An Encrypted, Content Searchable Scalable Distributed Data Structure

Thomas Schwarz[1], Peter Tsui[1], Witold Litwin[2]

[1] *Department of Computer Engineering*
Santa Clara University
{tjschwarz, ptsui}@scu.edu

[2] *CERIA*
Université Paris Dauphine
Witold.Litwin@dauphine.fr

Abstract

Scalable Distributed Data Structures (SDDS) store data in a file of key-based records distributed over many storage sites. The number of storage sites utilized grows and shrinks with the storage needs of applications, but transparently to them. An application can search records by key or by content in parallel at all storage sites. The need for privacy of the data at the storage sites might require the encryption of the records. However, the scheme needs to preserve the capability to search in parallel. We propose a scheme that achieves this goal. We create a collection of additional SDDS indices. We encrypt these so that we can still perform string searches performed in parallel at the storage sites. We present the scheme and evaluate its strength as well as storage and access performance.

1. Introduction

The bandwidth of modern networks and the price of commodity computers have lead to an explosive interest in "multicomputers", systems utilizing many interconnected computers (called the nodes or sites). Multicomputer data might be stored in the distributed RAM of the sites or on their disk drives. Often, the components might be used in other capacities as well. For example, a large organization might store high-traffic data on the workstations of their employees making use of excess capacity. Farsite [A&al02, BDET00, DW01] is an example of this architecture for a large organization where files are stored redundantly and in encrypted form on the various workstations. Whenever a legitimate user has access to a node of a multicomputer, the privacy of the stored data becomes an issue.

Scalable Distributed Data Structures (SDDS) is a class of data structures that offer constant speed operations in a multicomputer, independent of the number of nodes. For example, the SDDS version of linear hashing, LH* [LNS96], and its scalable high availability version, LH*$_{RS}$ [LMS05], both offer fast, constant time lookup of records in a multicomputer. They also allow for parallel (sub-)string searches.

In our setting, we want to encrypt records for SDDS such as LH* and LH*$_{RS}$ while retaining the capacity for parallel searches at the components of the multicomputer. If we use strong encryption, then the contents become unsearchable. The sheer size of the database makes it impossible to send encrypted data to a client (or trusted node), decrypt the data there, and search the data. If we use weak encryption, then an attacker can break the scheme. We therefore compromise as follows. We strongly encrypt the records themselves. In addition generate *index records*. These are also encrypted, but not as strongly, in order to allow generic pattern searches on the main records through the index records. Our contribution in this paper is an index encryption method that makes the overall storage of the application data secure for typical applications.

The structure of the SDDS records is usually assumed to be flat, i.e. to consist of a key and a single data (non-key) field. Techniques like the one of Agrawal *et al*. [AKSX04] are appropriate for encrypting numeric keys and *a fortiori* for other keys in a relational database table with many fields, but do not solve the problem for flat records. Next, in contrast to the work by Song *et al.* [SDW00], we want to be able to search for arbitrary patterns, not just words. We aim at a unit of search as close as possible to a single data symbol without compromising security.

In more detail, we use Electronic Code Book encryption (ECB) on blocks of characters to generate our index records. We further mitigate ECB's

susceptibility to a frequency analysis. The way of doing this at the index records while reducing the overall operational and storage costs of maintaining index records is the core of our proposal. First, we enhance the protection of our index records by *redundancy removal*, where we use lossy compression to make the encrypted index records look more like random data. A side-effect of redundancy removal is the creation of false positives that reduce the accuracy of string searches. We show the trade-offs between search accuracy and the overall strength of our encryption. Second, we disperse our ECB-encrypted and compressed index records over several sites. This limits the amount of information available to an attacker at a single site. Our experimental, initial results (gained from a hard case) show these trade-offs. They do not yet measure the efficiency of dispersion.

Our threat model worries about data confidentiality than about data integrity and availability. We primarily protect data against the owners and other users of the storage nodes and only incidentally against an intruder. We are not concerned with network security e.g. the authentication scheme. For instance, we do not deal with the design of the authentication scheme needed. We also assume the use of any of the schemes that make traffic analysis difficult, e.g. [CGKS95].

All things considered, our scheme seems to achieve a medium level of security. It is secure enough to defeat a determined attacker with commonly available computing resources. It is probably not secure enough to withstand an attacker with outstanding computing resources and with insider knowledge of the underlying data. In addition, dispersion is vulnerable against collusion among those storing index records. However, in an SDDS environment, collusion should be rather difficult since a nodes does not have access to the data dispersion scheme and consequentially cannot easily determine the other nodes where a particular index record has been dispersed.

Below, we give the details of our scheme. We then explain redundancy removal from the index records and index record dispersion. Next, we describe the complete scheme. Finally, we propose methods for assessing security and give a preliminary evaluation. We use an extract from the San Francisco phone book consisting only of names and numbers. This is a difficult case for our scheme because the records are so small consisting only of a full name and a phone number.

