Secure Deletion from Inverted Indexes on Compliance Storage

Soumyadeb Mitra
Dept. of Computer Science
University of Illinois at Urbana-Champaign
mitra1@cs.uiuc.edu

Marianne Winslett
Dept. of Computer Science
University of Illinois at Urbana-Champaign
winslett@cs.uiuc.edu

ABSTRACT

Recent litigation and intense regulatory focus on secure retention of electronic records have spurred a rush to introduce Write-Once-Read-Many (WORM) storage devices for retaining business records such as electronic mail. A file committed to a WORM device cannot be deleted even by a super-user and hence is secure from attacks originating from company insiders. Secure retention, however, is only a part of a document's lifecycle: It is often crucial to delete documents after its mandatory retention period is over. Since most of the modern WORM devices are built on top of magnetic media, they also support a secure deletion operation by associating expiration time with files. However, for the deleted document to be truly unrecoverable, it must also be deleted from any index structure built over it.

This paper studies the problem of *securely* deleting entries from an inverted index. We first formalize the concept of secure deletion by defining two deletion semantics: *strongly* and *weakly* secure deletions. We then analyze some of the deletion schemes that have been proposed in literature and show that they only achieve weakly secure deletion. Furthermore, such schemes have poor space efficiency and/or are inflexibe. We then propose a novel technique for hiding index entries for deleted documents, based on the concept of ambiguating deleted entries. The proposed technique also achieves weakly secure deletion, but is more space efficient and flexible.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design

General Terms

Security, Legal Aspects, Design

Keywords

Inverted Index, Regulatory Compliance, Secure Deletion

StorageSS'06, October 30, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Documents such as electronic mail, financial statements, meeting memos, drug development logs, and quality assurance documents are valuable assets. Key decisions in business operations and other critical activities are based on information in these documents, so they must be maintained in a trustworthy fashion. Businesses increasingly store these documents electronically, making them relatively easy to delete and modify without leaving much of a trace. Ensuring that records are readily accessible, accurate, credible, and irrefutable is particularly imperative given recent legal and regulatory trends. The US alone has over 10,000 regulations [14] that mandate how records should be managed. Many of those focus on ensuring that records are trustworthy (e.g., Securities and Exchange Commission (SEC) Rule 17a 4 [12] and the Sarbanes-Oxley Act [4]).

This has led to a rush to introduce Write-Once-Read-Many (WORM) compliance storage devices (e.g., [5, 8, 10]) for proper data retention. A file committed to the WORM device is read-only and cannot be deleted or altered even by the super-user. A WORM device hence secures critical documents from certain threats originating from company insiders or hackers with administrative privileges.

However, document retention is only one component of its life-cycle. The ability to delete electronic records is as important as the act of securely maintaining them. It is often crucial for an organization to properly dispose of records after a certain point in time. For example, regulations or standards may prohibit their retention after a certain time period [1]. Alternatively, once the mandated retention period for a record has passed, the company is free to dispose of the record. At that point, the record may be a liability for the company and corporate policies might require its deletion (e.g. so that it cannot be subpoenaed in future lawsuits). Since most modern WORM storage [5, 8, 10] devices are built atop conventional re-writable magnetic disks, with the write-once semantics enforced through software, these devices support file deletion operations too. Every file committed to the device is assigned an expiry date that cannot be moved forward in time. Once the file expires, the device makes the file deletable, thereby allowing external cleanup applications to erase it. The WORM device takes precautions to prevent deleted files from being recovered by data forensics. For example, deletion is usually carried out by overwriting the file data multiple times with specific patterns to completely erase any remnant magnetic effects. Techniques for securely deleting files from versioning filesystems have also been proposed by researchers [11].

Unfortunately, just erasing the file from the disk is not adequate to ensure that its contents cannot be recovered. Records are usually indexed on multiple fields to facilitate search and retrieval. The record contents can be reconstructed from the corresponding index entries even after the original record is deleted. For example, all the words of a text document can be obtained from a full-text inverted index built over the document database. When deleting a record, its index entries hence must also be erased.

