Efficient and Secure Search of Enterprise File Systems

Aameek Singh **Mudhakar Srivatsa** **Ling Liu**

College of Computing, Georgia Insitute of Technology

{aameek, mudhakar, lingliu}@cc.gatech.edu

Abstract

With fast paced growth of enterprise data, quickly locating relevant content has become a critical IT capability. Research has shown that nearly 85% of enterprise data lies in flat filesystems [12] that allow multiple users and user groups with different access privileges to underlying data. Any search tool for such large scale systems needs to be efficient and yet cognizant of the access control semantics imposed by the underlying filesystem. Current multiuser enterprise search techniques use two disjoint *search* and *access-control* components by creating a single system-wide index and simply filtering search results for access control. This approach is ineffective as the index and query statistics subtly leak private information. The other available approach of using separate indices for each user is undesirable as it not only increases disk consumption due to shared files, but also increases the overheads of updating the indices whenever a file changes.

We propose a distributed approach that couples search and access-control into a unified framework and provides secure multiuser search. Our scheme (logically) divides data into independent access-privileges based chunks, called access-control barrels (ACB). ACBs not only manage security but also improve overall efficiency as they can be indexed and searched in parallel by distributing them to multiple enterprise machines. We describe the architecture of ACBs based search framework and propose two optimization technique that ensure the scalability of our approach. We also discuss other useful features of our approach – seamless integration with desktop search and an extenstion to provide secure search in untrusted storage service provider environments. We validate our approach with a detailed evaluation using industry benchmarks and real datasets. Our initial experiments show secure search with 38% improved indexing efficiency and low overheads for ACB processing.

1 Introduction

In recent years, the total amount of digital data has grown leaps and bounds, doubling almost every eighteen months [13]. As the cost of storage hardware drops, enterprises are storing more data and also keeping it for a longer period of time. It is for both business intelligence as well as regulatory compliance purposes. With this large amount of data available, keyword based search to quickly locate relevant data has become an essential IT capability.

According to many estimates, as much as 85% of enterprise data is stored in unstructured repositories like enterprise and local filesystems [12]. These filesystems have multiple users with different privileges to data and access is controlled using native access control mechanisms like Unix/Windows permissions models. This access control needs to be enforced even while searching through the data. As a simple case in point, a user should not be able to search through data that is not accessible to that user. Also, there are additional subtle requirements that complicate this process and if unhandled, can result in information leaks. For example, looking at the results of a query a user should not be able to extract any

information that could not have been inferred by that user by accessing the underlying filesystem. We refer to this principle as *Access Control Aware Search* or ACAS in short. Simply put, ACAS requires that no additional information can be extracted about the filesystem by using the search mechanism. More formally we define it as:

Definition: ***Access Control Aware Search (ACAS)***
Let $\mathcal{I}_F^U$ be the information that a user U can extract from the filesystem F by accessing it directly (dictated by access rights for U) and let $\mathcal{I}_S^U$ be the information that U can extract by searching on the indices over F over any period of time (based on the search mechanism). The access control aware search (ACAS) property requires that $\mathcal{I}_S^U \subseteq \mathcal{I}_F^U$.

Surprisingly most enterprise search products in the market, like Google Enterprise [30], Windows Enterprise Search [9], IBM OmniFind [17], do not satisfy the ACAS principle. These tools treat search and access-control as two disjoint components and can result in malicious users extracting unauthorized information using the search mechanism. In their approach, a single system wide index is created for all users and it is queried using traditional information retrieval (IR) techniques (the *search* component). Finally the results (the list of files containing query keywords) are filtered based on access privileges for the querying user (*access control*). However, the ordering and relevance score of results, typically based on Term-Frequency-Inverse-Document-Frequency (TFIDF) measures [33], reveal information that violates the ACAS property. Intuitively, since the index was created based on the lexicon and documents of the complete system, simple post-processing of results would fail to adequately protect system-wide statistics against carefully crafted attacks, for example, obtaining the number of files that contain the word "*bankruptcy*" even when the attacker does not have access to all files containing that keyword. We describe this issue and demonstrate an example attack in §2.3.

A technique that satisfies ACAS can be found in common desktop search products like Google Desktop [6] and Yahoo Desktop [7]. For multiple users on the desktop, these tools create distinct indices for each user on the system, with each user index including all files accessible to that user (the *access-control* component) and then querying only that index for the user (the *search* component). While this satisfies ACAS, it is highly inefficient as it requires every shared file to be indexed multiple times in the indices of each user that can access that file. Since in modern enterprises, a large amount of data is shared by many users, this approach not only causes greater disk consumption (due to increased index size), but the overheads of updating the indices when a file changes also become significant. In addition, the desktop search products incur high overheads to handle dynamism in access control and do not scale well with the number of users and the number of files in the filesystem.

In this paper, we propose a distributed enterprise search technique that couples search and access-control into a unified framework to provide secure and efficient search. We use a novel building block called access control barrel (ACB) that ensures access control aware search. An ACB is a set of files that have the same access privileges for users and groups in the system and by dividing[1] the filesystem data into independent ACBs, we can ensure that the index for a user is only derived from the data accessible to that user in the underlying filesystem, thus satisfying the ACAS requirement.

The ACB-based approach is also space and update efficient as it ensures that each file is included in only a single index. This *minimality* property makes it especially suitable for shared multiuser enterprise environments. In addition, the ACB-based approach elegantly accommodates dynamism in access control while incurring nearly zero cost overheads.

[1] This is a logical division and no physical data movement or modification is required.

Further, by dividing data into independent barrels, data indexing can be distributed to multiple machines for parallel processing. This can significantly reduce total indexing time. We also describe two optimization techniques that ensure the scalability of our approach even in complex enterprise environments.

Besides security and efficiency, the ACB approach has various other benefits in its design. It uses distributed information retrieval [20], which is well suited to searching over *data islands* - enterprise data stored under different administrative domains, most notably PCs and laptops. This is an especially pressing concern in modern enterprises. Finally, using off-the-shelf cryptographic mechanisms our approach can even be extended to support secure search in storage service provider (SSP) environments, where data and indices are stored at an external (untrusted) service provider. ACBs provide an efficient mechanism for key management in this environment. We describe this scheme in §5.2.

In summary this paper makes the following contributions:

- **Access Control Aware Enterprise Search**: In this paper, we have formally characterized the access control problem in enterprise search. We also propose a new Access Control Barrel (ACB) concept that prevents any information leaks to unauthorized users through search.
- **Space and Update Efficiency**: By ensuring that a file is included in a single index, our approach provides superior space and update efficiency. Further, we propose two optimizations that provide a mechanism to ensure scalability of our approach even in complex settings.
- **Distributed Enterprise Search Architecture**: In contrast to existing centralized approaches, we have developed a distributed architecture that parallelizes indexing and search for better performance and is thus better suited to modern enterprises with numerous underutilized machines.
- **Design Extensions**: We also describe two extensions of our ACB-based distributed design - seamless integration with desktop search and secure search for untrusted service provider index hosting.

The rest of the paper is organized as follows. In §2, describe relevant details of enterprise storage architecture and discuss current search approaches and their limitations. We describe the design of our approach in §3. We present a detailed evaluation of our approach in §4. We discuss various design benefits of our approach in §5.2. In §6, we describe the related work in this area. Finally, we conclude in §7.

2 Background and Limitation of Existing Approaches

In this section, we cover necessary background of the modern enterprise architecture, that serves as the model environment assumed in this paper. Later, we will discuss the limitations of existing enterprise search techniques in this model.

2.1 Modern Enterprise Architecture

Modern enterprises today are highly data-rich environments with storage capacities running into multiple terabytes and even petabytes in many cases. This data is stored into different repositories like databases, data warehouses, web servers or enterprise file servers. As pointed earlier, a majority of this data resides in flat filesystems [12] and we focus on indexing and search of such filesystem repositories in this paper.

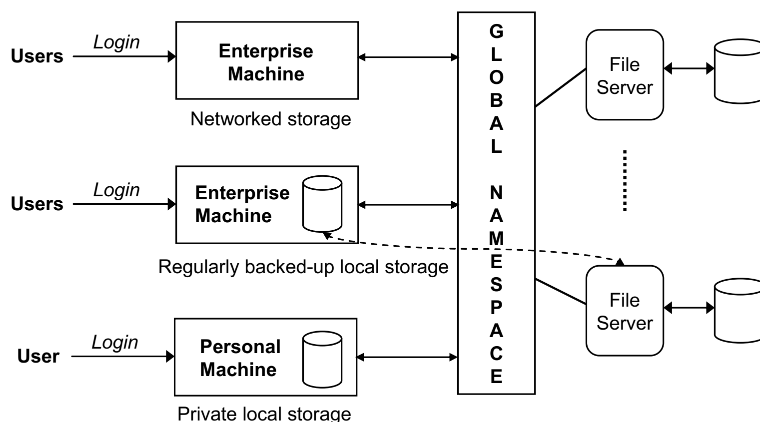

Figure 1: Enterprise Storage Architecture

A typical enterprise storage infrastructure has multiple file servers, most commonly accessed via *nix based Network File System (NFS) or Windows based Common Internet File System (CIFS) protocols. These file servers could be directly attached to storage systems (Network Attached Storage, NAS architecture as shown in Figure-1) or through a Storage Area Network (SAN). In the context of our work, we only require a common global namespace[2] which is widely supported in both architectures.

On the client side, this namespace is accessed by many always-connected (and mostly wired) enterprise machines. These could be user workstations, shared laboratory machines or application servers. Most of these machines have networked storage (for example, NFS mounted). Some of these machines might also have some local storage that is regularly backed up to global file servers. We call such machines *enterprise machines* since they are under administrative control and thus can be potentially leveraged for indexing and search tasks.

Other prominent client category consists of single-user personal machines like laptops. *Laptops* are notorious for hiding data from administrative control; with intermittent connectivity and frequent suspend-resume, there are no regular backups of their local storage. We call such clients *personal machines* and consider them to be unavailable for administrative usage. There is still substantial interest in the ability to index and search over their local data (as evident by recent desktop search initiatives [6, 7, 34]). However, most current systems offer isolated products for desktop search that can not be seamlessly integrated with *multiuser* enterprise search, an issue we attempt to resolve using our distributed search approach.