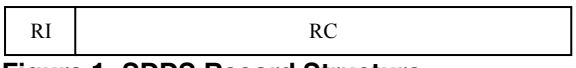

Figure 1: SDDS Record Structure

2. Basic Scheme

A record consists of a key, that is the Record Identifier (RI) and of the Record Content field (RC) (Figure 1). We assume that the key is artificially created number and not sensitive information. The RC field is a flat, zero terminated string consisting of symbols, typically either 8-bit ASCII symbols or 16-bit Unicode symbols. We store the database (collection of records) as an SDDS over multiple sites in strongly encrypted form. We then generate the index records as follows. We *chunk* an RC into chunks of equal length. We then encrypt each chunk using Electronic Code Book (ECB) encryption. This is our *Stage 1*.

ECB allows (at least in principle) frequency analysis. We strengthen the encryption by two different methods: First, we remove redundancy through lossy compression, that is, we preprocess the symbols by placing them into a smaller number of buckets and encode them by bucket number. This introduces false positives, but makes attacks on the encryption more difficult. This is our *Stage 2*. Second, since even removing redundancy from the original records might not be sufficient, we disperse the index records over several sites, our *Stage 3*, in order to limit the amount of information available to the attacker at any single site. We illustrate our scheme in Figure 3.

2.1. Creation of Index Records, Stage 1

From a single RC, we create s "chunked" RC made up of chunks of *s* consecutive symbols in the following way: Assume that the RC is (r_0, r_1, $r_2 \ldots r_N$). The first file consists of the chunks (r_0, r_1 … r_{s-1}), (r_s,r_{s+1} … r_{2s-1}), … Thus, chunk i is made up of ($r_{(i-1)s}$, $r_{(i-1)s+1}$, … r_{is-1}). If necessary, we pad the last chunk with zero symbols (denoted by 0) to create the last chunk. The second chunked RC starts the chunking with an offset of one instead of zero. Thus, this chunked RC is (0,0, …, 0, r_0), (r_1, r_2,…, r_s), … Chunk i is ($r_{(i-1)s+1}$, $r_{(i-1)s+2}$, … r_{is}). We similarly proceed to create the chunking for chunked RCs 2, 3...s−1. We then use *Electronic Code Book* (ECB) encryption on all the chunks in the *s* chunked RC records to generate *s encrypted, chunked RC*. (Basically, ECB uses standard secret key encryption to generate a seemingly random, reversible mapping of clear-text chunks to encrypted chunks of the same size.) We store the *s* RC on *s* different sites.

Beginning and ending chunks can create a security problem. For example, a beginning chunk in the second chunked RC has the form (0,0,…0, r_0). This can be recognized because there are at most as many

encrypted first chunks than there are symbols and exploited through an elementary frequency attack. The gained decoding can then be used to speed up password cracking on the ECB or might prove useful all by itself. A simple counter-measure such as not storing these "partial" chunks limits our search capability, but is otherwise perfectly feasible.

2.2. Example

Assume that we chose s = 4, a rather low value that might allow a successful frequency analysis. Assume that our RC is "ABCDEFGHIJKLMNOPQR STUVWXYZ". Then, the first chunked RC consists of "(ABCD),(EFGH),(IJKL),(MNOP),(QRST), (UVWX), (YZ00)," the second chunked RC of "(000A), (BCDE), (FGHI), (JKLM), (NOPQ), (RSTU), (VWXY), (Z000)", the third RC of "(00AB), (CDEF), (GHIJ), (KLMN), (OPQR), (STUV), (WXYZ)", and the fourth one of "(0ABC), (DEFG), (HIJK), (LMNO), (PQRS), (TUVW), (XYZ0)". To generate the encrypted RC, we encrypt each chunk individually.

2.3. Searches

To search for a substring $(q_0, q_1, \ldots q_{l-1})$, we create all possible chunkings of this string. Assume that $l = x \cdot s + y$. Then we create the chunking series $(q_0, q_1, \ldots q_{s-1})$, $(q_s, q_{s+1}, \ldots, q_{2s-1})$, $\ldots$ $(q_{(x-1)s}, q_{(x-1)s+1, \ldots} q_{xs-1})$; $(q_1, q_2, \ldots, q_s)$, $(q_{s+1}, q_{s+2} \ldots, q_{2s})$, $\ldots$, $(q_{(x-1)s+1}, q_{(x-1)s+2, \ldots} q_{xs})$; … Here we only include chunks into the *s* series that consists entirely of *s* substrings of length *s* of the search string. We then send all series to all the sites, who try to match consecutive chunks. If the substring is contained in the record, then all sites indeed report a hit. However, since we do not search for the complete string at all sites, some sites might report a false positive. It is not possible that a search results in false positives from all sites.

Unfortunately, our search strategy does not work for search strings of length less than *s*. We can "kludge" a search strategy for search strings of length *s*–1 by adding all possible characters to the end of the string. This method is wasteful and might pose a security risk if an attacker snoops network traffic.