Since a record can be "logically" modified or deleted by modifying or deleting the index entries pointing to it [17, 9], an index structure stored on a read-write media is not trustworthy. To address this problem, researchers have introduced the concept of *fossilized index*, which is impervious to such logical modifications [17, 9], when maintained on WORM storage. For securely deleting a document, its index entries hence must also be deleted from any such fossilized index maintained on WORM storage.

Unfortunately, the current generation of WORM devices are not suitable for supporting deletion from an index maintained on WORM. For example, they do not support deletion at a byte granularity, which is required for supporting fine-grained deletion of index entries. It is also not feasible to enhance the WORM device with such support, because of the high space and deletion time overhead it will incur. Researchers hence have proposed alternative solutions for hiding index entries of deleted records by encrypting them and discarding the encryption key on expiry [17].

In this paper, we formally study the problem of secure deletion from inverted index. We first define two semantics for *secure* deletion: *strongly secure* and *weakly secure* deletions. We give a hypothetical but impractical scheme for achieving strongly secure deletion and show that the other index deletion schemes that have been proposed only achieve weakly secure deletion. We then analyze some of the problems with the current deletion schemes, and propose a novel technique for hiding the entries of a full-text index, based on the idea of ambiguating the index entries, through merging posting lists.

The rest of the paper is organized as follows. Section 2.1 discusses the threat model. Section 2.2, gives an overview of the inverted index and the deletion problem, under that threat model. Section 3 discusses some of the current index deletion schemes and identifies their deficiencies. Section 4 discusses our proposed solution. Finally we conclude in Section 5.

2. BACKGROUND

2.1 Threat Model

In this paper, we use the terms *document*, *record* and *file* interchangeably. Furthermore, since the focus of this paper is full-text inverted indexes, we assume that the documents are text documents. A text document is an ordered sequence of words, with the words coming from a dictionary.

A document's life cycle spans multiple stages, starting from its creation, its retention and finally its disposal. The document is subject to different threats at each stage of its life cycle. The threat model for the stages of creation and retention was discussed by Mitra et al. [9]. The focus of this paper is the stage after the document retention period is over and the document has to be securely deleted.

A *secure deletion* routine should prevent the adversary Mala from reconstructing the contents of a deleted document. A document can be reconstructed at multiple levels. A *full reconstruction* is one in which Mala is able to reconstruct the full contents of a deleted document, as an ordered sequence of words. In a *set reconstruction*, Mala is able to determine the set of words contained in the document, but not necessarily the order in which they appear. As the set of words in a document is often adequate to infer the meaning of a document, secure deletion method must prevent set reconstruction.

As Mala has superuser privileges, she can read any file or index structure stored on the WORM storage. We assume that Mala does not have access to external information about the file system, such as old backups, logs of past file operations or other meta-level information, such as specific kinds of files always contain certain keywords (for example emails have the *subject* keyword).

We assume that Mala knows the file system metadata (document ID, pathname, commit time, delete time) of the file d she is trying to reconstruct. Let S be a set reconstruction of d obtained by Mala using the above information.

We say that d's deletion was *strongly* secure iff

$$\forall w \; P(w \in d | w \in S) = P(w \in d)$$

$$\forall w \; P(w \in d | w \notin S) = P(w \in d)$$

where $P(w \in d)$ denotes the probability of the word w belonging to document d, while $P(w \in d | w \in S)$ denotes the probability, given that the word w is in S.

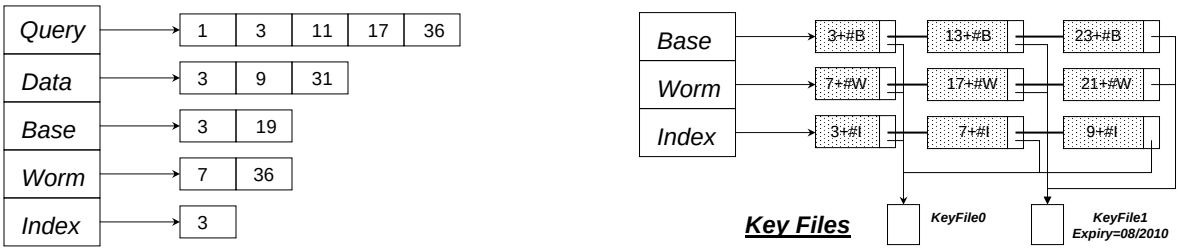
The above definition captures the fact that for a deletion to be strongly secure, the presence (given by $P(w \in d | w \in S)$) or absence ($P(w \in d | w \notin S)$) of w in any reconstruction of d should not convey any information about its presence in the original document. In the next section, we propose a simple, but impractical scheme to achieve strongly secure deletion.