Lately, with increased costs of storage management, enterprises have increasingly looked at using outsourced storage services [8, 25]. In such environments, enterprise data is stored remotely (typically encrypted) at a storage service provider's location. Keyword based search on this data is even more important as data access is much slower (thus cannot do a brute-force `grep` [22] like search). Only downloading relevant content is economically motivated as well since in the revenue models for such services, enterprises typically pay a charge on the number of bytes accessed. For example, the Amazon S3 service [31] costs US $0.20 for each GB of data accessed. Outsourcing of search indices along with the data seems a natural choice to provide a single point of access. However, current enterprise approaches [30, 17, 9, 3] can not support this as they need a completely trusted search server to do access control at query runtime. In contast, by using

[2] A global namespace allows enterprise machines to view a common directory structure irrespective of where data is physically stored.

a novel access control barrel based primitive, our approach can be extended to support secure search over enterprise data even in untrusted service provider environments. We discuss this in §5.2.

2.2 Access Control

In an enterprise environment, not all data is accessible to all users. Access to data is controlled through the underlying filesystem's access control mechanisms. For example, NFS file servers follow the UNIX permissions model (described below) and Windows-based file servers follow the Windows permissions model. In this work, because of its widespread deployment and easier availability for experimentation, we use the UNIX based NFS architecture (also supported by other *nix flavors like Linux, FreeBSD). However, as it will be evident later, our work can be extended for Windows based access control models as well.

In UNIX access control model [27] each filesystem object (file, directory[3]) has an associated owner who controls the access to that object. This access can be granted to three classes of users: (1) *owner*, (2) *group*, and (3) *others*. The *owner* is the object owner, the *group* is the user group the owner belongs to (for example, a user group for students, faculty) and *others* are all users except the owner and the group. Permissions for multiple users and multiple groups can be set using POSIX Access Control Lists (ACLs) [14]. Further, the granted access is of three types:

- **Read:** For a file, this means that a user can read a file. For a directory, this allows a user to list its contents.
- **Write:** For a file, the write permission allows a user to write to it. For a directory, it allows a user to create, remove or rename directory contents.
- **eXecute:** For a file, the execute permissions allows running the file as a program. For directories, it allows users to traverse that directory and if they know the names of directory contents (due to read permissions or some out-of-band information exchange), they can then access those subdirectories/files.

In context of indexing and search, there is a need for a notion of **searchability**. Clearly *read* permissions on a file allow it to be searched and *write* permissions do not influence searchability (and are ignored in the process). The *execute* permissions for directories have a non-obvious influence on searchability. Users with *execute-but-not-read* permissions on a directory can access its contents only if they know the exact names of subdirectories or files. As this out-of-band notion can not be adequately and safely captured, consistent with the *nix `find/grep/slocate` [22] paradigms, we consider such directories as being not searchable. Formally, searchability is defined as:

Definition: ***Searchability* –** *A file, F is* ***searchable*** *by user u_i (or group g_m) if there exist* read ***and*** execute *permissions on the path leading to F and* read *permissions on F, for u_i (or g_m).*

2.3 Limitations of Existing Approaches

In this section, we explore existing desktop and enterprise search solutions – the two popular options for integrating access control and analyze their pros and cons from security and performance perspectives.

[3] We ignore symbolic links in this work and the filesdirectories they point to, are handled through their absolute filesystem path.

Most desktop search products like Google Desktop [6], Yahoo Desktop [7] and MSN Toolbar [34] integrate access control during indexing. Each user has a separate index for accessible files with duplication for files that are shared with other users. This ensures that each user has an index created from data that was accessible to that user, satisfying the ACAS requirement. In addition, there are no additional costs at query runtime for access control based filtering. However, this causes *additional* disk consumption of $\sum(n_i - 1) * I_i$ where n_i is the number of users accessing file F_i and I_i is size of index for F_i. Additionally, each update to F_i causes updates to n_i indices. In an enterprise, where n_i could be in hundreds or thousands, such costs can be prohibitive.

The enterprise search products like Google Enterprise Search [30], Coveo Enterprise Search [29] and IBM OmniFind [17] integrate access control at query runtime by creating a single system-wide index and filtering results based on access privileges of the querying user. This provides maximum space and update efficiency. However, querying will be more expensive, especially when access permissions for files in the query results are obtained at runtime, which requires disk I/O for inode lookups. Also importantly, these products do not satisfy the ACAS requirement and by carefully crafting queries, a user can obtain information about the underlying filesystem which could not have been inferred otherwise. Below, we describe an example attack that can determine the total number of files containing a particular keyword even when the attacker does not have access to all files containing that keyword. For example, an attacker could monitor the enterprise filesystem to see the number of files containing the word "*bankruptcy*". A sudden increase in the number of such files could alert him/her to sell off company stock, practically amounting to insider trading. This violates the ACAS property, as this information could not have been determined by the attacker through the underlying filesystem directly.

We assume that the relevance score of a result file f_i is computed by the standard TFIDF measure

$$rel(f_i) = \sum_{t_j \in Q} w_{ij} * w_{Qj} \qquad (1)$$

where t_j are the terms in the query Q and w_{ij} is the normalized weight of term t_j in f_i given by

$$w_{ij} = \frac{o_{ij} * log\left(\frac{|N|}{n_{t_j}}\right)}{\sqrt{\sum_{t_k \in f_i} (o_{ik})^2 * \left(log\left(\frac{|N|}{n_{t_k}}\right)\right)^2}} \qquad (2)$$

where o_{ij} is the number of occurrences of term t_j in f_i, $|N|$ is the total number of files in the system and n_{t_j} is the number of files that contain t_j. w_{Qj} is defined similarly.

The attacker, Alice, wishes to know the number of documents that contain the term t_q (e.g. "*bankruptcy*"). The attack works in three steps. First, Alice picks two *unique* terms t_1, t_2 (no file contains these terms) and creates two new files: f_1 containing terms $\{t_1, t_2\}$ and f_2 containing terms $\{t_2, t_q\}$. Note that after creating the files, o_{11}=o_{12}=o_{22}=o_{2q}=1, n_{t_1}=1 and n_{t_2}=2. In the second step, she queries for term t_1 and from (1) and (2) she can calculate $|N|$

$$|N| = 2^{\frac{1}{1-\sqrt{\frac{1}{rel(f_1)^2}-1}}} \qquad (3)$$

In the final step, Alice queries for term t_q and calculates n_{t_q} from (1), (2), (3). This completes the attack.

$$n_{t_q} = 2^{\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}} * |N|^{\left(1-\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}\right)}$$

Such attacks are possible on most TFIDF based measures including the popular measure Okapi BM25 [28]. Additionally, even when relevance scores are not returned as part of the result, good approximations to n_{t_q} can be obtained by exploiting ordering of the results [3]. A recent effort [3] describes an ACAS compliant approach that uses a complex query transformation at runtime for access control. Additionally, it requires to maintain access control lists for all files in the filesystem in-memory which is extremely inefficient for large enterprise environments. Also, as we discuss in later sections, our distributed search approach provides various design benefits.

3 Distributed Secure Enterprise Search

Most of the desktop search products are secure but inefficient for enterprise search, whereas enterprise search products are insecure in terms of ACAS. Bearing these issues in mind, we describe the design and implementation of our distributed approach to enterprise search based on the concept of access control barrels (ACBs), which provides both security and efficiency.

3.1 Design Overview

The main design principle of our approach is to efficiently integrate access control into the indexing phase such that the indices used to respond to a user's query are derived only from the data accessible to that user. This will ensure that we satisfy the ACAS requirement and do not have to do access control based filtering on query results. We accomplish this goal with a pre-processing step that (a) constructs a user access hierarchy for the users and user groups in the system (§3.1.2) and (b) logically divides data into access-privileges based *access control barrels (ACB)*. We first provide a brief description of ACBs and then describe in detail how they work in conjunction with the user access hierarchy.

3.1.1 Access Control Barrels

An ACB is a set of files that share common searchability access privileges (as defined in §2.2). That is, all files contained within an access control barrel can be accessed (and thus searched) by the same set of users and user groups. For example, one barrel could contain files accessible to user *bob* and another for a user group *students*. Intuitively, the idea of barrels is that if we can efficiently create collections of files based on their access privileges, to provide secure search to a user, we can pick the collections that this user has access to and serve the query using only those indices.

This might sound similar to the index-per-user (IPU) desktop search approaches [6, 7, 34] where all files accessible to a user are grouped into a single collection and files accessible to multiple users are duplicated in their collections. ACBs avoid their inefficiencies by following an additional neat property of *minimality*. This property ensures that each file can be uniquely mapped to a single barrel, avoiding duplicating them in multiple collections (we defer the discussion of implementing such minimal ACBs to the next section). Now, files accessible to multiple users are grouped into *shared* collections and secure search for a user combines the user's private collections with these shared collections using distributed information retrieval [20]. This is efficiently accomplished using the user access hierarchy which is described next. We will also compare the ACB based approach with the index-per-user approach in detail in §4.1.

3.1.2 User Access Hierarchy

The user access hierarchy data structure has two main tasks: (1) provide a mechanism to map files to access control barrels, and (2) provide techniques to efficiently determine all barrels that contain files searchable by a user. In what follows, we first give a high level description of the data structure and later detail how the hierarchy is constructed for *nix permissions model.

For most access control models a user is associated with two types of credentials: (i) a unique user identifier (uid), and (ii) one or more group identifiers (gid) corresponding to the user's group memberships. We represent the set of all such user and group credentials as a directed acyclic graph called Access Credentials Graph or ACG in short. For example, there could be a node for credential uid_{bob} or $gid_{students}$. Every node V_i in this graph is associated with a corresponding barrel ACB_i. Our example nodes uid_{bob}, $gid_{students}$ are associated with barrels containing files with searchability privileges to user *bob* and group *students* respectively. Now, mapping files to barrels is equivalent to assigning files to a node in ACG (the first task mentioned above).