2.4. Example (Continued)

Assume that we want to search for the string ``BCDEFGHIJK". We produce all possible s = 4 chunkings of this string (without zero symbols). That is, we produce the following *chunked* search strings:

- (BCDE)(FGHI)
- (CDEF)(GHIJ)
- (DEFG)(HIJK)
- (EFGH)

We then send all the chunkings to any one of the storage sites. Site 1 will have a hit for the fourth string; Site 2 will have a hit for the third search string; etc. We observe that each chunked search string has a hit in exactly one index record. In general, each chunked search string has to be found in one of the index records, because of false positives or because of repeating characters, there might be more hits. This shows that we could use less storage or less search strings, but in this case, it is possible to have false hits. For example, assume that we only use one storage site. As our example shows, the string "ACDEFGHI" will generate the same hit for storage site one – if that is the one we picked – as our original search string because the critical chunked search string, number 4, is the same as that generated from the original search string.

2.5. Limiting Storage Overhead

Our basic scheme stores the same data in *s* sites and creates and searches for *s* search strings for every search operation. We can avoid this overkill at the costs of a small increase in false positive hits. For example, assume a chunk size of $s = 8$. We can decide on 4 storage sites, containing the following chunkings

$(r_0, r_1, \ldots, r_7)$, $(r_8, r_9, \ldots r_{15})$, $(r_{16}, r_{17}, \ldots r_{23})$, …
$(0, 0, r_0, r_1, \ldots r_5)$, $(r_6, r_7, \ldots, r_{13})$, $(r_{14}, r_{15}, \ldots, r_{21})$, …
$(0, 0, 0, 0, r_0, \ldots r_3)$, $(r_4, r_5, \ldots r_{11})$, $(r_{12}, r_{13}, \ldots r_{19})$, …
$(0, 0, 0, 0, 0, 0, r_0, r_1)$, $(r_2, r_3, \ldots r_9)$, $(r_{10}, r_{11}, \ldots r_{17})$, …

for each record D-field. When we search for substring $(q_0, q_1, \ldots q_{l-1})$, we generate two search chunkings

$(q_0, q_1, \ldots q_{s-1})$, $(q_s, q_{s+1}, \ldots q_{2s-1})$, …
$(q_1, q_2, \ldots q_s)$, $(q_{s+1}, q_{s2}, \ldots q_{2s})$, ...

and send both to all the sites. For each occurrence of the substring, only one site will report a hit. Thus, false positives will be more numerous. In addition, using four data sites implies that the length of the search string needs be at least *s*+1.

Alternatively, we can use only two storage sites, but now have to send four search strings. The minimum length of a search string is now *s*+3.

We give a simpler example in Figure 2, adapted from our telephone database that we use as a base for our experimentation. The records has identifier RI = 007 and consists of a (fake) telephone number and a name. We use underscores to denote the space. When we search for the last name "Schwarz", we should actually search for " Schwarz " with a leading space and a

trailing zero. Since we generate two index records with chunk size 4 (Figure 2a), we only need to generate two index search records (Figure 2b). When we actually perform the search, we have a hit for the first search string in the second index record.

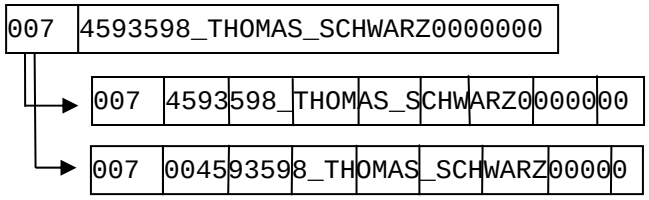

Figure 2a: Example Record Chunking

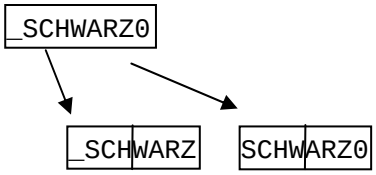

Figure 2b: Search Strings

Figure 2: Search Example

3. Redundancy Removal (Stage 2)

Separating record store sites and record index sites is motivated by the possibility to compress the index records, but mainly because a lossy compression of the index records makes a frequency analysis more difficult. We use the following strategy for a lossy compression. Assume that we start out with 2^f symbols. If our symbols are ASCII characters, then f =8 and we start out with 256 symbols. Assume a chunk size of g symbols. Since we want to retain the capability for searching, we replace each chunk by a smaller chunk. For example, we might decide to map each chunk consisting of 4 byte-symbols by a chunk of 16 bits only. When we pick the mapping, we try to equalize the frequency for each nibble, both in order to make a frequency attack more difficult and in order to limit the new false positives in searches that lossy compression creates. In order to come up with a good mapping, we can preprocess a representative part of the database and count the occurrence of each chunk. We then place these characters into buckets, one for each encoded symbol, in order of frequency of occurrence. For small chunk sizes, this will not necessarily result in an equal frequency distribution of the encoded symbols, but it will certainly remove the larger frequency spikes. For example, the frequency distribution of letters in English text is well known [PE82]. Were we to create a lossy encoding of letters only in only 3 bits, then each encoded symbol should occur at a frequency of 1/8 = 12.5%. But the letter "E" alone already accounts for 12.702% of all occurrences. Our procedure becomes impossible for larger chunks sizes simply because there are just too many possible chunks. In this case we can at least preprocess the records encoding each symbol into a smaller one or using the same procedure for subchunks.