Unfortunately, the practical schemes [17] that have been proposed do not support strongly secure deletion. A much weaker concept of secure deletion is *weakly* secure deletion, in which the adversary Mala cannot prove that the deleted document had a specific set reconstruction. A middle path between strongly and weakly secure deletions would be *probabilistic* deletion, where the probability of the document having a specific reconstruction is bounded above by some constant. Unfortunately it is quite hard to come up with such probabilistic definitions because of lack of precise models for natural languages.

2.2 Inverted Index

Search engines typically use inverted indexes to support keyword search [15]. As shown in Figure 1(a) an inverted index comprises a dictionary of keywords and associated posting lists of document identifiers (IDs) for each keyword. The document IDs are usually assigned through an increasing integer counter[1]. Some search engines store the difference between consecutive document IDs in the posting elements, instead of the full document IDs. This difference usually has a skewed distribution and can be represented using more efficient encoding schemes than fixed-width encoding [15].

[1]We assume that the WORM device maintains the association between document IDs and their full paths. Alternatively, this can be maintained explicitly by the indexing application



(a) Inverted Index

(b) Encrypted Inverted Index

Figure 1: Logical Disposal: The posting elements are encrypted (shown as shaded elements) using the disposal key. To prevent join attacks, the document ID is XORED with the hash of the keyword before encryption.

Keyword queries are answered by scanning the posting lists of all the query keywords, thereby obtaining a list of documents containing those keywords. When a new document is added, the index can be updated by appending its document ID to the posting lists of all the keywords contained in the document. The inverted index can be maintained on WORM by keeping each posting list as a separate appendable file [9].

A traditional inverted index contains sufficient information for creating its set reconstruction. For example, from the index in Figure 1, one knows that document 3 contains the keywords Data, Base and Index. If the index is a full-text index, all the words contained in the document can be reconstructed, even after the document has been deleted. Thus, it is imperative to dispose of all the index entries associated with a record when the record itself is deleted.

3. DELETION FROM INVERTED INDEX

3.1 Secure Deletion

A document d can be deleted from the inverted index by removing its document ID from the posting list of all its keywords. The most secure way of doing that is to create new copies of the document keyword's posting lists, with the document ID erased from them. The original posting list must also be erased. Irrespective of the initial contents of the document, the resulting index structure is the same after the document is deleted from the index. In other words, the resulting index structure, and hence any set reconstruction obtained from it, is independent of the initial contents of the document. Thus, this scheme achieves strongly secure deletion.

Unfortunately, this scheme is impractical. Maling a new copy of many posting lists on every document deletion is prohibitively costly. Furthermore, in the current generation of WORM devices, one must associate an expiry time with the posting list file when the file is created. In the above scheme, however, the expiry time of the posting list should be set to that of the document expiring the soonest, and hence cannot be assigned unless documents are added, and once set it cannot be moved forward in time.

3.2 Physical and Logical Deletion

An alternate way of removing the index entries of a deleted document is to erase (zero-out) the document ID and associated meta-information from the posting list files. Unfortunately, the presence of holes in the posting lists can leak information about deleted documents. For example, the document ID of an erased posting element can be guessed from the neighboring elements. For example, a hole surrounded by 'posting elements with document IDs 4 and 6 must correspond to a deleted document 5. Such attacks can also be based upon commit time order. A hole between two posting elements can only correspond to a document which has been committed in between those two times. Thus this scheme does not provide strongly secure deletion. It also does not achieve weakly secure deletion, always. It may be possible to reconstruct all the keywords in the document, as described above (although it is unlikely that all the posting elements of this document will be surrounded by its successor and predecessor document IDs).