For a node, V_u (associated with the uid credential of a user u), let V_u^* denote the set of all nodes in the *directed* graph ACG that are reachable from the vertex V_u. Our construction of the graph ACG will ensure that a file F is searchable for a user u if and only if F is assigned to some vertex $v \in V_u^*$. With this property, the results for user u's query can be computed by combining indices from barrels associated with nodes in V_u^*. The set V_u^* can be efficiently determined using a simple depth or breadth first search on the graph ACG. This accomplishes the second task of the user access hierarchy data structure.

Next, we explain in detail the process of constructing the graph ACG with aforementioned properties from *nix-like user credentials. In a *nix-like access control model a credential C can be expressed in Backus Naur Form (BNF) as:

$$
\begin{aligned}
C &= root \mid all \mid P \\
P &= uid \mid gid \mid P \wedge P \mid P \vee P \qquad (I)
\end{aligned}
$$

Note that $root$ is a special user with super-user privileges and all indicates a credential for all users and groups. We need the $\vee$ operator on the principles to handle POSIX Access Control Lists [14] that allow associating multiple users and groups with a file F (as opposed to the usual $\{owner, group, others\}$ model (see Section §2.2)). We need the $\wedge$ operator on the principles to handle the implicit conjunction operation that occurs while traversing the directory hierarchy leading to file F (for example, directory `X/Y` where `X` has access only for user group *students*, and `Y` has access only for user group *grad-students*; only users that belong to both groups can access data under `Y`). We define an implication operator $\Rightarrow$ which specifies if one credential can *dominate* another. For example, $\forall u$, $root \Rightarrow u$ says that $root$ can access data that any other user can.

Permissions on a file can also be expressed based on a credential defined as above. For example, for a file F that allows access to users x, y and group z, we say that it has a credential C_F=$\{uid_x \vee uid_y \vee gid_z\}$ and is interpreted to say that access is allowed to users who have a credential that dominates this file's credential (user x has access to F since $uid_x \Rightarrow C_F$). Now, if we can create a barrel for each such file credential in the system, we can uniquely map a file to a barrel, achieving ACB minimality. While theoretically the total number of barrels (one for each possible access control setting, thus exponential in number of users and groups) can be very large, practically, this is hardly the case as many files in the system share common file credentials. Other studies have also made similar observations, for example, the

filegroups concept of Plutus [18] uses this to ease key management for encrypted data storage. Regardless, in §3.2, we will describe two optimization techniques that address this potential scalability issue.

Finally, we construct the graph ACG as follows. First, the set of vertices and edges are initialized by adding vertices for all user and group credentials and adding edges for the simple $\Rightarrow$ relationships: $root \Rightarrow u \, \forall u \in U$; $u \Rightarrow g \, \forall g \in G(u)$; for any group g, $g \Rightarrow all$, where $G(u)$ denotes the set of all groups to which the user u belongs to. Formally,

$$\begin{aligned} V_{ACG} &= \{V_{root}, V_{all}\} \cup \{V_u \mid \forall u \in U\} \cup \{V_g \mid \forall g \in G\} \\ E_{ACG} &= \{V_{root} \to V_u \mid \forall u \in U\} \cup \, \{V_u \to V_g \mid \forall u \in U, \forall g \in G(u)\} \cup \, \{V_g \to V_{all} \mid \forall g \in G\} \end{aligned}$$

where $\to$ indicates a directed edge in the graph.

Next, the $\vee$ or $\wedge$ nodes are added when we encounter files with such credentials. This is done during the pre-processing step while assigning files to their appropriate barrels (as described in §3.3). For each such file, we insert a new vertex V_C for the file's credential C. Then, we find the set of all vertices whose credentials minimally dominate the new credential C (say, $minDom(C)$) and for every such vertex, $V \in minDom(C)$, we add an edge from vertex V to V_C. Note that this guarantees that the vertex V_C is reachable from all vertices that have a dominating credential, by the transitive nature of the $\to$ operator on the graph ACG. Similarly, we find the set of all vertices whose credentials are maximally submissive to the new credential C (say, $maxSub(C)$). For every such maximally submissive credential $V \in maxSub(C)$, we add an edge from the vertex V_C to V. Additionally, we remove redundant edges between vertices in $V_1 \in minDom(C)$ and $V_2 \in maxSub(C)$ if V_2 is now reachable from V_1 via the new vertex V_C. Formally, it can be represented as:

$$\begin{aligned} Dom(C) &= \{V_{C'} \mid C' \Rightarrow C\} \\ minDom(C) &= \{V \in Dom(C) \wedge \neg\exists V' \in Dom(C), V \in Dom(V')\} \\ Sub(C) &= \{V_{C'} \mid C \Rightarrow C'\} \\ maxSub(C) &= \{V \in Sub(C) \wedge \neg\exists V' \in Sub(C), V \in Sub(V')\} \\ E_{ACG} &= E_{ACG} \cup \{V \to V_C \mid \forall V \in minDom(C)\} \cup \{V_C \to V \mid \forall V \in maxSub(C)\} \\ E_{ACG} &= E_{ACG} - \{V_1 \to V_2 \mid \forall \, V_1, V_2, V_1 \in minDom(C) \wedge V_2 \in maxSub(C) \wedge V_1 \to C \in E_{ACG} \wedge \\ &\quad\; C \to V_2 \in E_{ACG}\} \end{aligned}$$

Using this access hierarchy graph, we can map all files to their appropriate barrels and also identify barrels searchable by a particular user (equivalent to finding V_u^* – a simple depth first search operation on ACG).

3.1.3 Dynamic access Control

In this section we study the effect of dynamic access control policies on the IPU and the ACB approach. In this section we the effect of two types of access control updates: adding a new group to a user and adding a new $\wedge$-group to a file. In the ACB approach adding a new group g to a user u just requires a new edge to be added from the vertex V_u to the vertex V_g incurring nearly zero cost. On the other hand, in the IPU approach addition of a new group makes several files which were previously inaccessible to user u becomes accessible now. Now, the indices of all such files need to be added to user u's search index and thus incurs very high cost. In addition, the indexing mechanism must examine all the files in the filesystem to determine which files are now newly accessible (or inaccessible if a group membership is removed)

to user u, thereby incurring heavy i-node lookup costs. Clearly, this largely limits the scalability of the IPU approach in large filesystems.

Next, we study the effect of adding or removing a $\wedge$-group to a file's access control expression. Adding or removing a $\vee$-user to a file's access control expression follows a very similar procedure. Let $B(f)$ denote the access control expression for file f. When a new $\wedge$-group g is added the new expression $B'(f) = B(f) \wedge g$. In the ACB approach, we first check if the vertex $V_{B'(f)}$ exists in the user access hierarchy; if it does not exist we create the vertex and add edges to its parents and children vertices in the user hierarchy. Then, we remove the index for file f from $V_{B(f)}$ and add it to $V_{B'(f)}$. Additionally, if $V_{B(f)}$ has no files associated with it, it is deleted and the edges are appropriate modified. Note that this operation involves only fast and efficient operations on the ACG and only one file index copy. On the other hand, in the IPU approach when a new $\wedge$-group g is added to a file f, a large number of users that could previously access file f can no longer access it. Similarly, when a $\wedge$-group g is removed from a file f, then a large number of users who were previously unable to access file f are permitted to access it. In addition, the indexing mechanism must examine all the users in the filesystem to determine which users are newly permitted (or denied) access to the file f.

3.1.4 ACB Minimality

In this section, we formally state the minimality property satisfied by our ACB construction in Section 3.1.1. We use this claim 3.1 in §3.2 to present optimization techniques on our ACB approach.

Claim 3.1. *It is impossible to reduce the number of ACBs without either duplicating files in barrel indices or violating the ACAS property.*

Proof. We can prove this by a simple contradiction argument. Let ACB' denote a set of access control barrels such that the number of barrels in ACB' is smaller than ACB and it contains exactly one copy of each file in a barrel index and it respects the ACAS property. Since ACB' has smaller number of barrels, there exists a barrel $b' \in ACB'$ such that it has two files f_1, $f_2 \in b'$, where f_1 and f_2 belong to two distinct barrels b_1 and b_2 in ACB (by pigeon hole principle). Since f_1 and f_2 are in different barrels in ACB, the files must have different access control expressions. Hence, there may exist a user u such that u can access only file f_1 but not file f_2.

Now, for the file f_1 to be searchable by user u, there has to be some barrel bar in ACB' such that $f_1 \in bar$ and the barrel bar is reachable from the vertex V_u on the ACG. Since there is only one copy of the file index f_1 in ACB', the barrel b' must be reachable from V_u on the ACG for ACB'. Since the barrel b' is reachable from V_u all the files in the barrel b' must be accessible to the user u. Clearly, allowing the user u to search file f_2 violates the ACAS property. Thus, in order to reduce the number of ACBs, we have to allow either duplication of files indices in barrel or compromising the ACAS property. □

3.1.5 Bounds on the Number of Barrels

In this section, we present theoretical bounds on the total number of barrels and the average number of barrels accessible to a user. Since a user's index is composed of all barrels that the user can access, it is important that the number of barrels per user is reasonable. Let us for the sake of simplicity consider only $\wedge$-group based access control expressions for files; the arguments for $\vee$-user based access control expressions is very similar. Let $B(f)$ denote the access control

expression on file f. Let the number of literals ($\wedge$-groups) in $B(f)$ be chosen from the range $(1, ngf)$ using some arbitrary distribution.

First, we note that the maximum number of ACBs is bounded by the minimum of the number of files ($|F|$) and the number of *possible* access control expressions. Note that we create a new ACB only if there is at least one file that belongs to that barrel. In the worst case, when every file belongs to a different barrel the number of ACBs is $|F|$. The number of possible access control expressions for files is given by: $\sum_{i=1}^{ngf} {}^{|G|}C_i$, where ${}^{y}C_x$ denotes the number of ways of choosing x balls from y non-identical balls. For example, if $G = \{g_1, g_2\}$ the set of possible access control expressions are $\{g_1, g_2, g_1 \wedge g_2\}$. If ngf is $O(1)$, then the number of ACBs is only polynomial in $|G|$. However, the worst case bound for the number of ACBs is $min(|F|, 2^{|G|})$. A large number of ACBs incur only a marginally larger user hierarchy graph maintenance cost. Using a simple depth first search (DFS) algorithm we can limit the effort required to search any vertex/edge in this graph to $O(|U| + |G|)$. In addition, one can build a hash table on top of the graph data structure to reduce the search time to $O(1)$.

