Zerber: r-Confidential Indexing for Distributed Documents

Sergej Zerr[1], Elena Demidova[1], Daniel Olmedilla[1], Wolfgang Nejdl[1],
Marianne Winslett[2] and Soumyadeb Mitra[2]

[1]L3S Research Center
University of Hannover
Hannover, Germany
{zerr,demidova,olmedilla,nejdl}@L3S.de

[2]Department of Computer Science
University of Illinois at
Urbana-Champaign, USA
{winslett,mitra1}@uiuc.edu

ABSTRACT

To carry out work assignments, small groups distributed within a larger enterprise often need to share documents among themselves while shielding those documents from others' eyes. In this situation, users need an indexing facility that can quickly locate relevant documents that they are allowed to access, without (1) leaking information about the remaining documents, (2) imposing a large management burden as users, groups, and documents evolve, or (3) requiring users to agree on a central completely trusted authority. To address this problem, we propose the concept of *r*-confidentiality, which captures the degree of information leakage from an index about the terms contained in inaccessible documents. Then we propose the *r*-confidential ZERBER indexing facility for sensitive documents, which uses secret splitting and term merging to provide tunable limits on information leakage, even under statistical attacks; requires only limited trust in a central indexing authority; and is extremely easy to use and administer. Experiments with real-world data show that ZERBER offers excellent performance for index insertions and lookups while requiring only a modest amount of storage space and network bandwidth.

1. INTRODUCTION

The number of sensitive documents shared over enterprise intranets is growing rapidly. Sharing of access-controlled documents has traditionally been accomplished through point to point sharing techniques such as email, or through centralized repositories such as shared file directories on intranet servers, access-controlled web pages, wikis, and even source code control systems. Each of these approaches has drawbacks with respect to security, management overhead, and/or ease of use. Point-to-point sharing techniques do not scale up well: emailing new versions is awkward when more than a handful of collaborators are reading and updating a document. Centralized techniques address these problems but place too much trust in the administrator controlling the central server; in a large enterprise, conflicting interests make it unlikely that everyone will be willing to give their sensitive documents to a particular superuser. Further, if the central server is compromised, all the documents stored there are also compromised.

Another option is for each user to publish their own documents on access-controlled web pages on their own web server, whose administrator they presumably trust. Alternatively, a document can be placed on a public server after encrypting it with a key known to all members. In both cases, just securing the documents is insufficient, as they need to be indexed to support efficient search and retrieval. *Inverted indexes* are the standard choice for keyword (full-text) search of documents. An inverted index is a sequence of *posting lists*, each of which contains the IDs of all documents containing one particular term. Figure 1 shows an inverted index with three posting lists and nine *posting list elements* (*elements* for short). The index contains sufficient information to reconstruct the set of words in a sensitive document, so it needs to be protected against unauthorized access. It is unlikely that all project groups can agree on a single trusted central authority to enforce access control on index entries; even if they can, centralized indexes are attractive targets for attack and will need additional protection. For example, even if the exact content of the elements is obscured, the lengths of the posting lists can tell an industrial spy which compounds are used in the development of a new chemical process [11]. Protecting an inverted index is a challenging problem when there is no single trusted central authority to enforce access control on posting list elements - as is the case in the project group scenarios we target.

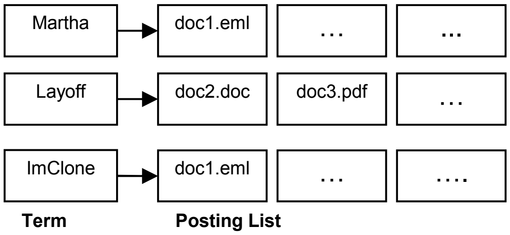


Figure 1: Inverted Index with 9 Elements

One possible solution is for each document owner to keep an inverted index over the documents it owns locally. Then a user's query for the term "ImClone" can be broadcast to all document owners, and the resulting answers can be collected by the user and, if desired, ranked. Each document owner retains absolute control over her index as well as her documents, and can enforce access control on each index lookup. However, this shotgun approach to querying is relatively slow, and wastes network bandwidth and computing power, since most document owners will not have posting list elements matching most queries.

EDBT'08, March 25-30, 2008, Nantes, France.

Another potential solution is for the document owner to encrypt any sensitive information in each posting list element, and insert the element in a global inverted index [12, 31]. Encryption schemes usually have complex key management schemes that make the system hard to use and administer, and can compromise its effectiveness. Further, in practice, each element includes a *term frequency*, that is, a count of the number of times that term appears in that document, divided by the document's length. If the index provides ranked query answers, term frequencies must not be encrypted, as they are a major factor in the relevance score computed by the ranking algorithm. But from plain-text term frequencies, one can reverse-engineer the terms themselves [11].

To address these problems, we propose *r-confidentiality* as a measure of the degree of information that can leak from an index about inaccessible documents, given an adversary's background knowledge of the corpus or language statistics. Then we propose ZERBER[1], an *r*-confidential global inverted index for sensitive documents. ZERBER relies on a centralized set of *largely untrusted* index servers that hold posting list elements encrypted with a ***k*** out of ***n*** secret sharing scheme [29], which provides complete resistance against inappropriate information disclosure regarding pre-existing documents even if ***k***-1 index servers are compromised. To provide tunable resistance to statistical attacks, ZERBER employs a novel term merging scheme that has minimal impact on index lookup costs. ZERBER guarantees freshness of shared documents at low cost, makes economical use of network bandwidth, requires no key management, and answers most of the queries almost as fast as an ordinary inverted index.

The rest of the paper is organized as follows: Section 2 describes the collaboration scenarios we want to support and presents the characteristics of the ideal indexing scheme for these environments. Section 3 describes related work, and Section 4 introduces the security model. Sections 5 and 6 describe ZERBER, and Section 7 evaluates the performance of ZERBER with real-world data and queries. Section 8 concludes the paper.

2. THE IDEAL SOLUTION

We target the problem of supporting efficient keyword search for sensitive unstructured documents shared within collaboration groups. These groups reside within a large enterprise, or may even span multiple enterprises, so there is no central authority that all members trust with the content of their documents. Members are, however, willing to trust the enterprise's authentication facilities. Such environments are common in large companies, large universities, and large government groups.

The content of the shared documents evolves over time, as does the group membership. Groups come and go relatively quickly, as projects start and finish. As each person can only accomplish a certain amount of work, in practice she will belong to a limited number of collaboration groups.

Given a keyword query, the *ideal* indexing scheme's answer will be identical to that of a trusted centralized ordinary inverted index that incorporates an access control list check on the ranked document list just before returning it to the user. The ideal indexing scheme will answer queries and handle updates as fast as an ordinary inverted index, and with no greater network bandwidth or storage usage. Changes in group membership will be immediately reflected in the query answers of the ideal indexing scheme. The ideal indexing scheme will impose no management burden on group members, beyond the requirement that the group coordinator maintain a list of the identities of the people in the group, the group members know how to authenticate to those identities, and the group members have trusted desktop or local web servers where they can upload their sensitive content into appropriate access-controlled directories and have a daemon automatically ensure that the corresponding index updates are carried out quickly. If the server or servers containing the ideal scheme's index are compromised, no information about the content of the documents should be revealed. No user or superuser on a non-trusted machine should be able to obtain any information about the content of sensitive indexed documents that they are not authorized to access.

The ideal indexing scheme will be unattainable in practice, but we can quantify the degree to which any proposed approach falls short. We can also provide schemes that provide tunable tradeoffs between the confidentiality guarantees provided by the index and its query and update processing overhead.

3. RELATED WORK

Index security has been addressed by many techniques designed for the outsourcing threat model, where the goal is to secure the index from tampering by the untrusted storage server. These techniques store the index in plain text, and so do not address confidentiality. For example, Merkle hash trees let one verify the authenticity of any tree node entry by trusting the signed hash value stored at the root node. Authenticated dictionaries [9, 18] support secure lookup operations for dictionary data structures.

Encryption is a standard technique for storing data confidentially [8, 18, 23]. [5, 27] provide a framework for policy-based protection of XML data by encryption. Other techniques include suppressing and/or generalizing released data into less specific forms, so that they no longer uniquely represent individuals [16, 22]; k-anonymity is one popular form of generalization (e.g., [4, 25, 26]). Unfortunately, it is not possible to directly apply these techniques to secure an inverted index. Even if posting list entries are encrypted, they can leak critical statistical data.