Furthermore, zeroing index entries is not very practical because of the prohibitive cost of implementing such fine grained deletion. Firstly, the WORM device would have to support maintaining and disposing of individual posting elements. Supporting expiry times at a byte granularity, instead of a file granularity, has considerable space and time overhead. Furthermore, deleting small ranges of data within a file can only be accomplished by reading in a full block from disk, erasing the specific entry and writing back the block. In other words, one would have to incur a disk seek for deleting every posting element of a document. This can be prohibitively slow. For example, consider a file with 100 keywords. Assuming 2.5 msec per random I/O and block transfer, it would require about 750 msecs (100 random I/Os and 200 block transfers to read/write the block) to delete one document. In other words the document deletion rate would be 1.3 docs/sec, which is not adequate for most business environments. The presence of huge storage cache might reduce this overhead by caching the posting list blocks in memory. However as observed by Mitra et al. [9] caching only benifits the posting lists corresponding to the popular terms and is usually not very effective in reducing the number of random I/Os, beyond a certain point. Furthermore, since the individual posting lists are usually scattered around on disk, filesystem prefetching schmes are unlikely to be very effective too.

The space and time overhead for deletion can be amortized over consecutive posting list elements if they have the same expiry time. However, adjacent elements of the posting lists of infrequent keywords can correspond to documents which have been committed quite far apart in time and

hence have different expiry times. Furthermore, even if the documents have been committed close in time and are adjacent in a posting list, they can have different expiry times if they are of different types (email, financial records) and hence governed by different retention regulations, or have different intrinsic value (business, personal).

To address these problems, researchers have proposed the concept of *logical deletion* [17]. The key idea behind logical deletion is to encrypt the document IDs in the posting list with a per document secret key. Once the document expires, this encryption key is disposed of along with the document. This prevents the adversary from getting the list of words contained in a specific deleted document. Unfortunately, an adversary can still join on the encrypted document IDs and prove the existence of a document containing a certain set of keywords. Equivalently the adversary can join on the key-pointers, stored with each posting list element.

As shown in Figure 1(b), Zhu [17] addresses these problems by dividing the documents into deletion groups according to their expiration time. The posting elements of all the documents in a disposition group are encrypted using the same key. This key is stored in a separate *key file*, whose expiry time is set to that of the disposition group. To avoid having the same ciphertext for each occurrence of a particular document ID, which would enable an adversary to reconstruct the record by performing a join on the ciphertext, the combination of the index key and document ID is encrypted together. Disposition with this scheme is very efficient, as one has to erase just one file (the key file) to dispose of all the posting elements corresponding to that disposition group.

Once the key is deleted, the adversary can still obtain the set of all posting elements corresponding to that disposition group, by joining on the key file pointers stored with the posting elements. That is, she can determine a set of keywords that were committed in documents in that disposition group. However, she cannot determine the exact association of those words with documents. Unfortunately, this is not adequate when context information can be used to partition words into documents. For example, if keywords "Martha Stewart", "Ralph" and "ImClone" were mentioned in the same disposition, one might guess that they appeared in the same email document, particularly if the other documents in the same disposition group are of different types or are email communication between users who are unlikely to communicate with either Ralph or Martha. Furthermore, document IDs of "logically" deleted documents can be guessed from neighboring undeleted document IDs.

Both the physical and logical deletion schemes also considerably increase the size of the index. Posting lists can usually be compressed to under a byte per posting element [15] by storing the encoding of the difference between document IDs. Such compression schemes cannot be used if the posting elements are discarded: One must have access to all the intermediate document ID differences to obtain any specific document ID. Furthermore, for logical disposition one also has to store a key file pointer (typically 3-4 bytes) with every encrypted document ID. Thus the above scheme can increase the index size by a factor 8.

Logical disposition is also inflexible. It is not possible to extend the retention period of any individual document in the index, since the disposition key is associated with a group of documents. One option to is to make a new copy of the document in the index with the new expiry time, but this can be highly space inefficient if the retention period of a large set of documents has to be changed (for example if the company decides to increase the retention period of a subset of financial emails).

4. MERGED POSTING LISTS

The concept of merged posting lists was motivated by the need to support efficient online updates of posting lists [9]. As explained above, each posting list can be maintained as a separate file on WORM. The index can be updated when a new document is added by appending its document ID to the posting lists of all the keywords it contains. Unfortunately, this operation can be prohibitively slow, as each file append will require a random I/O. For example, if each document contains almost 250 keywords on average and each append incurs a 2 msec random I/O, it would take 1/2 second to index a document. Mitra et al. [9] showed that even adding a large amount of storage cache (of the order of 4-64 GBs) does not reduce the I/Os per document enough to make it acceptable for typical business needs.