4. Dispersion of Index Site Records over Several Sites (Stage 3)

A chunked record in encrypted form still contains the same information that the original record has. To reduce the amount of information to an attacker (assumed primarily to have access to a single site), we can divide the information contained in it and store it on separate sites. We now propose a way to disperse the index records over k sites. A good value for k needs to divide the chunk size in bits and be small enough to limit the number of false hits (as we will see). For this reason, a good value for k would be 2 or 4.

Assume then that the chunk size is $c = s \cdot f$ bits, where s is – as above – the number of characters in each chunk and 2_f is the number of characters. k has to be a divisor of c. Write $c = g \cdot k$. We construct a Galois field (finite field) $\Phi = GF(2^g)$. In short, the elements of Φ are bit strings of size g. Addition and subtraction are defined as the bitwise XOR of two operands. Multiplication and division are more involved operations, but there exist a number of good methods to implement them in the literature. The rules of algebraic manipulations in the field of real numbers, complex numbers, rational numbers, etc. apply as well in Φ. Let **E** be an invertible k by k matrix over Φ. A good **E** seems to be one where all coefficients are non-zero. (Since k is small and g is larger than k, such matrices exist in abundance, e.g. as Cauchy matrices of Vandermonde matrices.) Let **c** be a chunk, i.e. a bit string of size c. By breaking **c** up into pieces of size g, we can write **c** as a row vector of dimension k in Φ: $\mathbf{c} = (c_1, \ldots, c_k)$. We then calculate $(d_1, \ldots, d_k) = \mathbf{d} = \mathbf{c} \cdot \mathbf{E}$ for each chunk in a index record. We then store d_1 in the first site, …, and d_k in site k.

We could use other ways of breaking up chunks and distributing them to different sites. Since we only need to tell whether chunks are the same or not, any dispersion algorithm (such as erasure correcting codes popularized as IDA [Ra89]) that maintains the same information as in the original chunk will do. Our encoding seems to be appropriate because it is fast (if Galois field multiplication can be implemented by small tables) and because a dispersed symbol d_i is calculated from the whole chunk and not just a piece of

the chunk. This makes a frequency analysis on the contents of one of the dispersion sites more difficult.

To implement a search, we break the chunks in the search string up according to the same rules. We then send the broken up pieces to the dispersion sites. If all dispersion sites containing dispersed chunks from the same index record report a hit in the same location, then the search string is contained in the index record at that location. However, since each dispersion site contains less information than the original index site, we can expect to increase the number of false positives.

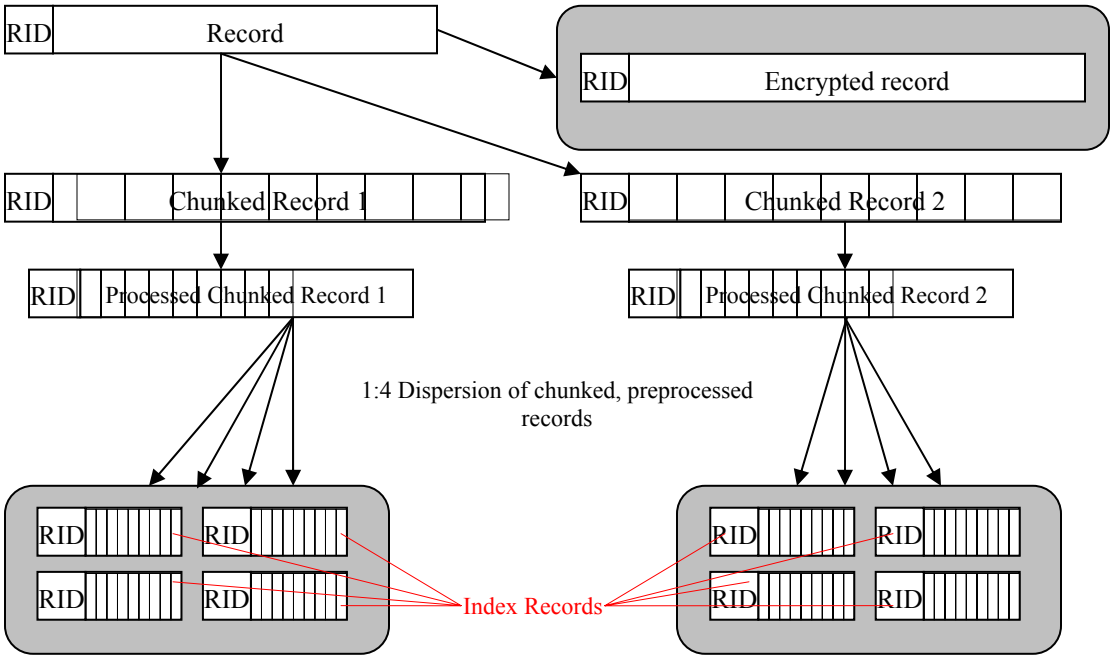

Figure 3: Generation of Records in our scheme: A single record is stored here at 9 different sites. One site, the record store site, contains the record in strongly encrypted form. The other 8 sites, the record index sites, contain the result of dispersing the preprocessed chunks of the records, generated from two different chunkings. The number of chunkings and the ratio of dispersion are application specific parameters.