The research most relevant to our problem is μ-Serv, a system developed at IBM to index distributed access-controlled documents [3]. μ-Serv has a centralized index based on a Bloom filter; it responds to a keyword search by returning a list of *sites* that have at least *x*% probability of having documents containing one of the query keywords, where *x* is a preset parameter. Users then repeat their query at each suggested site. The lack of precision in results from the central index represents a tradeoff between search efficiency and confidentiality preservation. This approach lengthens the querying process and wastes cycles at sites that do not contain query-relevant entries. For example, if *x* = 5%, the user must query 20 times as many sites to get the relevant results. Further, μ-Serv does not support centralized ranking; the user must get ranked search results from individual sites and combine them. ZERBER'S centralized indexes direct users to *documents* that definitely satisfy the user's query, and provides a level of confidentiality equivalent to μ-Serv at much lower query cost: because ZERBER provides exact search results, users can

[1] Zerber (Цербер): a mythical three-headed watchdog.

rank their search results locally and visit only the top-K document server sites to obtain document snippets.

While many other researchers have addressed aspects of data confidentiality, none of their schemes are intended for an environment with many dynamic collaboration groups. For example, researchers have suggested ways to search encrypted text or tables stored on a remote untrusted server (e.g., [10, 12, 19, 31]). In a situation with many collaboration groups with dynamically changing membership, these approaches are not easy to use or manage. Document owners and/or project group managers must generate and distribute keying material for all group members, so that they can encrypt keywords and decrypt search results. If a key is lost, stolen, or even published, the index entries encrypted with it are compromised. When a key is compromised or a member leaves a group, the key must be revoked and all the content associated with that key must be re-encrypted and re-indexed. Modern group key management schemes, such as logical key trees [13] and broadcast encryption, reduce the costs associated with giving keys to members, but still require content re-encryption. Some approaches also require that the entire index for a particular collection of documents be regenerated by the collection owner every time an entry is added to or deleted from the index. ZERBER does not use keys.

Distributed indexes such as Distributed Hash Tables (DHTs) are popular for P2P networks [2]. ZERBER distributes complete instances of an encrypted index to multiple servers for security reasons, while in DHTs each peer typically stores only a fraction of the index. The extension of *r*-confidential indexing to a DHT-based infrastructure is an interesting area for future research.

4. THREAT MODEL

To give a sense of the set of potential dangers, consider the following three goals of a potential attack on an index.

Reconstruct the exact content or the term frequencies of an inaccessible document. Exact reconstruction is clearly undesirable. The *term frequency* distribution, i.e., the number of occurrences of each term in the document, is sufficient to characterize the subject matter of a lengthy document, and the likely content of a short email.

Determine the aggregate term/document frequencies for the set of inaccessible *documents at a participating site.* The *document frequency* is the number of documents at a site that contain a particular term. In an ordinary inverted index, the length of a term's posting list is its (global) document frequency. These frequency distributions will often suffice to characterize the nature of a project that the site's owner is participating in. For example, one might be able to tell that the CEO is considering a buyout offer from a particular suitor, or tell what solution approach the smartest project group in a course has adopted. Document frequencies can also tell an industrial spy which compounds are used in the development of a new chemical process [11].

Determine whether a particular term appears in a particular inaccessible document at a particular site, or at any indexed site. For example, curious employees may want to know whether Mildred Hesselhofer of IBM is a finalist for the CEO job at HP. Rare terms like "Hesselhofer" especially need this protection.

The conflicting interests of enterprise project groups make it hard for them to trust a centralized corporate index server. Even with a distributed index, an attacker Alice will already have some background knowledge about the possible contents of a document collection. We will restrict Alice's ability to increase this knowledge, even if she takes over an index server and can examine the contents of that server. More formally, we will **bound the ability of an adversary to make probabilistic claims about the contents of a document collection**. From her background knowledge ***B*** and the parts of the index structure ***I*** that she can access, Alice will know a priori that a term ***t*** is contained in document ***d*** with a probability *P*(***t*** is in ***d***). For example, for a set of emails, ***B*** should include *P*("Subject" is in ***d***) = 1, for all ***d*** and ***I***. We cannot control the probability estimate ***P(X|B)*** about fact ***X*** that Alice can make based on ***B***, but we can limit her ability to refine that estimate when she computes ***P(X|I,B)***. In the remainder of the paper, we will consider only facts ***X*** of the form "term **t** is in document ***d***" and "term **t** is not in document ***d***".

Definition 1. An indexing scheme is *r-confidential* iff

$$\frac{P(X \mid B, I)}{P(X \mid B)} \leq r. \qquad (1)$$

Here, ***r*** is the factor of maximal probability amplification for term ***t*** in ***d*** given ***I***. In other words, *P*(***t*** is/is not in ***d*** | ***B***) ≤ α implies *P*(***t*** is/is not in ***d*** | ***B, I***) ≤ ***r*** · α. The indexing scheme offers maximal protection when ***P(X|B) = P(X|I,B)***, i.e., ***I*** does not provide any additional knowledge about ***X***.

r-Confidentiality focuses on document content confidentiality. In addition, secure communication channels such as https should be used to provide confidentially for the content of queries and updates. If no one should be able to tell that a particular user sent a request to an index server, we recommend the use of MIX networks and other standard techniques from network security that foil traffic analysis attacks. An attacker could compromise a non-index site so that it, for example, gives query results or actual documents to unauthorized parties. Such attacks should be guarded against using standard techniques, and we do not consider them further, as our goal is to secure the *index*.

5. ZERBER DESIGN

ZERBER is an *r*-confidential inverted index that incorporates secret splitting and term merging. After a quick overview of the reasons for these choices, we discuss each feature in detail.

As mentioned earlier, encryption is the classic way to protect information on an untrusted server, but has significant drawbacks with respect to key compromise, revocation, and manageability. We avoid the need to distribute *any* keys to group members by applying a ***k*** out of ***n*** secret sharing scheme [29] to posting list entries. Each entry is divided into ***n*** shares, such that any ***k*** of the shares can be used to reconstruct the entry, and one share is given to each of the ***n*** index servers. Each non-compromised index server authenticates the user and ensures she belongs to the right group, before giving her an element in response to her query. Even if an index is designed and implemented perfectly, the platform it resides upon is still vulnerable to compromise; one can bribe the sysadmin, measure radiation, take over root, etc. If one server is compromised, its secret shares do not provide enough information to decrypt any element-***k*** secret shares from ***k*** different servers are required. Thus, at least ***k*** servers must authorize a user before she can decrypt any posting list element.

Each index server should be owned and managed by a different part of the enterprise. With this setup, no person in the enterprise can decrypt an index entry she is not authorized to see, unless she can find colluders from ***k***-1 other factions, or compromise that many index servers. In other words, at least ***k*** index servers must be compromised to decrypt an index entry that should remain inaccessible. This minimizes the amount of trust that project groups must place in the ***n*** "centralized" index servers.

The ***n*** server boxes can provide extra resilience to attack by running *only* the index and no other services; providing only a narrow interface to the outside world (i.e., only insert, delete, and look up posting list elements); and using different hardware, operating systems, and systems software versions. Without this diversity, a successful attack on the underlying platform of one server may succeed against all ***n*** servers.

5.1 Encryption of Posting Elements

Each posting list element in ZERBER is encrypted using Shamir's ***k*** out of ***n*** secret sharing scheme [29]. According to this scheme, all the operations described later in this section are carried out in the finite field Z_p. The secret splitting algorithm starts by choosing a large prime number p, such that any element (secret) to be shared is in Z_p. In addition, each server i is assigned a unique random value x_i in Z_p. We call this the x-coordinate of the server. These numbers p and x_i are made public, so all users know them.

