Searchable Encryption through Dispersion

Carlos Aiello Montandón
Universidad Católica del Uruguay
Montevideo, Uruguay
C@it.com.uy

Luis Vidal Introini
Universidad Católica del Uruguay
Montevideo, Uruguay
LuisVidalIntroini@gmail.com

Thomas J. E. Schwarz, S.J.
Universidad Católica del Uruguay
Montevideo, Uruguay
TSchwarz@ucu.edu.uy

***Abstract*—Cryptography is the universal tool to protect the privacy of data. Today's cryptography still requires encrypted data to be decrypted before it can be searched. We propose here an alternative way of protecting the privacy of data through dispersion of a compressed version of the original data that can be searched without recovering the original data. Our scheme compresses the original data and then generates several chunks that are stored at different nodes. The chunks are stored in the form of an index. To search for a string, we convert the string into chunks with the same scheme and then have each site consult its index in order to obtain a list of all possible positions where the search string might be found. These local results are then sent to the user who performs a logical intersection to find all likely positions in the original, where the search string might be located in the text. The user can then decrypt only those parts or records to obtain all parts or records where the search string is.**

Our scheme has no false negatives (all occurrences of the search string will be found). We show that the precision becomes close to 100% for longer strings using a corpus consisting of texts in the English language. We also show that the chunks are somewhat, but not quite similar to random bit streams, and that each individual stream has less information content than a typical English language stream of the same length.

I. Introduction

Cryptography is a universal tool to protect the confidentiality of data. Unfortunately, with today's best cryptographical schemes, processing encrypted data requires decryption before processing. Homomorphic encryption holds the promise of directly processing data in its encrypted forms, including searches, but despite great progress in the last years, [8], [25], especially through the work of Gentry who presented the first *fully* homomorphic encryption scheme [9], [10], it appears that homomorphic encryption light-weight enough for practical applications (other than voting schemes) is still far in the future.

The current state of the art uses symmetric or asymmetric encryption to allow searches for keywords and protects data providing provable, high security.

In this article, we evaluate dispersion as an alternative method for searching in protected data. We see two different approaches. The first is to search directly in the protected data. Using dispersion, the original text is broken into chunks and a search for a string recovers fragments of the chunks that are then reassembled into fragments of the original text likely to contain the search string. Unfortunately, this method has the drawback of creating large amounts of data to be sent from the nodes to the user. Instead, we propose a search structure that works like an index, but for all possible substrings. In this second method, the search gives the user a list of possible locations of the substring and the user then decodes the blocks or records of the encrypted original containing these locations. This second method, which we investigate here, trades traffic reductions for higher storage use.

original text

```
Cryptography is a universal tool to
```

```
Site 0  01100101001100100010011110100111001
Site 1  00001010001001100011110010101001010
Site 2  11110111111101001011100101000110001
```

Pattern	Site 0	Site 1	Site 2
0000	[]	[0]	[]
0001	[15]	[1,7,15]	[26,31]
0010	[3,12,16	[2,8,22,29]	[14,21]
0011	[8,19,27]	[11,16]	[27]
0100	[6,13,17,25]	[5,9,27]	[12,24]
0101	[4]	[3,23,30]	[15,22]
0110	[]	[12]	[28]
0111	[20,28]	[17]	[4,17]
1000	[14]	[6,14]	[25,30]
1001	[2,7,11,18,26,31]	[10,21,28]	[13,20]
1010	[5,24	[4,24,31]	[11,23]
1011	[]	[]	[3,16]
1100	[1,10,30]	[13,20]	[19,29]
1101	[23]	[]	[2,10]
1110	[22,29]	[19]	[1,9,18]
1111	[21]	[18]	[0,5]

Fig. 1: Processing Example with index key length of only four

In the reminder of this article, we first present our scheme, evaluate its costs and safety, and discuss related and future work.

II. Architecture

Our scheme creates a distributed search structure from the unencrypted data. The original data is stored in encrypted form elsewhere. The goal of our application is to provide the client quickly with a list on all the positions where a given search string in the original text is located. While there are no false negatives (each position of the searched for string in the text is in the returned list) there is a usually small possibility for false negatives. The cost of a false negative is the decryption of the block that contained the erroneous position.

While the idea of such a structure is very general, its details depend on the type of data. However, the same is true for searches within the data. For example searches within an image (not of a set of images) will depend heavily on the format of the image, and the same is true for maps, movies, *etc*.