5. The Complete Scheme

Figure 3 gives an overview of the record storage in our scheme. A given record is identified by a unique RID that we presume is not security sensitive. This RID tags all records that we will ultimately store (indicated by the gray boxes.) We maintain one copy of the record in strongly encrypted form (upper right corner). To generate the index records, we use in this example only two different chunkings. Before or after the encryption, we can use lossy compression in order to reduce the size of the index records, but also in order to even out frequencies of encrypted chunks. Finally, we disperse the chunked records over a number of dispersion sites.

In order to search all data fields, we first need to create chunkings of the index string. The number of chunkings depends on the number of record chunkings and the number of symbols in each chunk. For example, if each chunk consists of eight symbols and we use two chunkings of the records, then we need to chunk each index string four times. Next, we use the same encoding to compress the index string chunkings. Finally, we send the compressed index string chunkings to all dispersal sites. These report back their hits. If all dispersion sites belonging to a certain record chunking – enclosed by the same gray rounded boxes in Figure 3 – report a hit at the same offset, then this is reported as a hit to the client. It will depend on the type of data and the size of the system whether it is not advisable to elect a leader among the dispersion sites to filter out false positives. The client eventually receives all hits in the form of a list of RIDs. (In order to avoid a traffic analysis, these lists are not transmitted in the clear, but rather

in a way that hides the size of the list to a snooper.) It then requests the corresponding records from the corresponding record store site.

As an aside, a standard SDDS such as LH* or its high-availability version LH*$_{RS}$ is used to store index records and the records themselves. The keys for the index records are made up of the RID and the chunking identifier and the dispersion site identifier (3 bits in Figure 3) appended as the least significant bits to the RID. In this way, index records belonging to the same original record will be stored in different LH* buckets if the number of LH* buckets is greater than 8.

```
ADRIAN CORTEZ%%%%%%%%%%%%%415-409-0271$$
AFDAHL E%%%%%%%%%%%%%%%%%%415-409-0817$$
AKIMOTO YOSHIMI%%%%%%%%%%%%415-409-0019$$
ALBAREZ G%%%%%%%%%%%%%%%%%%415-409-0788$$
ALEXANDER GINA%%%%%%%%%%%%%415-409-0464$$
ALGAHIEM ALI%%%%%%%%%%%%%%%415-409-0185$$
ALGHAZALY EBREHIM%%%%%%%%%%415-409-0723$$
ARBELAEZ LIBIA MARIA%%%%%%%415-409-0247$$
ARMENANTE MARK A%%%%%%%%%%%415-409-0910$$
```

Figure 4: Extract from SF Phone Directory Database (Numbers changed)

6. Evaluation of the Scheme

Table 1: χ^2-values for the SF Phone Directory

χ^2 (Single Letter)	2,071,885
χ^2 (Doublets)	10,725,271
χ^2 (Triplets)	40,450,503
A	11.1%
E	9.89%
N	8.55%
R	7.55%
I	6.98%
O	6.27%
AN	3.21%
ER	2.33%
AR	2.11%
ON	1.87%
IN	1.71%
CHA	0.69%
MAR	0.64%
SON	0.50%
ONG	0.50%
ANG	0.49%

Ideally, the contents of the dispersed, chunked, and preprocessed index records are indistinguishable from random bits. Knuth's seminal work [K99] discusses a number of statistical tests for randomness, and the work at the National Institute of Standards and Technology (NIST) [R&al01, S99] used similar statistical tests in the context of selecting a code for the AES standard. Weaker, though probably strong enough would be a proof that the information contained in a (dispersed) index record is too small. In general, a letter in an English text contains between 2 and 3 bits of information [S51], thus storing only 2 bits for each byte should be safe. However, things are not so simple. In one of Shannon's experiments [S51], he gave a human subject one hundred characters of an English text and asked them to guess the next character; in this manner he showed that the information contained in a single letter is closer to a single bit. (See [MGR98] for a further study of entropy in English texts.) If we would have to disperse English text at the rate of 1:8, then the resulting index records would contain so little information to hand out a large number of false positives for every search. In addition to these difficulties, our database does not contain text from books written in English, but generic data with presumably very different entropies. Thus, it appears that we need to settle for index records that are provably close to being true random numbers.