To index a document, its owner first parses the document and computes its elements. For each element $\boldsymbol{a_0}$ (viewed as an integer), she generates a pseudo-random polynomial ***f*** of degree ***k***-1 of the form $f(x) = a_{k-1}x^{k-1} + ... + a_0 \bmod p$, with coefficients a_i (except a_o) randomly picked from field Z_p. The number of random bits required for each coefficient is the number of bits in a_o. The secret share given to the ***i***th server is $f(x_i)$. ***k*** such shares are enough to reconstruct the polynomial. The encryption procedure is outlined in Algorithm 1a.

```
Encrypt a posting element (run by the document owner)

Input: posting element a0, prime p
1. Select random coefficients $a_1,...,a_{k-1} \in Z_p$
2. Create a polynomial of degree $k-1$, e.g.,
       $f(x) = a_{k-1}x^{k-1} + ... + a_0 \bmod p$, where $a_0$ is the secret
3. Take the $x_i$-coordinate of each of the n servers, $x_i \in Z_p$
4. For each $x_i$ compute $y_i = f(x_i)$
5. Send the resulting $y_i$-coordinate to server i
```

Algorithm 1a: Compute *k*-out-of-*n* Secret Shares

The owner repeats this process to split all the elements for the document across the ***n*** servers. She also tells the servers who can access this document. Algorithm 1a has a computational complexity of $O(\boldsymbol{nN})$, where ***n*** is the number of servers and ***N*** is the number of distinct terms in the document. For example, creation of the secret shares for one server for a document with 5,000 distinct terms requires only 33 msec on the platform described in Section 7.3.

To decrypt an element, a user must obtain ***k*** of its secret shares and determine the coefficients of the polynomial ***f*** by solving a system of ***k*** linear equations. This is presented in Algorithm 1b.

```
Decrypt a Posting Element (performed by the querying user)

1. Gather at least k shares of the element from any of the n servers
2. Recover $a_0$ by solving the following system of k linear equations:
     $y_i = a_{k-1}x_i^{k-1} + ... + a_0 \bmod p$ , $i \in [1, k]$
```

Algorithm 1b: Compute Secret from *k* Shares

The ***k*** linear equations can be solved in $O(\boldsymbol{k}^3)$ time with Gaussian elimination methods, which is affordable given that ***k*** is quite small in practice. For instance, on the platform described in Section 7.3, we can decrypt 700 elements in 1 msec on average.

Shamir's secret sharing scheme allows dynamic extension of the number ***n*** of servers without recalculating the existing secret shares, by just selecting additional points on the polynomial curve. Moreover, if an adversary learns some of the shares, proactive sharing techniques can be used to prevent the adversary from getting ***k*** shares [21]. With this technique, the shares are updated so that those she already knows become useless.

Just splitting the elements across ***n*** servers does not secure the documents against all the threats discussed above. For example, the number of elements in a term's posting list is its document frequency. Exploiting this, the adversary can learn about the presence (and even the exact number) of documents containing sensitive terms. The server cannot obfuscate the mapping of terms to posting lists on its own, as an adversary who takes over a server will be able to learn the mapping. Document owners could encrypt all terms before building posting elements for them; but then the other group members must know the encryption/decryption function, and we have already said that we want to free users from key management and from re-encrypting documents as groups shrink. Fortunately, there is a better way to increase security.

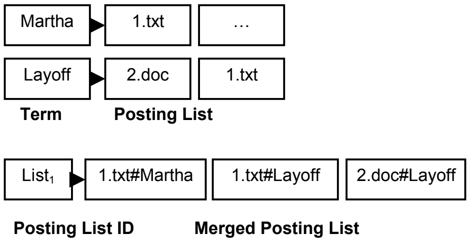


Figure 2: Merged and Unmerged *Unencrypted* Posting Lists

5.2 ZERBER Merged Posting Lists

To prevent the adversary on a compromised index server from learning a term ***t***'s document frequencies, we combine ***t***'s posting list with the posting lists for several other terms, as shown in Figure 2 for an unencrypted posting list (term frequencies are not shown). An additional encoding is stored with each element to identify the term for that element. This encoding is encrypted along with the document ID and term frequency tf, using the ***k*** out of ***n*** secret sharing scheme described in Section 5.1. An unencrypted element hence contains three fields:

secret = [document_ID, term_ID, tf].

With this scheme, the adversary on a compromised index server only sees the combined posting list length of the merged terms and cannot determine an individual term's document frequencies. Further, the elements for a document are encrypted separately, so the adversary cannot determine which elements correspond to the same document. We formally analyze this below.

Suppose the posting elements of terms $\boldsymbol{t_1}, \ldots, \boldsymbol{t_n}$ have been merged into one list. Upon examining an element in the merged list, an adversary can deduce only the following:

- Some document contains one of the terms $\boldsymbol{t_1}, \ldots, \boldsymbol{t_n}$.
- That same document can be read by users $\boldsymbol{u_1}, \ldots, \boldsymbol{u_m}$.

Although the adversary cannot determine the exact term in an element, she can make probabilistic claims about it using her background knowledge, e.g., general language statistics.

The probability p_t of occurrence of a term $\boldsymbol{t}$ in the document corpus $\boldsymbol{D}$ is represented by its normalized document frequency:

$$p_t = n_d(t) \Big/ \sum_{t_i \in D} n_d(t_i), \tag{2}$$

where $n_{d(t)}$ is the number of documents in D containing term t.

The probability of the posting element containing a particular term $\boldsymbol{t_u}$, given that it is one of the terms in the set $\boldsymbol{S} = \{\boldsymbol{t_1}, \ldots, \boldsymbol{t_n}\}$, is the ratio of the normalized frequency of that term to the sum of the normalized frequencies of all the terms in the set $\boldsymbol{S}$:

$$p_{t_u} \Big/ \sum_{t_i \in S} p_{t_i}. \tag{3}$$

For a scheme to be $\boldsymbol{r}$-confidential, this probability should not exceed $\boldsymbol{r}$ times the probability of $\boldsymbol{t_u}$ occurring in a document according to the adversary's background knowledge:

$$\forall t_u \in S : \left(p_{t_u} \Big/ \sum_{t_i \in S} p_{t_i} \right) \le r \cdot p_{t_u}. \tag{4}$$

Intuitively, $\boldsymbol{r}$ measures the additional information the adversary can extract from the index, beyond her background knowledge. Thus, for the merging scheme to be $\boldsymbol{r}$-confidential, we must have:

$$\sum_{t_i \in S} p_{t_i} \ge 1/r \tag{5}$$

The $\boldsymbol{r}$-confidentiality definition also encompasses the ability of the adversary to make claims about the absence of a term $\boldsymbol{t_u}$ in a document. Given an element for a merged set of terms $\boldsymbol{S}$, the probability that it is *not* an element for term $t_u \in S$ is $1 - (p_{t_u} \Big/ \sum_{t_i \in S} p_{t_i})$. This probability is smaller than the original probability $(1 - p_{t_u})$ in the document corpus.

Hence our objective is to find a merging strategy that satisfies the formula (5) and minimizes the expected query cost. In Section 6 we present several merging strategies that trade off between the degree of confidentiality $\boldsymbol{r}$ and the index size. Note that in the remainder of the paper, *all posting lists are merged*.

5.3 Access Control in ZERBER

To answer user queries, each ZERBER index server enforces access control on its posting elements. Upon query, the server authenticates the user and determines the groups the user belongs to. For this purpose, each index server records which users belong to each group, and which posting elements are accessible to each group. We do not consider this information sensitive here, given that we can bound the ability of the adversary to extract document contents from the inverted index. (If this information is sensitive, it can be blinded and/or stored on a separate server.)

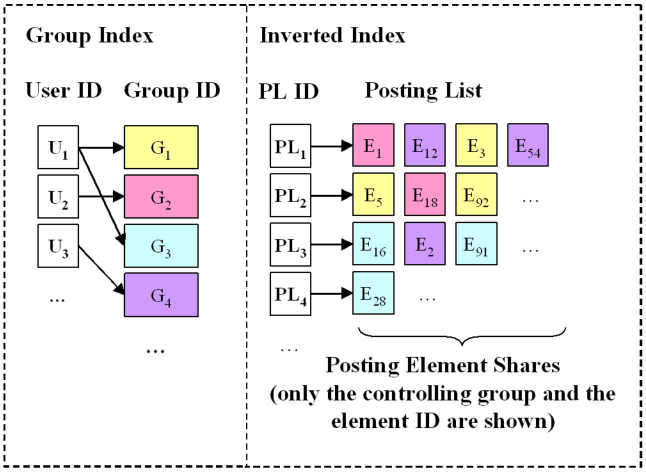


Figure 3: ZERBER Index Server

The structure of a ZERBER index server is shown in Figure 3. The architecture supports dynamic changes in group membership. To add or remove a user from a group, only the table containing the user-group metadata needs to be updated. (The metadata that controls who can update a group is not shown in Figure 3, and that process is outside the scope of this paper.).

5.4 Using ZERBER

In this section, we describe indexing and querying in ZERBER.