In order to address this problem, they proposed [9] the concept of merging posting lists, as shown in Figure 2. If the posting lists are merged so that the total number of lists is no more than the number of cache blocks in the storage device, then all the posting list updates hit the cache. Simulations on real world datasets showed that such merging gives a factor of 20 speedup over using a storage cache of 4 GB and no merging, and a factor of 500 speedup over the uncached case. Merging however, increases the query response time, because longer posting lists have to be scanned. However, simulations using a real world user query log obtained from a large business organization showed that the query throughput slowed down by under 10% when the posting lists were merged to form 32K lists

In order to disambiguate between the different keywords in the merged posting list, one must store the encoding of the keyword with each posting entry in a merged list. This encoding can be stored in $\log(q)$ bits, where q is the number of keywords that are merged together. Mitra et al. [9] evaluated different merging strategies like isolating the top queried terms, the top merged terms and random merging. Simulation results showed that uniform merging with 100-200 keywords merged into one list gives good query performance with a reasonable storage cache size (256 MB).

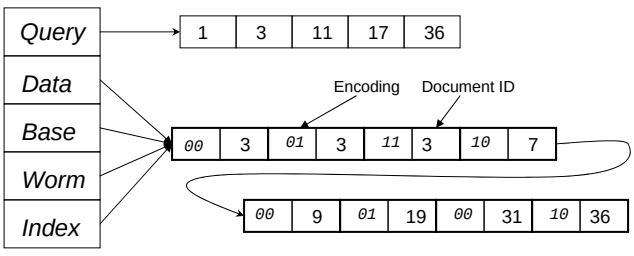
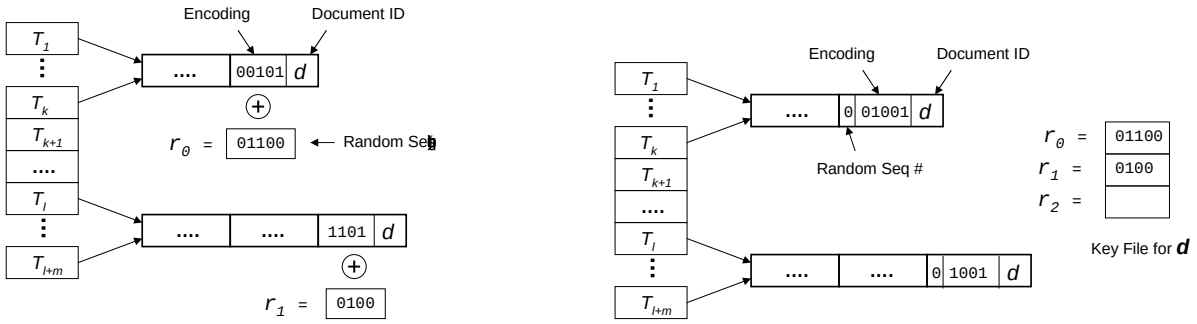


Figure 2: Inverted Index with Merged Posting Lists. An encoding for the keyword must be stored with each posting element in the merged list. In the above example, the keywords Data, Base, Worm and Index are assigned encodings 00, 01, 10 and 11 respectively.



(a) Original Merged Index

(b) Encrypted Merged Index

Figure 3: The keyword encodings of a document are XORed with a set of random sequences. The sequence itself is stored in a separate key file, which is discarded when the document expires. The posting elements must also maintain the sequence numbers (0,1,2..) that must be used to decrypt the encoding shown in (b).

4.1 Exploiting merged list for hiding

The basic idea behind our proposed scheme is to encrypt the keyword encoding stored with each posting element in the merged posting list. One possible encryption technique is to store, in the posting element, the XOR of the keyword encoding with a secret key k, instead of storing the pure encoding E itself. The key k can be stored with the document and can be deleted when the document expires. This scheme gives perfect encryption, without loss of functionality. While the key is present (document has not expired), the encoding E can be extracted from the posting element. However once the key has been deleted, the keyword encoding E cannot be retrieved from $E \oplus k$ stored with the posting element. In other words the adversary cannot determine which of the q keywords merged together, the posting element corresponds to, after the key is discarded.

