟩, D_f)
37.   insertWritePartition (⟨[s_l, e_p], U_p⟩, D_l)
38.   for each D = ⟨[s, e], U⟩ ∈ 𝔻 such that (D_f < D < D_l) do
39.     insertWritePartition (⟨[s, e], U⟩, D)
40.   end for
41. end for
```

Fig. 4. Algorithm for file partitioning based on ACP

Hence, applying traditional replication strategies and keeping a copy of the meta-data with each replica of the data file can provide robust availability without imposing significant storage overhead. We also reduce the risk of data corruption by providing a mechanism to hide the file location.

*3) Integrity:* We protect the integrity of the meta-data and the data-file by use of signatures. The meta-data file is signed by the file owner's private key and cannot be modified by other users. File portions of the data file are signed using different keys which are provided only to authorized users. Thus, any modification to the meta-data or the data-file by an unauthorized user can be easily detected.

## VIII. FUTURE WORK

As future work, we plan to extend our approach by developing key recovery mechanisms whereby the owner can recover the keys in case they are lost. Another interesting extension is to support delegation by the owner node to some trusted users to issue CSSs. This would result in a decentralized registration process. Also, in the current approach, we aim at minimizing the number of keys per file. As future work, we would like to study how to minimize the total number of keys per user. A possible approach would be to use a hierarchical key management scheme.

## ACKNOWLEDGMENTS

The work reported in this paper was supported by the National Science Foundation (NSF) grant 0712846 IPS: Security Services for Healthcare Applications, and the MURI award FA9550-08-1-0265 from the Air Force Office of Scientific Research.

## REFERENCES

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM'01*. New York, NY, USA: ACM, 2001, pp. 149–160.
[2] N. Shang, M. Nabeel, F. Paci, and E. Bertino, "A privacy-preserving approach to policy-based content dissemination," in *ICDE*, 2010.
[3] K. Zeilenga, "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map," RFC 4510 (Proposed Standard), Internet Engineering Task Force, Jun. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4510.txt
[4] E. jin Goh, H. Shacham, N. Modadugu, and D. Boneh, "Sirius: Securing remote untrusted storage," in *NDSS Symposium'03*, 2003, pp. 131–145.
[5] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *FAST '03*, Berkeley, CA, USA, 2003, pp. 29–42.
[6] "Dm-crypt: A device-mapper crypto target," www.saout.de/misc/dm-crypt.
[7] "Truecrypt: Free open-source on-the-fly disk encryption software for windows 7/vista/xp, mac os x, and linux," www.truecrypt.org.
[8] "loop-aes," http://sourceforge.net/projects/loop-aes/.
[9] M. Blaze, "A cryptographic file system for unix," in *CCS '93*. New York, NY, USA: ACM, 1993, pp. 9–16.
[10] M. Halcrow, "ecryptfs: a stacked cryptographic filesystem," *Linux J.*, vol. 2007, no. 156, p. 2, 2007.
[11] J. Li, Y. Cui, and B. Chang, "Peerstreaming: design and implementation of an on-demand distributed streaming system with digital rights management capabilities," *Multimedia Syst.*, vol. 13, pp. 173–190, 2007.
[12] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman, "Bringing P2P to the Web: Security and privacy in the Firecoral network," in *In IPTPS 09*, Boston, MA, Apr. 2009.
[13] J. Li and N. Li, "Oacerts: Oblivious attribute certificates," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 4, pp. 340–352, 2006.
[14] K. Loesing and S. Kaffille, "Open chord. an implementation of the chord dht in java," http://open-chord.sourceforge.net/.