5.4.1 Indexing a Document

To index a document, its owner extracts the document's terms, builds their elements, encrypts them with Algorithm 1a, gives each one an ID that will be globally unique within its posting list, and sends an element share to each of the $\boldsymbol{n}$ servers, along with the IDs of the merged posting list that the new element belongs to, the document's group, and the element ID. The index server authenticates the user, checks his group membership and accepts the update if appropriate. The element IDs help an index recover after failure, and tell users which shares to merge together.

Index updates in ZERBER can be performed in batches that insert or delete posting elements for multiple documents. Batching can reduce index freshness[2], but also reduces the average network and disk overhead per update (each append to a posting list incurs a random I/O). If Alice has compromised an index server, then batching also reduces the information she gets by watching updates. For example, if Bob inserts into the posting lists for {Martha, P} and {Ralph, Q}, Alice can guess that "Martha" and "Ralph" occur in the same document (if P and Q are unlikely to occur with the other terms). Inserting elements from several documents in one batch makes it hard for Alice to guess which terms co-occur. Bob can also pool his updates with other people's, or send his through a MIX network, to give himself anonymity and improve index freshness. If the user trusts that no

[2] Delaying index updates for some of the terms in an updated document does not affect *document* freshness: only the most recent copy of the document on a site will ever be retrieved.

index servers are compromised, then the indexes can be updated whenever a shared document changes, rather than in batches.

Batch size, frequency, and other batch parameters can be tuned by each document owner to trade off security and index freshness, based on the elapsed time since the last batch, term frequency, vocabulary changes, or query volume [1]. ZERBER runs a client program at the document owner that tracks local changes and performs only the necessary updates at the central indexes.

5.4.2 Processing Queries

To execute a keyword query, the user first authenticates herself to ***k*** or more index servers. The index servers rely on an enterprise-wide authentication service, such as one normally finds in today's large enterprises; Kerberos or any other approach to authentication in distributed systems can be adopted here. The index servers determine her groups by consulting the group table. This can be done in $O(\boldsymbol{N})$ time, where $\boldsymbol{N}$ is the number of groups. The user tells the servers which posting lists she wants (she does not divulge which *terms* she is querying), and each server returns a share of each posting element that she is allowed to access:

PL_ID, [{g_id$_1$, $\boldsymbol{e}$(doc$_1$, term$_1$, tf$_1$)},…,{g_id$_j$, $\boldsymbol{e}$(doc$_j$, term$_j$, tf$_j$)}],

where PL_ID is a posting list ID; $\boldsymbol{e}$ returns a secret share; and g_id$_i$, doc$_i$, term$_i$, and tf$_i$ are the global element ID, document ID, term ID, and term frequency of the i^{th} element, respectively. The document ID must identify both the machine on which the document is hosted and the document within that machine.

Based on the global posting element ID, the client determines the corresponding element shares from the different servers, decrypts the element with Algorithm 1b, then filters out false positives, i.e., elements for terms not queried. The client then ranks the results using any modern document ranking technique [30].

ZERBER uses client-side ranking with personalized collection statistics obtained from the set of all documents accessible to the user. We use a modification of Fagin's Threshold Algorithm [14] that lets one obtain the top-K ranked results in time $O(PLLength^{\frac{QT-1}{PLLength}} \cdot K^{\frac{1}{QT}})$, where *PLLength* is the length of the posting list and *QT* is the number of terms in the query.

Search engine results usually include a document ID and also a small portion of the document content surrounding the query term. Such context information cannot be stored on the index servers due to security and space concerns. ZERBER clients request snippets from the peers hosting the top-K documents before presenting the search results to the user. Finally, the user chooses among the top-K documents and clicks on those that are to be fetched from the hosting peers. Algorithm 2 summarizes query processing in ZERBER.

Servers can process queries much faster if they can quickly determine which search results may be in the top-K, and can scan and process only those results. To do this, the server traditionally stores the posting elements in relevance order. However, document ranking is typically based on term frequencies, and our servers should not be able to see these frequencies, as an adversary who takes over a server can reverse-engineer document contents from those statistics. Confidentiality-preserving server-side top-K ranking is an interesting topic for future work.

```
begin func main( )
  Server Servers[]:= getAvailableZerberServers();
  RankedPostingElements[]:=
  User.search(auth_token, query, Servers);
  //Display top-K elements
  for i:=1 to K begin
  url:= toUrl(RankedPostingElements[i]);
  snippet:= getSnippet(url);
  print snippet+url;
  end for
 end func
begin class User
//Process user query on the client side
 begin func search(query, Servers[])
   //PL_IDs: merged posting list IDs to retrieve
   PL_IDs[]:=mapQueryTerms(query);
   for i:=1 to Servers.length begin
    serverAnswers[i]:= Servers[i].
      getPostingLists(PL_IDs);
   end for
   PlainList[]:=decodeShamirsScheme(serverAnswers);
   //Remove false positive posting elements
   PostingElements[]:=
     filterElements(PlainList, query);
   return rank(PostingElements);
 end func
end class
begin class Server
 //Retrieve the posting lists on the server side
 begin func getPostingLists(auth_token, PL_IDs)
  userID:=authenticateUser(auth_token);
  if userID=false, then
   return error;
  end if
  //Select group indexes accessible to the user
  GroupIDs[]:=DB.execute
    ("SELECT groupID FROM groups WHERE userID = "+userID);
  //Retrieve accessible parts of requested posting lists
  PostingLists[]:=
    DB.loadPostingLists(GroupIDs, PL_IDs);
  return PostingLists;
 end func
end class
```

Algorithm 2: Query Processing in ZERBER

6. MERGING HEURISTICS

This section explores posting list merging heuristics that limit query processing costs while preserving ***r***-confidentiality.

Suppose that all the posting lists are merged into *M* lists $L_1,\ldots,L_M$. The total workload cost *Q* for a set of queries is:

$$Q \cong \sum_{L_i \in M} \left[length(L_i) \times \left(\sum_{j \in L_i} q_j \right) \right], \qquad (6)$$

where q_j is the query frequency of term *j,* and length(L_j) is the number of elements in the merged posting list L_i. An efficient posting list merging heuristic must satisfy the ***r***-constraint and minimize the expected workload cost. In other words, the optimization problem is to choose *M* lists such that *Q* is minimal and the ***r***-confidentiality constraint on each list is satisfied. This problem can be shown to be NP-complete by reduction from the minimum sum of squares [14]. Thus we look for merging heuristics that are good in practice.

We first consider a uniform term probability distribution. It can be shown that the ***r*** (confidentiality) value in this case is equal to the number of merged posting lists. For example, if all terms are merged into one posting list, then $\boldsymbol{r} = 1$ and no information about the keywords' document frequencies can be extracted from the index, beyond the adversary's background knowledge $\boldsymbol{B}$. With two posting lists, $\boldsymbol{r} = 2$ and we have half as much confidentiality.

In general, an index ***I*** with ***M*** posting lists increases the $P(\text{DocumentFreq} = DF \mid \boldsymbol{B},\boldsymbol{I})$ probability by a factor of ***M***.

The document frequency distribution in real documents is usually Zipfian, as in Fig. 7. This suggests two strategies for merging. Given an ***r***-value, we can merge terms into posting lists so as to minimize Q while maintaining ***r***-confidentiality. Or, given a maximum value for ***M*** (the number of merged posting lists), we can try to maximize ***r***. We consider three such approaches.

During merging, we create a publicly available *mapping table* that maps a term to the ID of its posting list. The three algorithms differ in the way this table is initialized. All the algorithms base merging decisions on keywords' document frequencies. Though basing merging decisions on query term frequencies is more effective at reducing the total workload cost [26], use of query frequencies would violate our confidentiality goals.