We can generalize this scheme to handle more than one keyword per document. Consider a document D with n keywords, $W_1, \ldots, W_n$. Suppose that the keywords belong to the merge sets $S_1, \ldots, S_m$ respectively: The posting lists of all the keywords in each S_i are merged together. Also, let $E_1, \ldots, E_n$ be the encoding for the words $W_1, \ldots, W_n$, such that E_i uniquely identifies W_i within the set S_i. As explained above, E_i can be stored in $\lceil \log(|S_i|) \rceil$ bits.

The encoding E_i can be encrypted by XORing it with a random sequence. Specifically, we generate a set of random bit sequences $r_1, \ldots, r_n$ of lengths $|E_1|, \ldots, |E_n|$ respectively, where $|E_i|$ $(= \lceil \log(|S_i|) \rceil)$ denotes the length of the ith encoding. The keyword encoding E_i is stored by XORing it with r_i. Figure 3 gives an illustration of the proposed scheme. The keywords $T_1, \ldots, T_k$ and $T_l, \ldots, T_{l+m}$ are merged together into single posting lists. The keyword encoding for document ID d is XORed with random sequence r_i before storing it in the index. The random sequences themselves can be stored in a separate key file, which can be deleted when the document expires. Assuming a typical document size of 100 keywords (average email size) and $q = 200$, one would need a key file of $100 * \log(100)$ bits or about 0.1 KB.

This scheme prevents set reconstruction of documents: The adversary cannot reconstruct the words $W_1, \ldots, W_n$ from the encrypted encodings. That is, the adversary cannot determine which of the $|S_i|$ different keywords this specific posting element corresponds to. We analyze the security provided by this scheme in the next section.

While scanning the posting lists during query processing, the corresponding key file must be read from disk. Since the document IDs themselves are stored in plain-text format, the corresponding key file can be identified by separately maintaining a mapping between document IDs and their key files. This is unlike the previous scheme for logical disposition where the document IDs were encrypted and hence key pointers had to be maintained in every posting element. Furthermore since the document IDs are never explicitly deleted, they can be compressed by storing the difference between consecutive document IDs, as was explained in section 2.2.

The above scheme however requires an additional key sequence number (i of r_i) to be stored with every posting element, so that the correct key is used for decryption. An expected posting element size in this case is about 3 bytes, 1 byte for posting element, 1 for encrypted encoding for $q = 100$ and 1 for key sequence numbers assuming 2^8 keywords per document.

Unfortunately, the above scheme is likely to have poor query performance. For every posting element, an I/O operation is required to fetch the corresponding key file from disk. The considerably large size of the key files (1KB) also makes it infeasible to store key files for all the documents in memory. One way to address this problem is to generate the set of random sequences using a keyed pseudo-random sequence generator. Instead of storing the entire key sequence, one needs to only store the key used as a seed to the sequence generator. This key can be stored along with the document and can be discarded once the document expires. Since the seed is much smaller (typically 8-16 bytes) than the key sequence, the keys for all the documents can be pre-loaded into memory during query processing.

4.2 Comparison with Logical Deletion

Although the proposed scheme does not achieve *s*trongly secure deletion, it offers certain advantages over the previously proposed logical disposition scheme. For example, unlike logical disposition, it is secure from correlation attacks.

With logical deletion, the presence of correlated words like "Martha Stewart", "Ralph" and "ImClone" in the same disposition, can point to a existence of a document containing all of those terms. With merged posting lists however, one cannot determine the exact term(s) that a particular document or a set of document contains, if the set of terms that are merged is choosen appropriately (We discuss some merging heuristics below).

The merging scheme offers other advantages too. For example, it works with compressed lists and has a substantially lower space overhead (by more than a factor of 2). It does not require the documents to be disposed of in groups and hence it can support fine grained deletion of documents. Increasing the retention period of a document does not require making new copies of the posting elements.

4.3 Merging Heuristics

The document anonymity provided by keyword merging scheme critically depends on the set of keywords that are merged together. One good merging heuristic can be to group keywords from the same parts of speech. For example, merging adjectives like "bad", "ugly" with their antonym words like "good" and "beautiful" can help ambiguiate the meaning of the documents. Such an analysis of merging strategies for words can be done offline and built into the indexing application.