The runtime overhead and the quality of distributed information retrieval algorithms is determined by the number of barrels that need to be searched to handle a user's search query. The number of ACBs per user is bounded by: $\sum_{i=1}^{ngf} {}^{|G(u)|}C_i$ (note that ${}^{y}C_x = 0$ if $y < x$), where $G(u)$ denotes the group membership for user u. Typically, $|G(u)| \ll |G|$ and $ngf \ll |G|$ and thus, the number of ACBs per user will be very small and in the worst case bounded by $2^{|G(u)|}$. Note that bound on the number of ACBs per user (unlike the bound on the total number of ACBs) is independent of $|G|$ and thus scales better.

3.2 Scalability Optimizations

In most real enterprise environments the number of barrels per user is typically very small (6–8 on average in our observations for two enterprise filesystems; similar observation was made in [18]). However, theoretically this number is exponential in the number of users and user groups. In this section we describe two optimization techniques that can preserve the scalability of our approach even in such rare hostile setups. We first start with two important properties of our approach that help design the two optimizations.

We base out optimizations based on our ACB minimality claim 3.1. Our first optimization trades off the number of barrels with the number of copies of a file index and the second technique trades off the number of barrels while allowing controlled violation of the ACAS property. Both techniques provide a control mechanism for administrators to choose appropriate trade-offs. The optimizations transform the access control graph (ACG) with the goal of decreasing the number of ACBs. However, such transformations must preserve *searchability*, that is, if a file f is accessible to user u then the file f must be searchable. Using ACBs this implies that if a file f is accessible to user u, then in any transformed ACG, the file f belongs to some barrel b such that b is reachable from V_u on the ACG ($b \in V_u^*$). We call this property *reachability* on the ACG.

3.2.1 File Index Duplication

In this section we propose algorithms to reduce the number of ACBs and satisfy the ACAS property at the cost of maintaining duplicate file indices. Let us consider any vertex V_C in the ACG such that the credential $C \neq u$, for any user $u \in$

U. One can eliminate the vertex V_C from the ACG (thereby decreasing the total number of barrels by one) by adding all the file indices in V_C to every vertex $v \in minDom(V_C)$, where $minDom$ is defined in Equation 1. Note that if $C \neq u$, then $minDom(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in minDom(V_C)$. If V_C is reachable from some vertex V_u (for user u and $C \neq u$) then at least one vertex $v \in minDom(V_C)$ is reachable from V_u; thus the above construction satisfies reachability on the ACG and thus preserves searchability. The construction preserves the ACAS property since the credential $domC$ associated with any vertex $v \in minDom(V_C)$ dominates the credential C ($domC \Rightarrow C$). Hence any user u that satisfies the credential $domC$ also satisfies the credential C. Hence, the above construction eliminates the vertex V_C at the cost of retaining $|minDom(V_C)|$ copies of file indices for each file in V_C.

In our implementation we define a tuneable parameter $minf$ – the minimum number of files per barrel. Our ACB construction in Section 3.1.1 achieves $minf$ = 1. If a larger $minf$ is chosen the number of barrels decreases at the cost of more duplication of files indices. Given the parameter $minf$ we present a greedy algorithm to reduce the number of barrels as follows. (i) Sort the barrels in increasing order on their size (number of files in the barrel) $b_0, b_1, \cdots b_k$. (ii) Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to u for any user $u \in U$. If there is no such barrel the procedure terminates. (iii) Eliminate the barrel b_i by suitably replicating all its file indices. Note that this may change the size of other barrels; so we resort the barrels according to their size and repeat the procedure. One can use an induction on the number of vertices eliminated and the arguments described above to show that this procedure is guaranteed to reduce the number of ACBs without compromising the ACAS property. Observe that setting $minf = \infty$ the above procedure would terminate with the $|U|$ barrels where each barrel b_u is the per-user index as generated by the IPU algorithm. This ensures that for any finite $minf$, our algorithm would have fewer file index duplicates for any file f when compared to the IPU approach.

3.2.2 Access Control Optimization

Our second technique reduces the number of ACBs while maintaining only one copy of each file index at the cost of violating the ACAS property. Let us consider any vertex V_C such that $C \neq all$. One can eliminate the vertex V_C from the ACG by adding all the file indices in V_C to some vertex $v \in maxSub(V_C)$, where $maxSub$ is defined in Equation 1. Note that if $C \neq all$, then $maxSub(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in maxSub(V_C)$. If V_C is reachable from some vertex V_u (for user u) then all vertices $v \in maxSub(V_C)$ is reachable from V_u; thus our construction satisfies reachability on the ACG and thus preserves searchability. However, there may exist a user u' such that a vertex $v \in maxSub(V_C)$ is reachable from $V_{u'}$ but not the vertex V_C. This is possible because the credential C dominates a credential $subC$ associated with vertex $v \in maxSub(V_C)$. Hence, the above construction maintains exactly one copy of every file index, but may violate the ACAS property. Unlike the single-index approach that violates the ACAS property for all the files in the file system, our approach allows us to control the number of such violations.

In our implementation we define a tuneable parameter $minf$ – the minimum number of files per barrel. Given the parameter $minf$ we present a greedy algorithm to reduce the number of barrels as follows. (i) Sort the barrels in increasing order on their size (number of files in the barrel) $b_0, b_1, \cdots b_k$. (ii) Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to all. If there exists no such barrel the procedure terminates. (iii) Eliminate the barrel b_i. This can be achieved by copying the file indices in b_i to at least one vertex $v \in maxSub(V_C)$, where C is the credential associated with barrel b_i. If $|maxSub(V_C)| > 1$, one can randomly pick a vertex v from

$maxSub(V_C)$. However, we heuristically pick a vertex v such that it satisfies the $minf$ requirement while incurring only a small number of access control violations. (iv) Eliminating a barrel may have changed the size of other barrels; so we resort the barrels according to their size and repeat the procedure.

Our first heuristic picks the vertex v that has the smallest barrel associated with it. This heuristic clearly favors our goal of achieving at least $minf$ files per barrel. Our second heuristic attempts to reduce the number of files violating the ACAS property by picking a vertex v whose credential is the least *popular*. For example, let us suppose that the credential associated with v is a $\wedge$-group $cred = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_k}$. We measure the popularity of the credential $cred$ as $pop(cred)$ = $\prod_{j=1}^{k} pop(g_{i_j})$, where popularity of a group g is determined by the number of members in the group (normalized by the total number of user $|U|$). Similarly, we measure the popularity of a $\vee$-user credential $cred = u_{i_1} \vee u_{i_2} \vee \cdots \vee u_{i_k}$ as $pop(cred) = \frac{k}{|U|}$. Clearly, the less popular a credential $cred$ is, the smaller number of users that satisfy $cred$; hence, fewer users can reach the vertex v from V_u. Note that every user u can that reach V_C can reach $v \in maxSub(V_C)$; hence the approach does not compromise on searchability. This approach attempts to minimize the number of users u' that can reach $v \in maxSub(V_C)$, but not the vertex V_C itself, thereby reducing the number of ACAS violations. Observe that setting $minf = \infty$ this algorithm reduces the single index approach and thus allows any user u to search over all the files in the enterprise file system. Similar to the single index approach, we incorporate post processing to suppress files that are inaccessible to a user u. However, as shown in Section 2.3 this algorithm is still vulnerable to statistical inference attacks. Nonetheless, our heuristics hopes to minimize the number of files that are susceptible to such inference attacks.

These two optimizations can be used to improve the scalability of the system in rare environments where the variation in access control settings increases the number of ACBs per user. Further, we allow an administrator to make an intelligent decision on the choice of the optimization strategy. For instance, files with high update rates may use only the second optimization technique. This ensures that we have only one copy of the file index and thus keeps the update costs low. Similarly, files with lot of critical information may use only the first optimization technique. This ensures that there are no ACAS violations on the critical file data. In the following section, we integrate our ACB based technique with the architecture of our indexing and search system.

3.3 Architecture and System Implementation

So far, we have introduced the concept of access control barrels (ACBs) and the user access hierarchy as a tool to (a) efficiently map files to ACBs and (b) determine accessible ACBs for a querying user. In this section, we will explain the overall architecture of our indexing and search system and how it fits into the enterprise infrastructure.

Figure-2 shows the architecture. One enterprise machine is chosen as a global orchestrator[4] and is responsible for managing the distributed environment. We will explain its various components in the next subsections. All other participating enterprise machines run a thin client version of the system and are responsible for barrel indexing and query processing for local users. Finally, the personal machines (outside admin control) that intend to integrate their local desktop search with enterprise search run a local orchestrator agent (explained later).

[4]It is possible to develop a decentralized orchestrator by distributing responsibilities of barrels to machines, e.g. by using distributed hash tables.