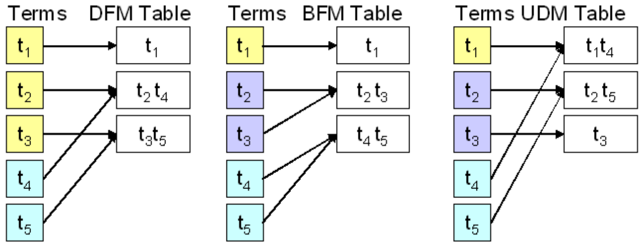


Figure 4: Mapping Table Construction

6.1 Depth First Merging (DFM)

DFM assigns the most frequent terms to separate posting lists, using a predetermined value of ***M*** (the number of merged posting lists) as the table size (see Fig. 4). This exploits the fact that frequently occurring terms are also queried more often. DFM fills the cells of the table from top to bottom with terms sorted by document frequency in rounds until the ***r***-condition in each cell is satisfied. Algorithm 3 gives the DFM procedure.

```
Depth First Merge of Posting Lists (terms [], M, r)

Calculate probability $p_t$ for each term t in terms, the array of all terms
Sort terms into descending order, based on $p_t$
Set the number of posting lists to M, and mark all of them as unfilled
while some term is not yet assigned to a posting list
        go to the next posting list that is not marked as filled
        if sum of the $p_t$ of terms assigned to this list exceeds $1/r$
          then mark the posting list as filled and go to the next list
        else assign term t to this posting list
```

Algorithm 3: Depth First Merging

6.2 Breadth First Merging (BFM)

The Breadth First Merging heuristic (Algorithm 4) sorts terms on document frequency, then assigns successive terms to the first posting list until the ***r***-condition is met. Then BFM moves to the second posting list, and so on until all terms are assigned to a list. BFM does not require us to predetermine ***M***.

```
Breadth First Merge of Posting Lists (terms [], r)

Calculate probability $p_t$ of each term t in terms (the array of all terms)
Sort terms into descending order, based on $p_t$
while more terms need to be assigned to a posting list
        create a new empty posting list
        while more terms need to be assigned and the sum of the $p_t$ of
              terms assigned to this posting list is less than $1/r$
              assign the next term to this posting list
if the r-condition is not satisfied for the last posting list
   // there are not enough terms left to reach a good r-value for this list
then delete the last posting list and randomly distribute its terms
        among the other posting lists.
```

Algorithm 4: Breadth First Merging

6.3 Uniform Distribution Merging (UDM)

UDM is a variation on DFM in which terms are assigned to lists in rounds as in Algorithm 3, but without considering the resulting accumulated probability value. Once all terms are assigned to posting lists, we calculate the resulting confidentiality value as:

$$1/r = \min_{L\ \in M}\left(\sum_{u\in L} p_{t_u}\right), \qquad (7)$$

where ***M*** is the number of posting lists in the mapping table.

DFM and UDM allow us to create an index with a predetermined number of posting lists, and compute the final confidentiality value after merging. BFM allows us to specify the confidentiality value, but the resulting number of posting lists is unknown until the merging is finished. We compare the query workload efficiency achieved by different merging heuristics in Section 7.

6.4 Additional Hash-Based Merging

An adversary can inspect the mapping table and see whether a term is not included in any indexed site. Also, if a rare term is subsequently added to the mapping table, an adversary who has taken over a server can see which site requested the term's inclusion. To avoid this, we use hash-based merging for rare terms that do not significantly change the total probability mass for a specific posting list. We consider a term rare if its original probability was below a certain cut-off threshold.

Hash-based merging works by assigning rare terms to posting lists using a public hash function, so that rare terms never appear in the mapping table. Therefore by inspecting the mapping table an adversary cannot find out whether a rare term appears at any indexed site or not. As the index does not contain any empty posting lists after its start-up period, an adversary cannot use emptiness of a posting list to check whether terms appear at any indexed site. Hash-based merging is also used to distribute the new terms randomly over the index.

7. EVALUATION

In this section, we discuss ZERBER's security guarantees and then evaluate its storage requirements, query performance, and network bandwidth usage compared with an ordinary inverted index, using a real-world web search query log. We also evaluate the effectiveness of the DFM, BFM, and UDM heuristics.

7.1 Security Guarantees

Alice can attempt to amplify her knowledge in many ways. For example, she can issue arbitrary queries and updates and scrutinize the responses. With ZERBER, she cannot learn anything this way that violates the principle of *r*-confidentiality. If Alice takes over a server, she can learn who sends each new query/update to that server; to prevent this, one would need to extend ZERBER to include only *opaque* user IDs in requests and in the user-group mapping.

Alice can see which posting lists each user queries at her compromised server, and see the (opaque) answers. We expect that in practice, the posting list query frequencies she learns will be consistent with her background knowledge. In other words, she will not be able to use this information to violate *r*-confidentiality by improving her guesses about which terms are in each document.

On her compromised server, Alice can see which posting lists are affected by each new update. By monitoring the sequence of updates, Alice can guess that a set of new posting elements refers to the same document. This lets Alice make correlation attacks. For example, suppose that an email contains terms from posting lists *p*1 and *p*2 and that the only two terms from *p*1 and *p*2 that are likely to co-occur are *t*1 and *t*2, respectively. Then Alice can guess that the email contains these two terms, even though the posting elements are encrypted. Thus Alice may be able to violate *r*-confidentiality for newly created documents, though in general she cannot be sure of the exact contents of an element. However, Alice cannot violate *r*-confidentiality for documents committed *before* she compromised the server, as she cannot tell which pre-existing posting elements refer to the same document.

Alice can collude with others to jointly take over multiple index servers, and pool the resulting knowledge. If the colluders take over fewer than ***k*** servers, they will not be able to violate ***r***-confidentiality for documents committed before the attack.

7.2 Storage Overhead

The number of posting elements that ZERBER maintains per index server is the same as in any conventional inverted index. However, ZERBER posting elements include additional fields to identify the term in the merged set and the global element ID, which increases element size by about 50%. Encryption under Shamir's ***k***-out-of-***n*** scheme does not change the element size. Hence, each ZERBER index server uses about 50% more space than an ordinary inverted index. Since ZERBER replicates the index on ***n*** servers, the total index space required is 1.5***n*** times more than for an ordinary inverted index.

Each document server maintains an inverted index (also useful for local search) of its local shared documents, to support efficient updates. This index includes the global ID of each element.

7.3 Network Bandwidth

Insertion and deletion. To index a document, the owner sends its elements to ***n*** servers, so ZERBER uses 1.5***n*** times more network bandwidth for this operation than an ordinary inverted index does.

Deletion from an ordinary inverted index can be implemented by sending the ID of the document to be deleted to the index server. ZERBER elements (and hence the document ID field) are encrypted, so the server cannot determine which posting elements have the same document ID. To delete a document, its owner must delete each element separately. The document deletion network cost is thus the same as its insertion cost.

Query processing. ZERBER query processing is performed in two steps: (1) the client sends the query to ***k*** index servers and retrieves the IDs of matching documents, and (2) the client requests the snippets for the top-K documents from their owners. Step (1) queries ***k*** index servers, requiring ***k*** times as much bandwidth as a traditional lookup. Moreover, a traditional inverted index might return only top-K search results, but ZERBER must return all of the elements accessible to the user. On the other hand, ZERBER uses no additional bandwidth to retrieve lower-ranked search results, while traditional inverted indexes do revisit the server for each page of results.

For our calculations, we assume the following intranet setup: users connect over a 55 Mb/s wireless LAN, while servers use 100 Mb/s LAN connections. We use 2-out-of-3 secret sharing. The document snippets arrive in XML format.

We use a real-world query workload and the Open Directory Project (ODP) data described in Section 7.4. For our experiments we assume the worst case: the user has access to all 100 document collections in the ODP data. In this workload, about 2700 elements are returned from the ODP index per query term on average. Assuming that each posting element is encoded using 64 bits, this is approximately 170 Kb (21.5 KB) per query term response. The queries in the workload contain on average 2.45 terms, which allows for execution of up to 35 queries/second per user and about 200 queries/second answered by each server on average. We expect that the number of queries answered by a server can be increased in an enterprise setting as users typically belong to a smaller number of groups (see Section 7.4.1). On average, each snippet contains about 250 B including XML formatting, which yields 2.5 KB for the top-10 snippets. Thus average total response size for the top-10 results is 24 KB.