7. Preliminary Experimental Results

We use a telephone database as a test case. Because the entries are very short – they only consist of names and are indexed by the telephone number – our example is a very bad case for our scheme. We obtained a telephone directory San Francisco in California. The directory is intended for lookup with a web-browser. We therefore processed the records to give us flat records containing the telephone number as the RID and the name of the subscriber as the RC. We were as yet not able to break the encoding to include address information. The names are in capitalized letters. Because of the heavy presence of Asian names, the frequency distribution of letters is somewhat unusual. Table 1 gives the χ^2-values for single characters, doublets, and triplets in the 282,965 entries large SF White Pages Phone Directory and lists the most common single letters, doublets, and triplets.

We first tested the effects of dispersal alone. Our records use the normal 8b ASCII encoding. We broke the record in chunks of length one and dispersed each record into four dispersion records using our method with a random non-singular matrix. Thus, a dispersion record contained one symbol of

length 2b for each 8b symbol in the original record. We abbreviate the 2b symbols 00, 01, 10, 11 as 0, 1, 2, and 3 respectively. As we can see in Table 2, this particular matrix (nor any other we tested) did not achieve an even distribution in the occurrence of these four symbols. Worse, doublets are not nearly evenly distributed either. However, the decrease in the χ^2-values as compared to Table 2 is encouraging.

Table 2: χ^2-values for the SF Phone Directory after Dispersion.

χ^2 (Single Letter)	178,849
χ^2 (Doublets)	335,796
χ^2 (Triplets)	486,790
0	33.5%
1	26.9%
2	21.8%
3	17.7%
00	6.98%
10	6.27%
01	3.21%
20	2.33%

Table 3: χ^2-values for the SF Phone Directory after Pre-Processing

Chunk Size = 1

# encod.	χ^2 single	χ^2 double	χ^2 triple
2	0.49	81,631.2	185,329.1
4	97.1	166,060	388,997
8	891.3	640,319	1,908,811
16	352,565..8	3,525,940*	12,931,474

Chunk Size = 2

# encod.	χ^2 single	χ^2 double	χ^2 triple
8	0.000,004	14,992	63,778
16	0.000,009	72,530	439,009
32	0.000,148	357,046	3,349,997
64	7,991.6	1,359,177	25,013,149
128	159,190	5,786,120	193,821,977

Chunk Size = 4

# encod.	χ^2 single	χ^2 double	χ^2 triple
16	0.000,06	5,621	16,177
32	0.000,3	23,696	90,309
64	0.001	92,139	585,739
128	0.005	355,383	4,099,420

Chunk Size = 6

# encod.	χ^2 single	χ^2 double	χ^2 triple
16	0.000,08	1,014	5,076
32	0.000,5	2,802	37,213
64	0.001,7	10,015	292,973
128	0.008	34,400	2,309,653

Next, we tested the effect of redundancy removal alone. We used chunks of size 1, 2, 4 and 8. We did not test larger chunk sizes (8 should give much better values) because of the difficulties in calculating the χ^2-values. In our experiment, we first group all symbols into chunks of size n, n = 1, 2, 4, 6. For example, the entry "LITWIN WITOLD" for n = 4 is transformed into ("LITW" "IN W" "ITOL"). We then assign an encoding – a number between 0 and (2^f-1) – to all possible chunks insuring that each encoding number occurs with about the same frequency. If the number of possible chunks is small compared to the number of encodings, then we did not succeed in equal distribution. Otherwise, we are quite successful. For example, if we use a chunk size of 4 and 128 possible encodings, then the χ^2 – value over the complete database is only 0.005. Clearly, some chunks follow others with much higher frequency then others. If the first chunk is "SMIT", then chances are that the next chunk will start with an "H " and similarly, "MILL" is a good predictor for "ER ". Not unexpectedly, we need larger chunk sizes in order to have less inter-chunk predictability. Our complete results in Table 3 show our relative success and failure.

Next, we tested for false positives when searching. Because exhaustive searches, we extracted 1000 random records from the file. We then searched for the 1000 last names in this subset of the database. In a first experiment, we encoded individual symbols (not chunks) with 8, 16, and 32 possible encodings. For example, the actual entry:

"ABOGADO ALEJANDRO & CATHERINE" encoded in 8 encodings (0-7, see Figure 5), yields:

"10661260172413246060316524532".

We then chunked all the data. For example, with chunk size 2, the record becomes now

"[10],[66],[12],[60],[17],[24],[13],[24],[60],[60],[31][65][24][53]".

"[06],[61],[26],[01],[72],[41],[32],[46],[06],[03],[16][52][45][32]".

In the first chunking, we deleted the last, incomplete chunk, in the second one, we deleted the first incomplete chunk.