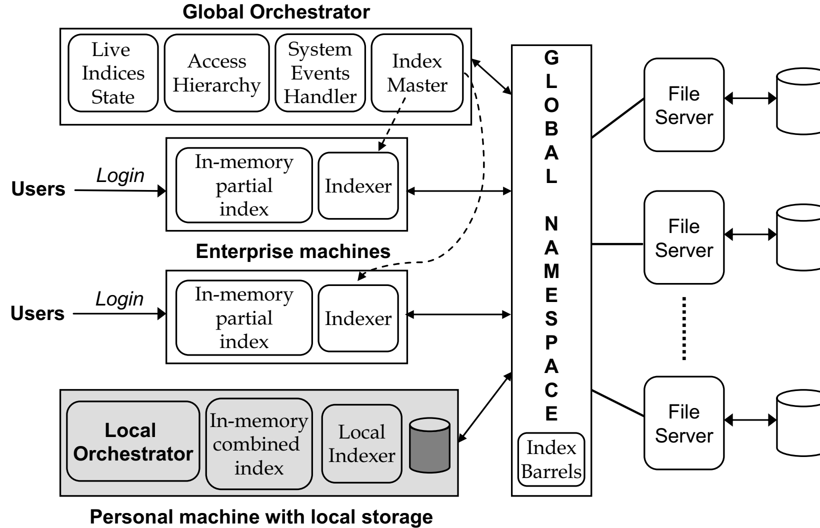

Figure 2: Distributed Indexing and Search

3.3.1 Pre-processing: Creating ACBs

As part of the pre-processing step, we first create the basic ACG from the user and user groups in the system as described in Equation-(II) above. For *nix systems user and group information is obtained from */etc/passwd* and */etc/group*. Next, we initiate a filesystem traversal for all data that needs to be indexed. This is required for mapping files to ACBs (represented by a vertex in ACG). During the traversal, we associate each file with a vertex in the directed ACG graph based on its searchability privileges. This mapping is done by finding the vertex that has the same credential as the file (e.g. V_{bob} for credential uid_{bob}). If the file has a $\vee/\wedge$ credential, a new node is added to the access hierarchy and the file is mapped to that node. At the end of this filesystem traversal, we have all barrels in the system and the list of files that are contained in each such barrel. These lists are written to per-barrel files, that are *securely* stored in the enterprise global namespace with access privileges only to the superuser. This stored file is the embodiment of our abstract ACB concept. This completes the pre-processing step and is usually performed by a single enterprise machine – the *global orchestrator*, which stores the user access hierarchy.

3.3.2 Indexing

After creating ACBs, the next step is to index documents for each barrel. These ACBs can be indexed independently unlike the single index approach where the computation of TFIDF statistics requires centralized indexing of data. The *index master* in the global orchestrator distributes this barrel indexing task to participating enterprise machines. As the barrels are stored in a global namespace and accessible to all enterprise machines, the orchestrator only needs to pass the barrel IDs to these machines. The orchestrator can easily optimize available resources by doing an intelligent distribution of barrels to machines (ensuring no single machine is overly loaded). As we show later in §4, this indexing task distribution provides excellent savings.

On receiving commands from the index master, the *indexer* component of enterprise machine agents retrieve the barrels from the global namespace and start indexing documents. An index is typically comprised of: (a) vocabulary for

words that appear in the documents and (b) a words to filename mapping along with their TFIDF statistics used later for ranking. Once indexed, these indices are stored back into the global namespace. Our access privileges based design of barrels provides a natural way of storing indices securely in this namespace. The index files are stored with the same privilege as the files contained in that barrel (all files in a barrel have the same privileges)[5]. This allows only the users that had access to files of a barrel (and thus can search through that barrel) to obtain these indices and provides a natural security mechanism for storing these indices using the underlying filesystem access control.

3.3.3 Search

In our approach, querying and search can also be handled in a distributed fashion. When a user logs into an enterprise machine, the agent on that machine retrieves the indices that are accessible to that user, from the global namespace and caches them in memory. Now whenever a user queries these indices, search can be handled completely in a local environment, saving on (a) query response time and (b) resource requirements of a centralized search server. Note that all available enterprise search products today [30, 17, 29, 3] have to use a highly capable search server (or a cluster) in order to deal with enterprise environments and querying always involves a network hop. In contrast, by integrating access control in a distributed fashion, we can reduce such requirements. However, our approach has an overhead of combining the multiple barrel indices using distributed information retrieval techniques. But unlike the centralized index approaches, we do not have to perform any access control on query results. Our experiments in §4 shows how these two factors tend to balance out.

Please note that distributed search yields benefits when enterprise machines have enough resources to handle its local users indices. In situations where there are only a few enterprise machines or all users log onto a single server, it would be more efficient to use a central search server which can be located at the global orchestrator.

3.3.4 Handling Updates

In an enterprise environment, there will be regular updates to data files and access privileges to data and the system needs to handle them appropriately. This task is handled by the global orchestrator which subscribes to all filesystem event notifications using available tools like inotify [23]. Once an event is received the orchestrator might need to make various kinds of changes. Change of file content is handled at a per-barrel basis by requiring that barrel indices to be appropriately modified (it usually does not require indexing the entire barrel again). This event is common to all enterprise search techniques and the ACB based approach does not incur additional overheads. In case of events when access permissions are modified that impact searchability, a document might need to be removed from one barrel and added to another (most indexers can handle this in an incremental manner as well), making it a low-cost event. Another filesystem event unique to us is the case of user/group membership modification, in which case the access hierarchy needsto be adjusted. A user/group addition is handled by adding a new node and corresponding edges (as done during initial ACG construction). Group membership modification is handling by changing the edges in the directed graph. Finally, a user/group deletion is handled by removing the appropriate node and all edges coming into or out of that node. The changes in groups and their membership are infrequent events and corresponding operations on ACG are efficient.

[5]The $(u_i \vee u_j)$ barrels are handled by using POSIX ACLs for (u_i, u_j) on the indices and $(g_k \wedge g_m)$ are handled by keeping indices under directory hierarchy `X/Y` with `X`, `Y` having privileges for g_k, g_m respectively.

Notation	Description	Default
$\|F\|$	Number of Files	10^7
$\|U\|$	Number of Users	10^3
$\|G\|$	Number of User Groups	32
pop	Group Popularity	Zipf(1, ng)
ngu	Number of Groups per User	Zipf(2, 10)
ngf	Number of $\wedge$-groups per File	Zipf(2, 4)
nuf	Number of $\vee$-users per File	Zipf(2, 4)

Figure 3: Parameters

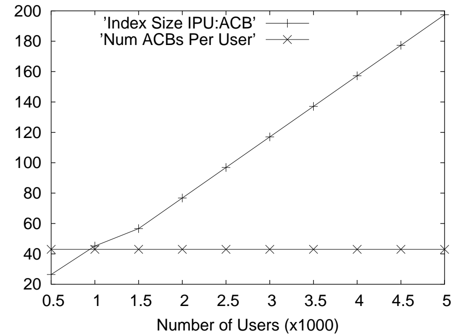

Figure 4: Num Users

Note that an update could occur in a barrel that is in-memory at one of the enterprise machines. In order to handle such scenarios, the orchestrator keeps state information of all such in-memory live indices and notifies the appropriate enterprise machine to flush the cached barrel index in that case and reload it after it has been suitably updated.

4 Evaluation

In this section, we present a detailed evaluation of our approach. We compare our approach analytically with the other SAC-compliant index-per-user approach in §4.1. In §4.2 shows the effectiveness of our optimization algorithms described in §3.2. In §4.3, we describe the datasets used in our indexing and querying experiments. §4.4 describes the indexing experiments including barrels pre-processing and §4.5 describes the querying and search related experiments. All experiments were done on a Pentium-III Linux machine with 512 MB RAM and all storage mounted via NFS. All numbers below have been averaged over multiple runs.

4.1 Comparison with Index Per User Approach

In this section, we analytically compare the performance and scalability of the ACB approach against the index per user (IPU) approach. For comparison purposes, we use synthetic data. The key parameters in our data are summarized in Table 4.1. We observe that the analysis is the same for both $\wedge$-groups and $\vee$-users and thus consider only $\wedge$-groups in this Section. We use Zipf(a, b) to denote a Zipf distribution with parameter γ = 1 that is truncated to the range (a, b). Hence, we choose ngu the number of groups a user is a member of using a Zipf distribution on the range $(2, 10)$; we then choose ngu groups from the set G using Zipf(1, $|G|$) and without replacements. Similary, we choose ngf the number of $\wedge$-groups per file using Zipf(2, 4); we then choose ngf groups from the set G using Zipf(1, $|G|$). The access control rule for the file is assumed to be an $\wedge$ over all the chosen groups.

4.1.1 Static Access Control

In this section, we measure the number of ACBs and the number of ACBs per user. We also compare the ratio of the size of indices maintained by the IPU and ACB approach. Figures 4 shows the scalability of our approach with the number of users $|U|$. In the IPU approach, the index size grows with the number of users in the system, typically because more users share a file. On the other hand, the ACB approach maintains exactly one copy of each file index and the total number

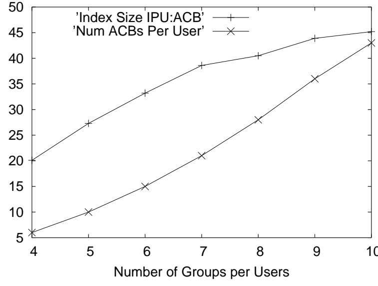

Figure 5: Num Groups per User

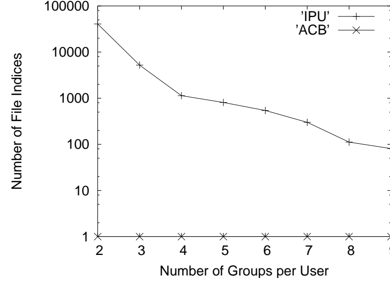

Figure 6: Updating Group Membership

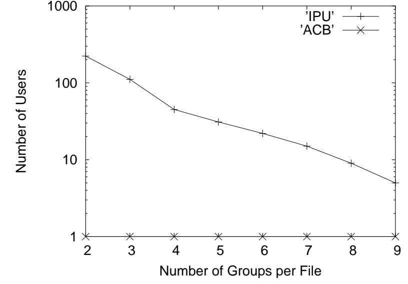

Figure 7: Updating File Access Control Expression

of barrels (and thus the average number of barrels per user) is *independent* of $|U|$. Figure 5 shows the scalability of our approach with ngu the number of group memberships per user. As ngu increases, so does the number of users that share a file and thus the index size in the IPU approach. In the ACB approach, as ngu increases it results only a marginal increase in the number of ACBs per user.