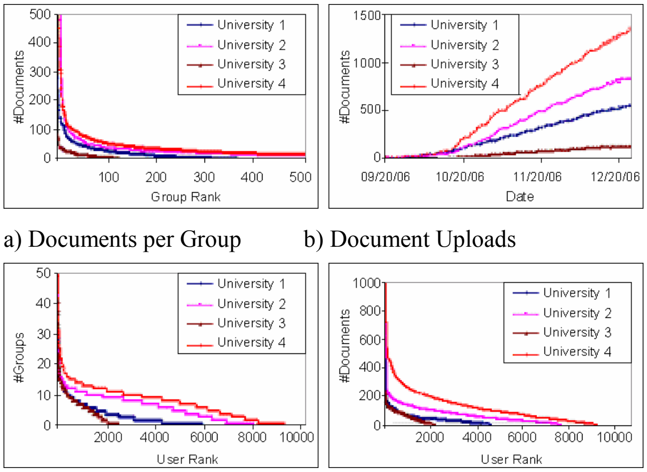
In comparison, Google's response for the top-10 results is about 15 KB, including the snippets as well as information used for presentation purposes (HTML, CSS, etc.), which is 1.6 times less than the ZERBER response size. Altavista returns 37 KB and Yahoo returns 59 KB of top-10 results, which are comparable to or bigger than ZERBER. However, ZERBER's element shares are almost random, so standard HTML compression is ineffective. The compressed responses of Google, Altavista and Yahoo are 3, 2.4 and 1.6 times smaller than ZERBER responses, respectively. (Of course, a ZERBER response contains *all* answers.) Further optimization can be achieved by adding search result checksums and caching them on the client, as defined in HTTP 1.0.

7.4 Experimental Setup

This section provides details about our documents and workload. We used two data sets, from the Stud IP Learning Management System and Open Directory Project crawl data. We used a web search engine query log as the workload, computed as follows. The time to scan a posting list is the sum of the seek time (to position the disk head at the start of the posting list) and the transfer time (the time to read the posting list). The total seek time for a given query workload is a constant, independent of the merging heuristic. The transfer time for a posting list is proportional to its length. Formula (6) is the sum of the posting list lengths, weighted by their query frequencies. Thus the total transfer time (and hence the total workload cost, since the seek time is constant) is proportional to formula (6), which we use as the workload cost in the experiments that follow. All experiments ran on a 2-processor 2.0 GHz Intel CPU T2500 with 2 GB RAM.

7.4.1 Stud IP Data

The Stud IP Learning Management System [32] allows sharing of access-controlled materials within groups of students and teachers. We had access to the Stud IP documents at four universities. Figure 5 provides insights into these data sets. For example, the installation at "University 1" has over 3,300 courses and 6,000 registered students. Most users belong to at most 20 groups and can access fewer than 200 documents. The amount of material stored for each course increases uniformly during the semester (Figure 5b). A mid-semester snapshot used for our experiments contained 8,500 documents with 570,000 terms. At the time of writing, Stud IP did not provide full-text search capabilities and thus we did not have an associated query log.



a) Documents per Group b) Document Uploads

c) Users per Group d) Documents Accessible per User

Figure 5: Stud IP Statistical Profile

7.4.2 ODP Data

We used a collection from the Open Directory Project (a human edited directory of the Web) crawled in 2005, with 237,000 documents and 987,700 distinct terms. The crawler's strategy was to find pages on a variety of topics [24], such that 100 topics were randomly selected; we used the set of documents on one topic as the set of documents of one group.

7.4.3 Web Search Engine Query Log

Our query log has 7 million queries and 135,000 distinct query terms. Figure 6 shows the correlation of the query frequency and the corresponding cumulative query workload (computed using formula (6)).

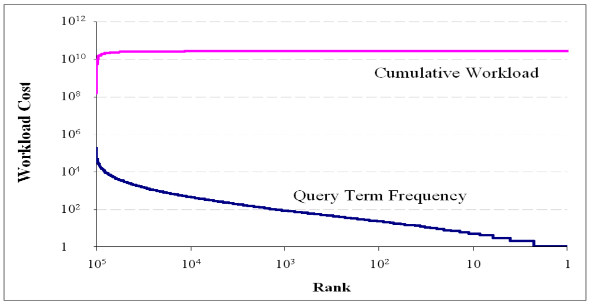


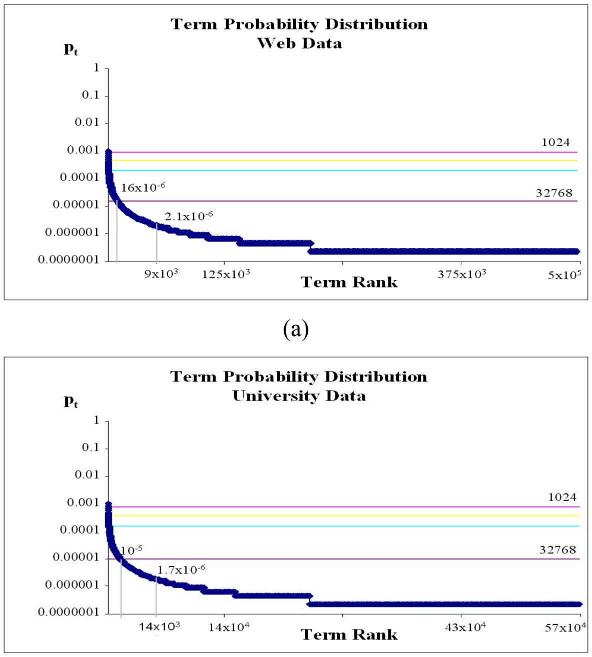
Figure 6: Cumulative Query Workload Cost

The log-scale X-axis shows the query terms in decreasing order of frequency. The most frequent queries constitute nearly the whole query workload. Thus to reduce the total workload cost, the merging heuristic should provide high efficiency for the most frequent queries. As explained earlier, confidentiality concerns require us to base merging decisions on document frequencies rather than query frequencies. These are correlated, though some frequent terms are rarely queried (e.g., "although"). The X-axis in Figure 6 lists the terms ordered from most to least popular.

7.5 Selection of the Confidentiality Level *r*

As described in Section 6, ZERBER's merging heuristics support a tradeoff between query efficiency and confidentiality level ***r***. Whether a particular value for ***r*** is appropriate depends on the (typically collection specific) document frequency distribution. In this section, we examine the choices for ***r*** for the test data sets.

We learned the document frequency distribution from the first 30% of the documents in the two data sets. In ODP this sub-collection contained 70,000 documents and 500,000 terms. As document frequencies follow a Zipfian distribution, this should be sufficient to learn the most frequent terms in the collection. (Terms introduced later should be less frequent, and we assigned them uniformly to the existing posting lists.) Since the central indexes contain documents from many collections on a variety of topics, and we did not remove stop words, the most frequent terms are not specific to any particular collection and the adversary should already know their occurrence probabilities from her background knowledge. Under this assumption, the most frequent terms are least in need of the protection that comes from merging, and our goal is to make it impossible for the adversary to distinguish elements containing the rarer terms from elements containing more frequent terms.



(a)

(b)

Figure 7: *r*-Parameter Selection

We used the term occurrence probability distribution (p_t in formula (2)) to help us set target values for ***r***. Figure 7 shows p_t for the Stud IP and ODP data sets. The X-axis shows the terms in descending order of frequency. The horizontal lines show the 1/***r*** values for 1,024, 2,048, 4,096, and 32,768 posting lists. Both subfigures show that the term probability distribution is Zipfian, with the top few percent of terms far more frequent than others. 10^{-6} is the smallest value of $\boldsymbol{p_t}$ among the 10% most frequent terms. When we merge posting lists, we would like the aggregate term probability of every merged list to be at least this big.

From Figure 7, we concluded that our test data sets should have at most 32K merged lists, because with 65K or more lists, the merging heuristics would not be able to attain our target $\boldsymbol{r}$-value of 10^{-6}. With 32K merged lists, every term with original probability $\boldsymbol{p_t} < 16.09*10^{-6}$ will reside in a posting list with aggregate term probability exceeding that of any but the 1.83% most frequent terms. In Figure 7a, these protected terms reside to the right of the intersection of the 32,768 line with the term probability distribution curve, projected onto the X-axis. Each term to the left of this point will have a posting list of its own under BFM and DFM, and each term to the right will be merged with at least one other term. Methods of choosing a target value for $\boldsymbol{r}$ that adapt to the characteristics of the document frequency distribution are an interesting direction for future work.

We used the DFM and UDM algorithms to create 1K, 2K, 4K, and 32K posting lists, and then computed the resulting $\boldsymbol{r}$ values using formula (7) for the minimal sum of probabilities accumulated in a merged posting list. We tweaked the input value of $\boldsymbol{r}$ given to the BFM algorithm so that it would also produce the same number of lists. Table 1 presents the resulting $\boldsymbol{r}$ values for the web data set. For a given number of posting lists, BFM and DFM produce the same $\boldsymbol{r}$ value. Table 1 shows that UDM offers less confidentiality on average (see also Figure 9).