Obviously, false positives can already result from this preprocessing. Since the letter "B" and the letter "V" are encoded both by "0", a search for "AVOGADO" would result in a hit for the record "ABOGADO", though in this case, by chance our encoding removes a common spelling error in Spanish where "B" and "V" sound the same. In addition, we chunked the records. Recall that chunking is another source of false positives. In addition to this source of false positives, chunking

itself can create additional false positives. For example searching for "ADAMS" might yield a false hit in a record with the fictional last name "DAMSTER". However, we did not count the occurrence of "ADAMS" in "ADAMSON" as a false positive, since the string "occurs". Of course, a search with leading and trailing space prevents that particular error.

Symbol	Quantity	Encoding
space	503	0
A	495	1
E	407	2
N	383	3
R	350	4
I	300	5
O	287	6
L	258	7
S	258	7
T	200	6
H	186	5
M	178	4
C	159	3
D	150	2
U	112	5
G	108	6
Y	97	1
B	87	0
K	74	7
J	72	4
P	71	3
F	59	2
W	49	7
V	45	0
Z	29	1
&	14	6
X	6	5
Q	5	4
‘	1	5
-	1	5

Figure 5: Encoding Assignment for 8 possible encodings

The results in Table 4(a) show a large number of false positives. Closer inspection for a subset of 500 records revealed that most of the preprocessing false positives originated from the presence of short names such as "Yu" (97 instances, but because it was a repeated part of a last name, it caused 485 false positives), "Ou" (93 instances of false positives), "Ip" (90 instances), "Ba" (87 instances), "Wu" (80 instances), "Li" (54 instances), and "Le" (54 instances), which would indicate that the Warsaw phonebook might have been a better choice for our database. The false positives due to chunking were also caused by short names: "Woo" (125), "Kay" (119), "KIM" (115), "Lee" (112), "See" (112) "Mai" (111), "LIM" (108), "Mak" (103), "Lew" (102). Our results are tabulated in Table 4. As we can see, for our small chunk size of 2, chunking itself created most of the false positives (the difference between FP2 and FP1 in Table 4). In fact, searches for short strings amount to *almost all* false positives, as is demonstrated by Table 4b.

Table 4: False Positives after symbol encoding (FP1) and after symbol encoding and chunking with chunk size = 2 (1000 records)

(a) All entries

En	χ^2 single	χ^2 double	χ^2 triple	FP1	FP2
8	1.49	1,800.7	6,064.9	6,253	18,838
16	1,175	12,450	48,185	911	6,490
32	11,759	64,665	363,535	0	4,669

(b) Entries with Names Longer than 5 Characters

En	χ^2 single	χ^2 double	χ^2 triple	FP1	FP2
8	2.46	1,137.4	3,757.6	24	41
16	651.0	7,391.2	31,582	1	13
32	7,388.2	41,115	247,621	0	11

Finally, we made a similar experiment on the same dataset, but this time, we encoded two-symbol chunks into n = 8, 16, 32, and 64 possible codes. For example, "ABOGADO ALEJANDRO & CATHERINE" creates chunks "[AB], [OG], [AD], [O_], [AL], …" and "[BO] [GA] [DO], [_A] [LE], …". We then collect all these chunks and encode them in one of n different ways such that each code is taken approximately equally often. In this experiment, it turned out that chunking did not create any additional false positives; therefore there is only one column in Table 5, which contains our results. Again, most false positives are caused by searches for small names (v. the difference between the FP column in Table 5a and Table 5b.) Both experiments show clearly the trade-off between seeming randomness and false positives. We notice that n possible encodings in Table 4 correspond to $2n$ possible encodings in Table 5. Thus, the last line in Table 5 describes compressing 2 8bit ASCII

characters into 6 bits, the same rate as the last line in Table 4.

We can draw some conclusions from our preliminary work. Redundancy removal works, but needs to be supplemented by dispersal, which we have not yet tested. In order to prevent false positives from overwhelming true positives, we cannot be too aggressive when removing redundancy. It appears that as a next step, we need to concentrate on large chunks (five or six characters and dispersion should be secure enough for our phone-directory case, but still allow searching), and modest preprocessing.

Table 5: False Positives after chunk encoding (FP1) and after symbol encoding and chunking with chunk size = 2

(a) All entries

Enc	χ^2 single	χ^2 double	χ^2 triple	FP
8	0.002	99.7	713.2	31,648
16	0.009	447.7	5,498	15,588
32	0.039	2,069.5	42,377	7,968
64	20.1	8,129.0	337,306.5	3,857

(b) Entries with last names longer than 5 characters

Enc	χ^2 single	χ^2 double	χ^2 triple	FP
8	.004	73.9	645.4	859
16	.013	344.4	4,835.5	96
32	.030	1,643.3	40,052.0	13
64	13.6	7,066.7	319,355.3	2

8. Conclusions and Further Work

Our proposal tries to reconcile two seemingly irreconcilable desires: to protect data while allowing arbitrary string searches. We make inroads by restricting the attack model and splitting our data in strongly encrypted records and index records used only for searching. We create the index records using an ECB, but use lossy compression and dispersion in order to make the result look more like random strings. If we were to make them look like random streams, then the standard attack tool against ECB, frequency analysis, cannot work. However, even codes that do not reach this goal might be quite secure, but it is very difficult to assess the strength of the code. Our results indicate that redundancy removal is indeed successful in letting the index records appear to be more random. However, the results do (not yet?) justify more than cautious optimism.