One should observe that our approach maintains exactly one copy of index for each file. Hence, when a file is updated at most one index needs to be modified. The IPU approach maintains multiple copies of each file index, one for every user who is permitted to access that file. Hence, when a file is updated the IPU approach has to update several indices (an average of 45.2 using default settings in Table 4.1) thereby incurring heavy disk access costs. In contrast the ACB approach incurs a small overhead of using distributed IR solutions to merge search results from a small number of barrels.

4.1.2 Dynamic Access Control

In this section we study the effect of dynamic access control policies on the IPU and the ACB approach using two types of access control updates: adding a new group to a user and adding a new $\wedge$-group to a file. As described in §3.1.3 the ACB approach requires a new edge from vertex V_u (for user u) to a vertex V_g (for group g) to reflect this access control update. In the IPU approach, adding a new group membership to a user u may permit the user to access files that it could previously not access. Now, the indices of all such files need to be added to user u's search index. Figure 6 shows the number of file indices that need to be copied when a user group is added to a user assuming the user is already a member of ngu groups. Note that the figure can also be interpreted as the number of file indices that need to be removed from a user's search index when a user that is already a member of $ngu + 1$ groups looses membership to one of its groups.

Next, we study the effect of adding or removing a $\wedge$-group g to a file's access control expression $B(f)$. As described in §3.1.3 the ACB approach requires only small manipulations on the ACG to reflect this access control update and thus incurs nearly zero cost. On the other hand, in the IPU approach when a new $\wedge$-group g is added to a file f, a large number of users that could previously access file f can no longer access it. The IPU approach needs to update the indices for all such users. Figure 7 shows the number of users whose search indices need to be updated when a file's access control expression is updated assuming the number of $\wedge$-groups in $B(f)$ is ngf.

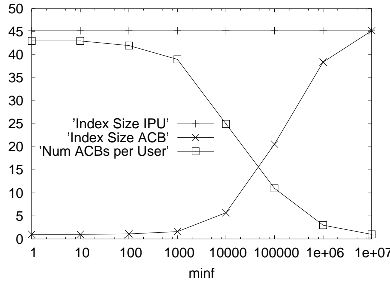

Figure 8: Optimization I

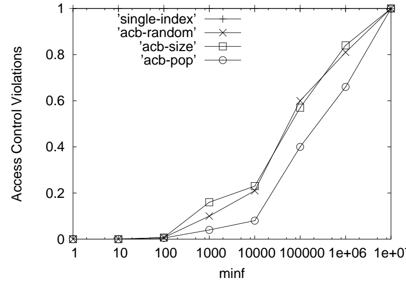

Figure 9: Optimization II: Index Size

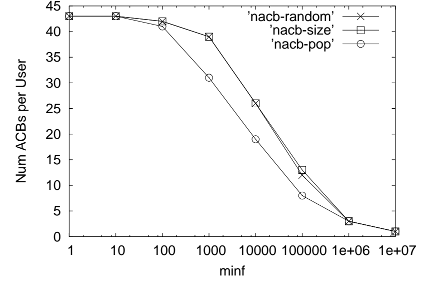

Figure 10: Optimization II: Num ACBs

4.2 Optimization Techniques

In this section, we show the effectivness of our optimization techniques in decreasing the number of barrels. Figure 8 shows the effectives of our first optimization technique that preserves the SAC property while maintaining multiple copies of each file index. We plot the tuneable parameter $minf$, the mininum number of files per barrel, on the x-axis. As described in §3.2 our index size increases with $minf$ and slowly reaches the index size of the IPU approach as $minf \rightarrow |F|$. This shows the flexibility of our technique in reducing the number of barrels while incurring significantly lower costs than the IPU approach.

Figures 9 and 10 show the effectiveness of our second optimization technique that maintains exactly one copy of every file index while violating the SAC property for some files. We have evaluated the effectiveness of our algorithm using three heuristics: `random` chooses a vertex v at random from $maxSub(V_C)$, `size` picks the vertex $v \in maxSub(V_C)$ that has the least number of files, and `pop` picks the vertex $v \in maxSub(V_C)$ that causes the least number of SAC violations. Figures 9 and 10 show that popularity based approach performs best in terms of both minimizing the number of violations and the number of ACBs. As described in §3.2 the number of violations increases with $minf$ and finally equals that of the single-index approach. This shows the flexibility of our technique in reducing the number of barrels while violating the SAC property for far fewer files than the single-index approach.

4.3 Indexing and Search Datasets

The first data set, called T14m, is a publicly available cleaned subcollection [16] of TREC Enterprise track (TREC 14) [35]. TREC 14 is a newly formed track specifically on enterprise search and includes data from the World Wide Web Consortium (W3C) enterprise filesystems. The T14m dataset characteristics are shown in Table-1. It includes emails (*lists*), web pages (*www*), wiki web pages (*esw*) and people pages (*people*). This dataset does not include any access control information.

A significant portion of the efficacy of our approach depends on actual filesystem structure and access privileges in the enterprise. In order to measure this, we collected statistics from a real multiuser *nix enterprise installation, whose characteristics are shown in Table-2. We collected anonymized directory structure and access privileges information for 339,466 files arranged in 23,741 directories and replicated the structure in our test environment. The T14m data was used as content for the files (duplicating documents to fill all 339,466 files).

Scope	Docs	Size	Avg. Doc Size
lists	173,146	485 MB	2.9 KB
www	45,975	1001 MB	23.8 KB
esw	19,605	80 MB	4.2 KB
people	1,016	3 MB	3.1 KB
Total	239,742	1569 MB	6.9 KB

Table 1: T14m dataset: Cleaned TREC 14 subcollections [16]

Number of users	926		
Number of user groups	1203		
Number of files	339,466		
Number of dirs	23,741		
Max depth of dir structure	23		
Size of data	2.05 GB		
Number of barrels	2132		
Barrels per user	Max	Avg	Median
	25	6.31	4.26
	21^*	5.78^*	3.96^*

Table 2: Real enterprise dataset characteristics. Barrels per user statistics were also computed at a second enterprise (shown by *)

4.4 Indexing Experiments

Indexing is perhaps the most important component of our approach. It includes a pre-processing step that creates the user access hierarchy and the access control barrels followed by actual content indexing of the files contained in ACBs.

4.4.1 Pre-processing

As pre-processing performance is entirely dependent on the enterprise infrastructure (users/groups and directory structure), we use the real enterprise dataset for these experiments. Table-3 shows the evaluation of our implementation.

Task	Performance
Access hierarchy creation	38.7 sec
Barrel creation	263.1 sec
# Files `stat`'ed	202,446 (60%)
# Dirs `stat`'ed	14,059 (59%)

Table 3: Pre-processing performance for real enterprise dataset

Creating access hierarchy for 926 users and 1203 user groups took a total of 38.7 seconds, which is a very small fraction of the total indexing time. It took another 263.1 sec to traverse the filesystem and create all ACBs. One $\wedge$-credential needed to be added to the access hierarchy during barrel creation. Additionally only 60% of the filesystem tree needed to be traversed to create all barrels as in many cases, a higher level directory was mapped to a restrictive credential (e.g. only uid_{bob} can access) in which case its contents are automatically added to that barrel without deeper traversal.

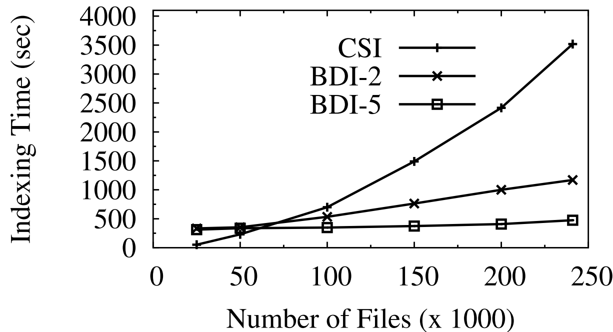

Figure 11: Indexing T14m dataset

Overall, these costs are only 10% of the distributed indexing approach and 6% of the centralized approach (Table-4).

4.4.2 Content Indexing

For indexing of documents, we used the *arrow* indexing and search component of the Bow Toolkit [24] developed at CMU. We modified the ranking algorithm of the toolkit to the distributed IR algorithm of [20]. For our experiments, we considered two architectures: (1) *Centralized Single Index (CSI)* - a centralized single index for the entire dataset analogous to the available enterprise search products[6], and (2) *Barrel-based Distributed Indexing (BDI)* - our barrels based distributed indexing approach. *BDI-m* denotes the case when m machines are used to index barrels in parallel.

Figure-11 shows the time to index different number of documents of the T14m dataset ranging from 25K to 240K. For BDI architectures, the documents were equally divided between the participating machines for indexing.
From the graph, CSI outperforms the BDI approaches when the number of documents is small. This occurs due to the pre-processing costs that the BDI approaches incur. However, as the number of documents increases, BDI approaches quickly outperform the CSI approach. The distribution of data into ACBs has allowed us to exploit available enterprise machines for faster indexing time (an 85% improvement for 240K files).

The results for the T14m dataset above are a little optimistic as it considers a uniform size and distribution of barrels. However, in reality there could be a few barrels that are significantly larger than the others and dominate the indexing times. To evaluate this, we performed indexing for our real enterprise dataset. As shown in Table-4, the barrel for the `all` node (files that can be read by all users) was significantly larger and took longer than all other barrels combined (and thus total time does not vary with number of machines). However, it was still 38% more efficient than the CSI approach. In general, distribution is most helpful when there are many such large barrels and we expect that to be true in an enterprise-scale environment.

Type	#Max-Docs	Time (s)	Savings
CSI	339,466	4640	–
BDI	189,546	2902	38%

Table 4: Indexing times for real enterprise directory structure. #Max-Docs is the number of documents in the largest barrel

[6] Recall that this architecture does not guarantee privacy preserving search

4.5 Searching

Recall that searching in our approach requires combining multiple barrels. However, given the small number of barrels per user, overheads should not drastically deteriorate query performance. Secondly, since our approach does not have to perform access control at runtime, there would be some savings in query runtime performance as compared to the CSI approach.

For the querying experiments we used 150 queries obtained from TREC 14 Email search. The queries had an average of 5.35 terms per query. The results for CSI and n-BDI (where n is the number of barrels combined) are reported in Table-5.