Nouns must be treated differently, because the set of nouns is domain (usage) specific and hence cannot be pre-trained into the index application. Furthermore the presence of nouns in a document often can reveal critical information. For example if all the words "Martha Stewart", "Raplh" and "ImClone", occur in a single document, it might be a strong indication of some communication between them. One possible strategy for handling such cases is to merge personally identifiable information together in the index. Merging fields like name, SSN and email address of multiple people can help anonymize the sender and the receiver of the message. Named entity recognition is a well researched field [2, 3, 6, 16] from which techniques can be adopted directly.

Non-personally identifiable nouns like names of cars, cities, etc can be treated by dividing them into conceptual classes. Keywords belonging to a similar "conceptual class" should not all be merged together (or at-least merged with other conceptual classes). As an example, if the merged set is obtained by merging the names of all possible cars, the adversary would know that an encrypted keyword from that merged set is about cars. Considerable work has been done on clustering documents and words [7, 13], which can be directly used in our scenario. Similarly numerical strings can also me merged together.

Another possible heuristic is to merge words having similar frequencies, so that the probability of the encrypted word being any of the words from the merged set is roughly equal for all the keywords.

5. CONCLUSION

In this paper, we studied the problem of *securely* deleting entries from an inverted index. We analyzed the deletion schemes that have been proposed in literature and identified their key deficiencies. We then proposed a new deletion scheme based on the idea of ambiguating keywords.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their invaluable feedback. We would also like to thank Windsor H. Hsu of IBM Almaden Research Center for useful discussion about the problem and Sruthi Bhandakavi for reading the paper and suggesting important changes. This research was supported by NSF under grants IIS-0331707, CNS-0325951 and CNS-0524695.

7. REFERENCES

[1] Dod 5015.2-std, design criteria standard for electronic records management software applications. http://www.dtic.mil/.

[2] X. Carreras, L. Màrques, and L. Padró. Named Entity Extraction using AdaBoost. In *Proceedings of CoNLL-2002*, pages 167–170. Taipei, Taiwan, 2002.

[3] H. Chieu and H. Ng. Named entity recognition: A maximum entropy approach using global information, 2002.

[4] Congress of the United States of America. Sarbanes-Oxley Act of 2002, 2002. Available at http://thomas.loc.gov.

[5] EMC Corp. EMC Centera Content Addressed Storage System, 2003. Available at http://www.emc.com/products/systems/centera_ce.jsp.

[6] R. Florian. Named entity recognition as a house of cards: Classifier stacking. In *Proceedings of CoNLL-2002*, pages 175–178. Taipei, Taiwan, 2002.

[7] T. Hofmann. Probabilistic Latent Semantic Indexing. In *Proceedings of the 22nd Annual ACM Conference on Research and Development in Information Retrieval*, 1999.

[8] IBM Corp. IBM TotalStorage DR550, 2004. Available at http://www-1.ibm.com/servers/storage/disk/dr.

[9] S. Mitra, W. Hsu, and M. Winslett. Trustworthy Keyword Search for Regulatory Compliance. In *VLDB 2006*, Sept 2006.

[10] Network Appliance, Inc. SnapLock^{TM} Compliance and SnapLock Enterprise Software, 2003. Available at http://www.netapp.com/products/filer/snaplock.html.

[11] Z. N. J. Peterson, R. Burns, and J. Herring. Secure deletion for a versioning file system. In *FAST 2005*.

[12] Securities and Exchange Commission. Guidance to Broker-Dealers on the Use of Electronic Storage Media under the National Commerce Act of 2000 with Respect to Rule 17a-4(f), 2001. Available at http://www.sec.gov/ rules/interp/34-44238.htm.

[13] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques, 2000.

[14] The Enterprise Storage Group, Inc. Compliance: The effect on information management and the storage industry, May 2003.

[15] T. C. Wittenm I. H., A Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufman, San Francisco, CA, 1999.

[16] G. Zhou and J. Su. Named entity recognition using an hmm-based chunk tagger, 2002.

[17] Q. Zhu and W. Hsu. Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records. In *ACM SIGMOD Conference 2005*, June 2005.