At this point, it would appear that a chunk size of 6 ASCII characters together with dispersing index records into 3 records might already result in a reasonable secure code. Analyzing the resulting encoding is quite difficult, since we have to deal with $8^6 = 2^{24}$ possible chunks (assuming ASCII data) and hence deriving the double χ^2-value is challenging. Currently, we are investigating the impact of dispersion. We will also need to investigate other types of lossy compression of the index records that still allow for searches (see [GN99], [M97]).

Currently, we are starting to use the work of Soto [S99] in order to evaluate closeness to randomness in a better manner. Simultaneously, we are pursuing searchable compression as a main mean of redundancy removal. In contrast to the work reported in [GN99] and [M97], our task is simpler, since the compression can be (and probably should be) lossy. We only need very good, but not perfect precision 100% recall and 100% recall for search results.

Finally, Song's *et al.* method of encrypting while allowing for word searches should be adapted to our system.

Acknowledgment

We like to thank Jim Gray and Microsoft Research (Bay Area Research Center, San Francisco) for their generous financial and moral support. We would also like to thank Ethan Miller (University of California at Santa Cruz) for his advice and helpful discussions regarding a first version of the scheme. Indeed, use of ECB on large chunks is his proposal. Finally, we would like to thank the anonymous referees for useful feedback and challenges.

References

[A&al02] A. Adya, W. Bolosky, M. Castro, G. Cermak, R Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer: FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, December 2002.

[AKSX04] R Agrawal, J Kiernan, R Srikant, Y Xu: Order-Preserving Encryption for Numeric Data, In Proceedings of SIGMOD 2004.

[BP82] H. Beker and F. Piper, *Cipher Systems*, Wiley-Interscience, 1982.

[BDET00] W. Bolosky, J. Douceur, D. Ely, and M. Theimer: Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs, Proceedings of the International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS) 2000, pp. 34-43.

[CERIA] SDDS bibliography of Centre de Recherche en Informatique Appliqueé, University of Paris, Dauphine, ceria.dauphine.fr / SDDS-bibliography. html.

[CGKS95] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan, Private Information Retrieval, 36th FOCS, pp.41-50, 1995.

[DW01] J. Douceur and R. Wattenhofer: Optimizing File Availability in a Secure Serverless Distributed File System, Proceedings of 20th IEEE Symposium on Reliable Systems (SRDS), 2001, pp. 4-13.

[GN99] G. Navarro and M. Raffinot: A General Practical Approach to Pattern Matching over Ziv-Lempel Compressed Text. Lecture Notes in Computer Science, Volume 1645, Jan 1999, Page 14.

[K98] D. Knuth: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3^{d} edition, Addison-Wesley, Reading, Mass. (1981)

[LMS05] W. Litwin, R. Moussa, T. Schwarz, LH*RS – A Highly-Available Scalable Distributed Data Structure, Transactions on Database Systems (TODS). Vol. 30(3). September 2005.

[LNS96] W. Litwin, M-A. Neimat, and D. Schneider: LH*: A Scalable Distributed Data Structure. ACM Transactions on Database Systems ACM-TODS, (Dec. 1996).

[M97] U. Manber: A text compression scheme that allows fst searching directly in the compressed file. ACM Transactions on Information Systems, 15(2), p. 124-136, 1997.

[MGR98] H. Moradi, J. Grzymala-Busse, J. Roberts: Entropy of English text: Experiments with humans and a machine learning system based on rough sets. Information Sciences, vol. 104, 1998, p. 31-47.

[Ra89] M. Rabin: Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance, Journal of the ACM (JACM), vol. 36(2), April 1989, p. 335-348.

[RAD78] R. Rivest, L. Adleman, M. Dertouzous: On data banks and Privacy Homomorphisms, in R. De Millo et al., eds., Foundations of Secure Computation, Academic Press, New York, 1978, p. 169-179.

[R&al01] A. Rukhin, J. Soto, J. Nechvatal. M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo: A Statistical Test Suite for Random and Pesudorandom Number Generators for Cryptographic Applications, NIST Special Publication 800-22, May 15, 2001. http://csrc.nist.gov/rng/SP800-22b.pdf

[S51] C. Shannon: Prediction and Entropy of Printed English. The Bell System Technical Journal, January 1951, p. 5̄056.

[SWP00] D. X. Song, D. Wagner, and A. Perrig: Practical Techniques for Searches on Encrypted Data. IEEE Symposium on Security and Privacy, Oakland, California, 2000.

[S99] J. Soto: Randomness Testing of the Advanced Encryption Standard Candidate Algorithms, Technical Report, National Institute of Standards and Technology. http://csrc.nist.gov/rng/rng5.html.