Type	Index size	Loading time	Avg. time / query
CSI	230 MB	2.5 s	131.12 ms
2-BDI	258 MB	3.37 s	112.89 ms
5-BDI	269 MB	5.68 s	130.68 ms
10-BDI	280 MB	6.90 s	149.90 ms

Table 5: Search performance comparison for TREC 14 *lists*. Loading time is the time to load all indices in memory

First notice that the BDI approaches have slightly larger indices. This is due to the fact that they have to store many words multiple times in different barrel vocabularies. Next, the time to load indices into memory also increases with the number of barrels as there are more file I/Os to gather the index data. However, this is only a one-time cost and once indices are cached, queries proceed normally. Finally, the average query time for BDI approaches is comparable to CSI with 2-BDI and 5-BDI even outperforming it by saving on the privileges check required at runtime in the CSI approach.

We also compared the ranking of the BDI approaches to CSI ranking. For this we evaluated the percentage of top-10 results of the CSI approach that occurred in top-100

of the distributed approach and their average ranks. As shown in Table-6, for our average case of 5 barrels per user, nearly 70% of top-10 results occurred in top-100 of the BDI approach with an average rank of 14. We believe that ranking can be further improved using more sophisticated distributed ranking measures.

Type	10-in-100	Avg. Rank
2-BDI	75%	13
5-BDI	68%	14
10-BDI	61%	15

Table 6: Ranking comparison for TREC 14 *lists*. 10-in-100 is the % age of CSI top-10 results in top-100 of x-BDI and avg-rank is the average rank of CSI top-10 results in x-BDI top-100

5 Design Extensions

The distributed nature of the ACB approach makes it suitable for search in a number of other enterprise scenarios. We discuss two extensions of the design for (a) seamless integration with desktop search, and (b) secure search in (untrusted) storage service provider setups.

5.1 Unified Desktop-Enterprise Search

A common challenge in enterprise environments is that data is typically spread into multiple islands lacking unified control. The primary contributing reason to this issue is the data on user laptops as it is not regularly backed up to file servers. Now, consider the problem of search in this environment from a user's perspective. A user *Jane* wants to quickly understand the nitty-gritties of enterprise search and remembers seeing related documents in the past, some on her laptop and some on her workstation in office. In the current data-islands scenario, she will have to issue two distinct queries (one on her laptop and another to the enterprise search system) and then manually rank the documents returned from the two systems in order to find the best ones.

In contrast, our approach can unify search over these data islands because of its intrinsic distributed nature. This is accomplished using a local orchestrator on the user's personal machine (see Figure-2) which has similar functionality to the global orchestrator though only at the scale of a single user. It interacts with the local indexer to index documents on local storage. For handling search, it treats the local index as another barrel and combines it with user accessible barrels (V_u^*) from the enterprise. This provides a seamless integration of local indices with accessible barrel indices in the enterprise.

5.2 Secure Search with Untrusted Search Service Providers

Another advantage of the ACB-based approach is that we can securely support search service provider (SSP) environments. As mentioned in §2, such environments have an external third-party that hosts search indices for data that is also typically outsourced to a storage service provider. The setup works as follows. Before an enterprise encrypts its data for archival at the storage service provider, it indexes the data using software or even a hardware appliance. The indices can then be sent to the same or potentially a different service provider. In such environments, it is crucial that no private information is leaked out to the SSP along with continuing to meet the intrinsic multiuser ACAS requirements. All existing single-index enterprise search products fail to support such environments since they perform access control at query runtime and an SSP environment lacks a trusted agent to do this. In contrast, our scheme can be extended to securely support these untrusted SSP-hosted indices. The description follows.

Recall that for a vocabulary of keywords, an index consists of multiple rows with a row for each keyword w. The row for w includes its TFIDF statistics from the files in barrel ACB_i and a list of files that contain w. Our primary task is to hide the file names and vocabulary from the SSP. We can do this using the following cryptography technique.

We associate each barrel ACB_i with a randomly generated barrel encryption key K_i. Only the users who have the required credentials to search the barrel ACB_i have the knowledge of K_i. This can be done by writing the key to a file stored in the enterprise global namespace using the same permission settings as the barrel itself. Only the users that can access ACB_i can get this key. Now, given a barrel encryption key K_i, we replace each keyword w in barrel ACB_i's vocabulary with a keyed hash value $KH_{K_i}(w)$, where KH denotes a pseudo-random function like HMAC-MD5 or HMAC-SHA1 [19]. Next, we present two approaches that differ in the level of trust placed at the SSP and the computation/communication overhead incurred at the client.

As a first approach (referred to as BDI-T) we leave the TFIDF statistics unencrypted and encrypt *each* file name *separately* with the barrel encryption key K_i. Now, given a search query with keyword w from a user u, the user's

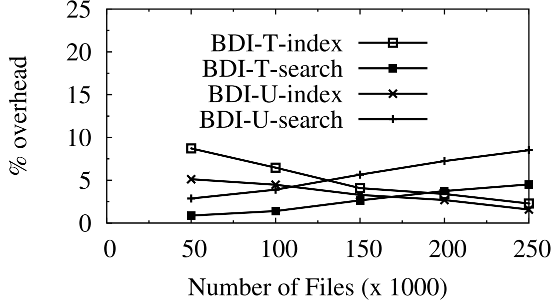

Figure 12: Computation Costs

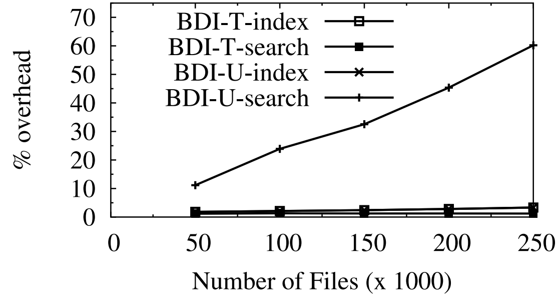

Figure 13: Communication Costs

client initiates a search for $KH_{K_i}(w)$ for all i such that ACB_i is searchable by the user u. Note that if a barrel j is not searchable by the user u, then the user u does not know the barrel encryption key K_j and thus cannot even guess the keyword $KH_{K_j}(w)$. The SSP performs a regular distributed IR search over user-accessible barrels for $KH_{K_i}(w)$ and returns a ranked list of encrypted file names which the user can decrypt with K_i. SSP can do this ranking as TFIDF statistics were left unencrypted. However, this approach is vulnerable to a frequency inference attack (on the frequency of keywords in the index). A frequency inference attack attempts to infer a keyword from its popularity, say the number of files that contain the keyword, information which is contained within the unhidden TFIDF statistics. Such frequency inference attacks can be thwarted using multiple SSPs. Please refer to §5.4 for one such technique.

A second approach, referred as BDI-U, is to hide the index statistics from the SSP as well. Similar to the first approach, we replace each keyword w in barrel ACB_i's vocabulary with a keyed hash value $KH_{K_i}(w)$, but instead of encrypting only the file names, we encrypt the entire row (including the TFIDF statistics) with the barrel encryption key K_i. When the SSP receives a search query with keyword $KH_{K_i}(w)$ from a user u, it returns the encrypted rows in the index corresponding to the keyword $KH_{K_i}(w)$. Now, the client has to perform some computation to decrypt and merge the results obtained from different barrels and present a final ranked list of files to the user. This approach preserves the privacy of the index statistics along with file names and vocabulary by incurring some computation and communication overhead at the client.

5.3 SSP Search Evaluation

Our SSP experiments compare the two approaches discussed above: BDI-T trusts the SSP with index statistics and only hides file names and words and BDI-U that hides everything.

Figure-12 and 13 show the computation and communication overhead incurred by the SSP approach over an approach wherein the SSP is completely trusted (nothing is hidden – used only for a baseline comparison).

The indexing cost for BDI-T is higher since we encrypt each filename separately, unlike BDI-U which encrypts the entire list of file names and index statistics for each keyword (encrypting file names separately requires each file name to be padded such that its length is an integer multiple of 16 bytes, a requirement of the encryption algorithms). On the other hand BDI-U incurs a higher cost for search. Computation cost is higher because BDI-U requires client side computation to decrypt and merge the results from multiple barrels. Communication cost is high because in BDI-U the SSP sends the entire list of file names for a keyword (along with the index statistics) to the client rather than sending a short-listed set of

ranked files. Note that the total number of files that match a keyword can be significantly larger than the result set, hence, the communication overhead in BDI-U is larger than BDI-T.

5.4 Thwarting Frequency Inference Attacks at SSPs

The first secure search approach for SSP environments (BDI-T) presented in §5.2 is vulnerable to frequency inference attacks. A frequency inference attack attempts to infer a keyword from its popularity, say the number of files that contain the keyword, information which is contained within the TFIDF statistics, left unencrypted. However, such frequency inference attacks can be thwarted using multiple SSPs assuming that the SSPs do not collude with one another. The primary reason a frequency inference attack works is that we expose the frequency of keywords to a SSP. However, one can largely obfuscate this information by partitioning the index across multiple SSPs. Indeed if all keywords appear equally popular to a SSP then the SSP cannot gain any additional information from a frequency inference attack. This technique is best explained with an example. Let us suppose that we have two keywords w_1 and w_2 with w_1 being twice as popular as w_2. Also assume that we have two SSPs, SSP_1 and SSP_2. We partition the index for keyword w_1 into two halves and store one half at each of the SSP. We store the index for the keyword w_2 entirely in SSP_1. From the perspective of SSP_1 both the keywords w_1 and w_2 are equally popular. Unless the SSPs collude with one another, the apparent popularity of all keywords for all SSPs is identical. In general one can extend this approach to multiple keywords and multiple SSPs as follows. Let f_1, f_2, $\cdots$, f_T denote the frequency (popularity) of T keywords. Let $minf = min(f_1,$ $f_2, \cdots, f_T)$. For a keyword w_i with popularity f_i, we calculate

$$NSSP_i = min\left(\frac{f_i}{minf}, NSSP\right)$$

where $NSSP$ is the total number of SSPs available. Now, divide the index for keyword w_i amongst $NSSP_i$ randomly chosen SSPs from the set of $NSSP$ SSPs.