Table 1: *r*-Parameter Value for 3 Merging Heuristics

# of Posting Lists	1/***r*** for BFM, DFM	1/***r*** for UDM
1,024	$9.30*10^{-4}$	$7.86*10^{-4}$
2,048	$4.45*10^{-4}$	$3.57*10^{-4}$
4,096	$2.07*10^{-4}$	$1.58*10^{-4}$
32,786	$16.09*10^{-6}$	$9.60*10^{-6}$

Figure 8 shows the correlation between $\boldsymbol{r}$ and the number $\boldsymbol{M}$ of merged posting lists for ODP and BFM/DFM. As $\boldsymbol{M}$ increases, the confidentiality level decreases according to the Zipfian term probability distribution in the underlying data (see also Figure 7).

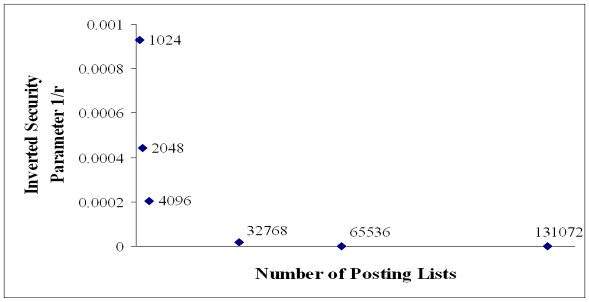


Figure 8: Correlation Between *r* and *M* for ODP & BFM/DFM

7.6 Comparison of Merging Heuristics

In this section, we analyze the security and the query efficiency provided by the different merging heuristics with the ODP data.

Figure 9 shows the amplification $\boldsymbol{r}$ of the original term occurrence probability with different merging heuristics, for ODP data. To improve visibility, we show only the top 1000 terms in the 1,024 index. UDM's curve deviates from the DFM curve and exceeds its $\boldsymbol{r}$-value in several places. However, UDM is comparable to DFM on average, and has the advantage of giving higher confidentiality to very common terms. DFM and BFM give the top 1.83% of terms their own individual posting lists, but UDM merges even these most popular terms.

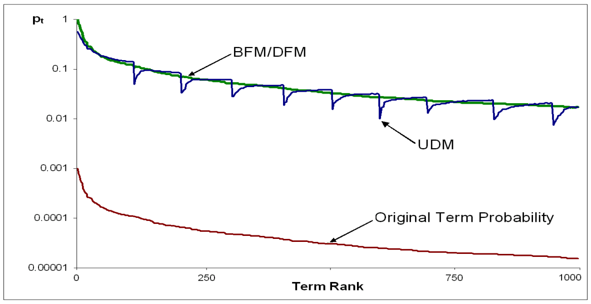


Figure 9: Term Probability Amplification with 1,024 Posting Lists and Different Merging Heuristics

We conducted extensive simulations to evaluate the effects of the merging heuristics on query efficiency. For each data set, we merged the posting lists using BFM, DFM, and UDM for 1,024, 2,048, 4,096 and 32,768 merged posting lists.

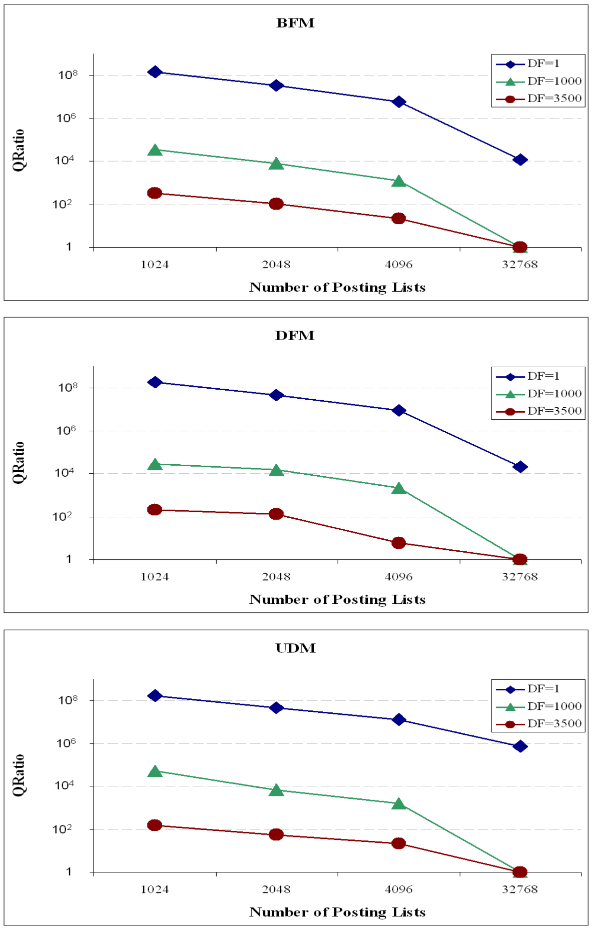


Figure 10: Ratios of Workload Cost for BFM, DFM and UDM

Figure 10 shows the average ratio of the total workload cost *QRatio*(*t*) due to term *t* in its merged posting list *L*, versus the workload cost attributable to t with unmerged posting lists. The curves in the figures correspond to terms with document frequency DF of 1, 1000, and 3500. The X-axis gives the number of posting lists in the index, and the Y-axis shows the workload cost ratio, calculated as:

$$QRatio(t) = \frac{\sum_{u \in L} DF_u \cdot \sum_{u \in L} qf_u}{DF_t \cdot qf_t}, \qquad (8)$$

where qf_x is the query frequency of term x.

Figure 10 shows that as expected, merging mostly affects the costs of queries with rarer terms. Overall, increasing ***M*** significantly improves the cost ratios for terms with low and medium DF. In the 32,768 index with BFM/DFM, queries over terms with high and medium DF are nearly unaffected by merging. Queries over high-DF terms perform well already with only 4K lists. UDM query performance for high- and medium-DF terms is comparable to that of BFM/DFM for 32K posting lists; However, UDM slows down queries over low-DF terms more than the other schemes do.

Comparing Figures 7 and 10, we also see that there is a trade-off between the ability of the index to hide the occurrence of the most frequent terms, and the query processing overhead.

We calculated the efficiency in query answering $QRatio_{eff}$ introduced by different merging heuristics as the ratio between the number of posting elements that correspond to the query term t and the total number of posting elements in its merged list L:

$$QRatio_{eff}(t) = \frac{DF_t}{\sum_{u \in L} DF_u} \qquad (9)$$

Figure 11 plots the efficiency in query answering $QRatio_{eff}$ for indexes with 32K merged posting lists. In this figure, the Y-axis shows $QRatio_{eff}$ and the X-axis represents the query terms in the workload (in %), ordered by $\mathrm{QRatio}_{\mathrm{eff}}$.

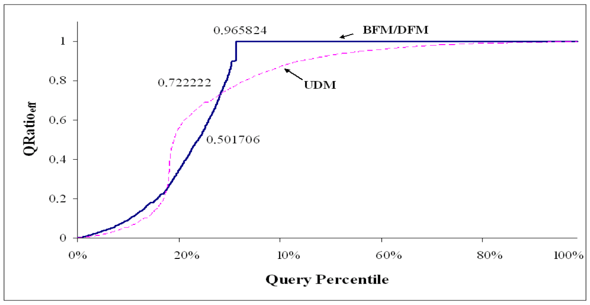


Figure 11: Efficiency in Query Answering

The best query efficiency distribution among the merging heuristics is attained using the DFM/BFM index with 32K lists. In that index, the longest running 70% of the queries in the workload have an efficiency value $QRatio_{eff} > 0.96$ and the next 10% longest-running queries have $QRatio_{eff} = 0.75$ on average. The shortest running 20% of the queries have average $QRatio_{eff} = 0.2$.

Figure 12 plots the response size from the DFM index with 32K lists. The X-axis shows the posting lists ordered by the number of elements they contain, and the Y-axis shows the total number of posting elements in the posting lists, computed as the sum of document frequencies of the terms in a merged posting list. Figure 12 shows that only 40% of the posting lists have a response size exceeding 100 posting elements.

The largest response obtained from the ODP test collection using a DFM-32,768 index contains 10K posting elements. On the platform described in Section 7.4, 700 posting elements are decrypted in 1 msec on average. Thus only 14.3 msec are needed to decrypt the search results from one server for this response.