To measure the effect of using multiple SSPs, we use entropy as the measure of an effectiveness of a frequency inference attack. Let f_1, f_2, $\cdots$, f_T denote the freqency of T keywords normalized such that $\sum_{i=1}^{T} f_i = 1$; then the entropy is computed as

$$S_{act} = -\sum_{i=1}^{T} f_i * \log f_i$$

Note that entropy denotes the degree of randomness in the system; hence, larger the entropy more the randomness and harder it is for the SSP to perform a frequency inference attack. If the popularity of all keywords are equal, that is, $f_i = \frac{1}{T}$ for all i, then the entropy is maximum $S_{max} = \log T$. Let $NSSP_i$ denote the number of SSPs in which keyword w_i is stored. Then the apparent popularity of keyword w_i as measured by the SSPs is $f_i' = \frac{f_i}{NSSP_i}$, where $NSSP_i$ is determined as described in §5.2. One can measure the apparent entropy of T keywords on normalized apparent frequencies $\sum_{i=1}^{T} f_i' = 1$; then the apparent entropy is computed as $S_{app} = -\sum_{i=1}^{T} f_i' * \log f_i'$.

Figure 14 shows the apparent entropy as the number of SSPs increases. Observe that even with five SSPs the apparent entropy can be almost close to the maximum entropy S_{max}.

Figure 15 shows the apparent entropy as the collusion amongst SSPs increases with $NSSP = 5$. As more SSPs collude the apparent frequency of a keyword stored by the colluding SSPs gets closer to its actual frequency. Indeed if all the SSPs collude with one another, the apparent entropy S_{app} = actual entropy S_{act}.

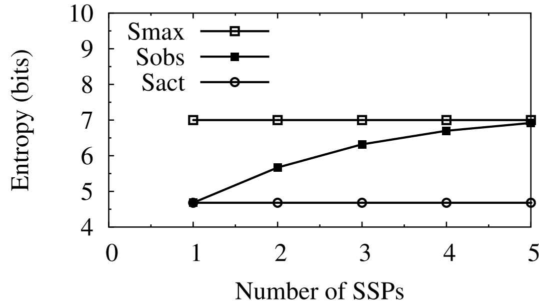

Figure 14: Entropy, no collusion

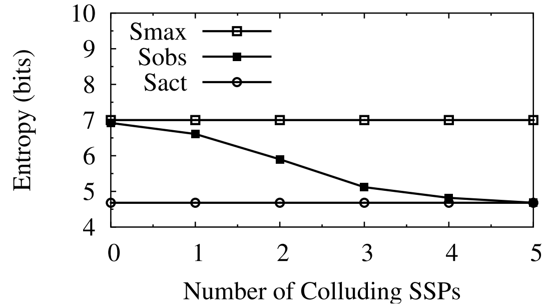

Figure 15: Entropy with collusion

	Desktop Search	**Enterprise Search**	**Wumpus** [3]	**Our Approach**
Access Control Integration	Indexing	Query Runtime	Query Runtime	Indexing
ACAS Requirement	Satisfied	Not satisfied	Satisfied	Satisfied
Service Provider Support	No	No	No	Yes
Search Consolidation	No	No	No	Yes
Architecture	Centralized	Centralized	Centralized	Distributed & parallel
Overheads	High space & update costs	Runtime privileges check	Runtime query transformation & privileges check	Barrels processing & runtime results merging

Figure 16: Comparison of search approaches

6 Related Work

We have already described existing work in enterprise search and how our approach tackles the problem differently. Table-7 summarizes this discussion and compares all approaches on various attributes. In the rest of this section, we will cover related work in distributed IR, private search, and keyword based search over encrypted data, similar to our SSP environment.

The growth of geographically separated file collections have made distributed information retrieval a necessity. Kretser et al [20] compare different approaches of distributed retrieval. They conclude that distributed IR systems can be as fast and effective (quality-wise) as the monolithic systems using four distributed collections. Xu and Callan [36] and Powell et al [26] suggest that the effectiveness of distributed information retrieval systems can drop by up to 30% when the number of collections exceeds 100. However, in our approach the number of collections (number of barrels per user) was on an average about 6 and a maximum of 25 (see Table-2). Our algorithms to construct a minimal set of ACBs (see §3.1.2) ensure that the number of barrels per user is small and thus one can hope to obtain high quality results as predicted by [20]. In addition, one could also deploy query expansion and careful source selection [36, 26] to enhance the effectiveness of our approach.

Bawa et al [1] present techniques for constructing a privacy preserving index on documents in a multi-organizational setting. Their goal is to construct a centralized index that can be made public without giving out any private information. Similar to other enterprise search techniques they apply access control at query runtime and incur higher overheads than our proposal. Our approach focuses on integrating access control with search in a single enterprise setting and is more efficient.

Private information retrieval (PIR) was first introduced as a problem by Chor et al [5] – a user wishes to retrieve the ith bit in a database without revealing any information about i. PIR schemes often require multiple non-colluding servers, operate in multiple rounds, are resource-intensive and do not support keyword search. Hence, several authors have focused on efficient solutions and their security guarantees. Another direction of work has focused on running queries over encrypted data at an untrusted server [15, 32, 4]. These schemes require the user to know a secret key with which the searchable content of the document is encrypted. They ensure that only the frequency profile of the queried keywords is revealed to the search service provider (similar to our BDI-T approach). However, these approaches do not consider a multiuser enterprise setting where in addition to keeping the data private from the SSP, one needs to enforce access control rules on the users. Our approach cleverly partitions the search problem into two parts: an access control problem that is handled by our barrels-based secure indices, and a privacy problem if such indices are hosted by a third-party search service provider. Indeed, we can leverage any approach [32, 2] that provide privacy preserving search over an untrusted service provider hosted index.

Google Search-Across-Computers [11] provides an insecure and non-privacy preserving solution to store files and indices at a remote service provider. There are some commercial storage service providers that allow files to be encrypted and stored securely at the data center [8, 25]. Several researchers have also focused on building cryptographic filesystems [18, 10, 21] that store files in an encrypted manner and use key management protocols to ensure that a user has a file's key if and only if the user is permitted to access that file. We can leverage any such approach that stores encrypted files in combination with a privacy preserving search capability.

7 Conclusions and Future Work

In this work, we presented an efficient and secure approach to enterprise search. We demonstrated the inadequacy of existing solutions at ensuring privacy preserving search and developed distributed techniques that elegantly capture access control semantics of enterprise repositories, using novel *access control barrel (ACB)* and *user access hierarchy* concepts. The distributed and parallel nature of our solution helps improve indexing efficiency and reduces resource requirements for search servers. It also seamlessly integrates single-user desktop search with enterprise search and unlike all existing approaches, can provide security in external search service provider environments. Our experimental evaluation on synthetic and real datasets shows improved indexing efficiency and minimal overheads for ACB processing.

As part of future work, we intend to integrate data-specific indexing and search mechanisms with our approach. This becomes necessary when users want to search (and rank) on higher level metadata concepts along with full content search.

References

[1] M. Bawa, R. Bayardo, and R. Agarwal. Privacy-preserving indexing of documents on the network. In *VLDB*, 2003.

[2] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Eurocrypt*, 2004.

[3] S. Büttcher and C. Clarke. A security model for full-text file system search in multi-user environments. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2005.

[4] Y.C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, 2005.

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, 1995.

[6] Google Desktop. *http://desktop.google.com*.

[7] Yahoo Desktop. *http://desktop.yahoo.com*.

[8] Arsenal Digital. *http://www.arsenaldigital.com*.

[9] Windows Desktop Search for Enterprise. *http://www.microsoft.com/windows/desktopsearch*.

[10] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: securing remote untrusted storage. In *NDSS*, 2003.

[11] Google. *http://desktop.google.com/features.html#searchremote*.

[12] Butler Group. Unlocking value from text-based information. *Review Journal Article*, March 2003.

[13] Gartner Group. *http://www.gartner.com*.

[14] A. Grunbacher and A. Nuremberg. POSIX Access Control Lists on Linux. *http://www.suse.de/%7Eagruen/acl/linux-acls/online*.

[15] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD*, 2002.

[16] Daqing He. Cleaned W3C Subcollections. *http://www.sis.pitt.edu/%7Edaqing/w3c-cleaned.html*.

[17] IBM WebSphere Information Integrator. *http://www-306.ibm.com/software/data/integration/db2ii*.

[18] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2003.

[19] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. *http://www.faqs.org/rfcs/rfc2104.html*.

[20] O. Kretser, A. Moffat, T. Shimmin, and J. Zobel. Methodologies for distributed information retrieval. In *ICDCS*, 1998.

[21] J. Li, M. Krohn, and D. Mazieres. Secure untrusted data repository SUNDR. In *OSDI*, 2004.

[22] Linux Manual Pages. *man command-name*.

[23] R. Love and J. McCutchan. inotify linux file system monitor.

[24] A. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. *http://www.cs.cmu.edu/%7Emccallum/bow*.

[25] Iron mountain. *http://www.ironmountain.com*.

[26] A. Powell, J.French, J.Callan, M.Connell, and C.Viles. The impact of database selection on distributed searching. *SIGIR*, 2000.

[27] D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7), 1974.

[28] S. Robertson, S. Walker, and M. Beaulieu. Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive. In *TREC*, 1998.

[29] Coveo Enterprise Search. *http://www.coveo.com*.

[30] Google Enterprise Search. *http://www.google.com/enterprise*.

[31] Amazon Simple Storage Service. *http://aws.amazon.com/s3*.

[32] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches over encrypted data. In *IEEE S & P Symposium*,

2000.

[33] Wikipedia Tf idf. *http://en.wikipedia.org/wiki/Tf-idf*.

[34] MSN Toolbar. *http://toolbar.msn.com*.

[35] TREC Enterprise Track. *http://www.ins.cwi.nl/projects/trec-ent*.

[36] J. Xu and J. Callan. Effective retrieval with distributed collections. In *SIGIR*, 1998.