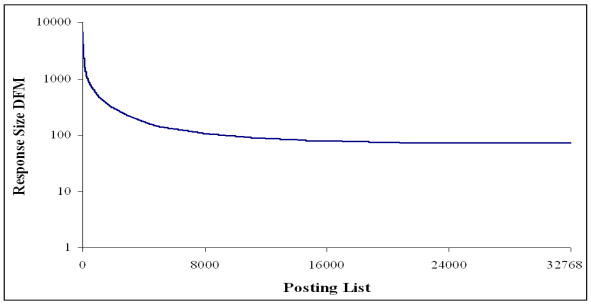


Figure 12: Response Size for the DFM Index with 32K Lists

The simulation results described above have shown that BFM and DFM heuristics offer very reasonable query performance. Our experiments show that the query workload cost ratio for long and middle-size posting lists in the central indexes can be kept comparable to a conventional inverted index while providing security guarantees for 98% of the terms in the data set (with 32K posting lists). The remaining 2% of the terms are common words, which are usually not collection specific (e.g., “remaining”) and do not require protection in the global index containing documents from a number of collections.

This performance can be achieved without learning query statistics, which is important for query confidentiality. BFM is more straightforward to implement given an ***r***-parameter value, as it does not require pre-estimation of the mapping table size.

One of our research questions was the preferable heuristic for posting list merging. In general, there were no significant differences between the BFM and DFM heuristics. In contrast, UDM requires increased bandwidth by queries over low-DF terms and slows down their processing at user peers.

8. CONCLUSION & FUTURE WORK

This paper addressed the problem of secure sharing of distributed documents within working groups in an enterprise. In this situation users need an indexing facility where they can quickly locate relevant documents they are allowed to access, without (1) leaking information about the remaining documents, (2) imposing a large management burden as users, groups, and documents evolve, or (3) requiring users to trust a central authority.

To address these problems, we proposed a tunable *r-confidentiality* measure, as the degree of information from inaccessible documents an index can leak, given an adversary compromises the index and possesses some background knowledge on the corpus and/or language statistics. We presented ZERBER, an r-confidential global inverted index for sensitive documents. ZERBER relies on a centralized set of *largely untrusted* index servers and offers resistance against inappropriate information disclosure even if k-1 index servers are compromised. To provide tunable resistance to statistical attacks, ZERBER employs a novel term merging scheme that has minimal impact on index lookup costs. Our experiments show that ZERBER makes economical use of network bandwidth, requires minimal key management, and answers queries almost as fast as an ordinary inverted index.

Currently, ZERBER returns all answers to a query, and ranking is performed on the client side. A challenging extension is to support top-K processing on the server side, while maintaining the confidentiality properties. Returning only top-K query answers will significantly reduce the network bandwidth and processing costs at user peers. Another interesting question is how to support query confidentiality, even when one server has been compromised and the adversary can view the incoming stream of requests for posting lists. BFM leaks probabilistic information in this situation, while the other merging heuristics are more robust.

9. ACKNOWLEDGMENTS

We are grateful to our colleagues Sergey Chernov and Mohammad Alrifai for supporting this work at the early stage as well as to Christian Kohlschütter, Paul-Alexandru Chirita and Ronny Lempel (IBM) for providing the experimental data. This research has been partially sponsored by the TENCompetence Integrated Project (contract 027087), the National Science Foundation under grants IIS-0331707 and CNS 05-24695, and an IBM Fellowship.

10. REFERENCES

[1] Balke, W., Nejdl, W., Siberski, W. and Thaden, U. Progressive Distributed Top-k Retrieval in Peer-to-Peer Networks. In Proceedings of the ICDE, 2005.

[2] Bender, M., Michel, S., Triantafillou, P., Weikum, G. and Zimmer, C. MINERVA: Collaborative P2P Search (Demo); In: Proceedings of the VLDB 2005.

[3] Bawa, M., Bayardo, Jr. R. J. and Agrawal, R. Privacy-preserving indexing of documents on the network; In Proceedings of the VLDB, 2003.

[4] Bayardo, R. and Agrawal, R. Data privacy through optimal k-anonymization. In Proceedings of ICDE, 2005.

[5] Bertino, E., Castano, S. and Ferrari, E. Securing XML documents with Author-X. In IEEE Internet Computing, May/June 2001.

[6] Bertino, E., Jajodia, S. and Samarati, P. Database security: research and practice. In Information Systems 1995, 20/7.

[7] Bertino, E. and Sandhu, R. Database Security-Concepts, Approaches, and Challenges 2005, Volume 2, Issue 1, 2-19.

[8] Blaze, M. A cryptographic file system for UNIX. In Proceedings of the CCS, 1993.

[9] Blibech, K. and Gabillon, A. Chronos: an authenticated dictionary based on skip lists for timestamping systems. In Workshop on Secure Web Services, 2005.

[10] Boneh, D., Crescenzo, G. D., Ostrovsky, R., and Persiano, G., Public-key encryption with keyword search, In Proceedings of Eurocrypt 2004.

[11] Büttcher, S. and Clarke, C. L.A. A Security Model for Full-Text File System Search in Multi-User Environments; In Proceedings of the FAST, 2005, San Francisco, California.

[12] Chang, Y.-C. and Mitzenmacher, M. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, Feb 2004. http://eprint.iacr.org/2004/051/

[13] Cho, T., Lee, S. and Kim, W. 2004. A group key recovery mechanism based on logical key hierarchy. J. Comput. Secur. 12, 5 (Sep. 2004), 711-736.

[14] Crescenzi,, P. and Kann,, V. A compendium of NP optimization problems. Available at: http://www.nada.kth.se/~viggo/problemlist/.

[15] Fagin, R. Combining fuzzy information from multiple systems. Journal of Computer and System Sciences 1999, Volume 58, Number 1, 216-226.

[16] Fung, B. C. M., Wang, K. and Yu, P. S. Top-down specialization for information and privacy preservation. In Proceedings of ICDE, 2005, Tokyo, Japan, 205–216.

[17] Goh, E., Shacham, H., Modadugu, N. and Boneh, D. Sirius: Securing remote untrusted storage. In NDSS, 2003.

[18] Goodrich, M., Tamassia, R., and Schwerin, A. Implementation of an authenticated dictionary with skip lists and commutative hashing. In DISCEX II, 2001.

[19] Hacigumus, H., Iyer, B. R., Li, C. and Mehrotra,, S. Executing SQL over encrypted data in the database-service-provider model. In Proceedings of SIGMOD, 2002.

[20] Hawking, D. Challenges in enterprise search. In Proceedings of the Australasian Database Conference, 2004.

[21] Herzberg, A., Jarecki, S., Krawczyk, H. and Yung, M. Proactive secret sharing or: How to cope with perpetual leakage. In Proceedings of the CRYPTO, 1995.

[22] Iyengar, V. Transforming data to satisfy privacy constraints. In Proceedings of SIGKDD, 2002.

[23] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q. and Fu, K. Plutus: scalable secure file sharing on untrusted storage. In Proceedings of the FAST, 2003.

[24] Kohlschütter, C., Chirita, P.-A. and Nejdl W. Using Link Analysis to Identify Aspects in Faceted Web Search, SIGIR'2006 Faceted Search Workshop, 2006, Seattle, WA.

[25] LeFevre, K., DeWitt, D. J. and Ramakrishnan, R. Mondrian multidimensional k-anonymity. In Proc. of ICDE, 2006.

[26] Machanavajjhala, A., Gehrke, J. and Kifer, D. l-diversity: Privacy beyond k-anonymity. In Proceedings of ICDE, 2006.

[27] Miklau, G. and Suciu, D. Controlling Access to Published Data Using Cryptography. In Proceedings of VLDB 2003.

[28] Mitra, S., Hsu, W. W. and Winslett, M. Trustworthy keyword search for regulatory-compliant records retention, In Proceedings of VLDB, 2006, Seoul, Korea, 1001-1012.

[29] Shamir, A. How to share a secret. Communications of the ACM, 1979 Volume 22 Issue 11, 612-613.

[30] Singhal, A. Modern Information Retrieval: A Brief Overview. In IEEE, Data Eng. Bull. 24(4), 2001

[31] Song, D. X., Wagner, D., Perrig, A. Practical Techniques for Searches on Encrypted Data. Proceedings of IEEE Security and Privacy Symposium, May 2000, 44-55.

[32] Stud IP LMS. Available at: http://www.studip.de/.