In the following, we present a version optimized for records in a natural language. Our scheme uses lossy compression. This shrinks the size of the text by half, but about doubles the information content of a single character. Each letter in the original alphabet is put into a bin and the letter is encoded by the bin number in the compressed version. The next step takes each character in the compressed text and distributes the bin number into various *chunk streams*. The number of streams is k with $k = 3$ and each bit of a chunk stream corresponds to one character in the original and compressed text. The number of bins therefore is 2^k. If we were to use text in a different alphabet or use data in a different domain such as audio data, the number of streams could and should be chosen differently. We will later discuss a version where we double the number of streams to cut the information contained within a single stream.

Searching for a text using these string of bits turned out to be time-consuming because of the byte-oriented nature of modern computing. In our first C-implementation, we did not want to store a chunk stream as an array of `char`, but then had to use too many shift and `and` operations for searches to be fast. We therefore use a well-known technique from information retrieval to build a complete search index for "shingles" (substrings of fixed length l). The search index uses all 2^l combinations of all possible shingles, i.e. l bits binary strings, as vocabulary and indexes all the occurrences. In practice, the list of postings (lists of occurrences) can be compressed so that the overall storage need is about that of the original text.

A. Binning

The information content of a single symbol in an English language text has long been of interest [24], not in the least because of the need for good compression. The best well-established compression techniques achieve a compression to 1.78 bits per character [2], whereas the best actual methods can use as little as only 1.269 bits per original character in the compressed text [15], even though they do not achieve this performance under all circumstances. This observation justifies assuming that in ordinary English written text, the information content of a single letter is larger than 1 bit. In our proposal, a chunk stream stores a single bit per original character. An adversary who obtains access to a single chunk should therefore not be capable of reconstructing the original text as (s)he lacks sufficient information. An adversary with additional information might still of course be able to draw conclusions on the original text.

Any selection of binning will try to obtain a distribution of zeroes and one in the bit chunk stream that appears to be indistinguishable from a random sequence. The precision of searches will depend on the specificity of the string for which we search. We therefore try to equalize the number of symbols that fall within a bin as much as possible.

Each corpus has its own frequency distribution for its symbols, mainly, but not completely depending on the language used. Since we chose to evaluate English texts, we use a

TABLE I: Frequency table in percent for English letters and a possible binning with 8 bins

Bin	Symbols	Frequency of Symbols	Fr. Bin
0	empty, K	12.17 + 0.41	12.58
1	E, B	11.36 + 1.05	12.41
2	T, M, F	8.03 + 2.76 + 1.79	12.58
3	S, D, L, V	5.68 + 2.92 + 2.92 + 0.97	12.49
4	A, C, P, W, J	6.09 + 2.84 + 1.95 + 1.38 + 0.24	12.50
5	O, R, Y, Q	6.00 + 4.95 + 1.3 + 0.24	12.49
6	N, I, G, X	5.44 + 5.44 + 1.38 + 0.24	12.50
7	others, H, U, Z	6.57 + 3.41 + 2.43 + 0.03	12.44

frequency table for English letters given by Lee [13] that has 26 letters, a white space, and an other category. By hand, we developed the binning given in Table I.

Languages other than English might benefit from preprocessing. For example, a text in Spanish might consider "ñ", "ch", "ll", and "rr" as different letters, placed possibly in different bins than the non-accented or single letter. If the text were in German, it makes sense to consider the *umlauts* and the "sch", "ch", "ei" and "ai" combinations different letters.

B. Creation of chunk streams

Each character belongs to a bin. We encode the character by the bin number (in binary). In the example of Figure 1, the text "Cryptography ..." is split first into chunk streams. The first letter "C" is in Bin 4 or binary 100 and therefore the first bit of the chunk stream for Site 0 is 1, and the ones for Site 1 and Site 2 is 0. The second letter "R" is in Bin 5, so the second bit is 1 in Site 0 and 2 and 0 in Site 1.

By simply permuting the bin numbers among the bins, we can create different encoding, The one presented in Table 1 tries to distribute zeroes and ones equally for each chunk stream.

C. Creation of Index

In order to facilitate searches, we use at each site an index structure that uses bit strings of length $l = 8$ (or $l = 10$) as indices (from 0000 0000 to 1111 1111) and associates to them a list of postings to obtain a classic structure used in information retrieval [16].

We give an example of the structure in Figure 1, but for obvious reasons, we did not want to depict indices with 2^8 or 2^{10} entries, so that we used an otherwise too tiny length of 4 for the bit strings that make up the vocabulary in Figure 1. The bit string 1111 appears in the chunk list of Site 0 beginning in first and fifth position, therefore the corresponding list of postings is [0, 5].

In general, the average difference between two subsequent occurrences of a bit string of length l is 2^l. Each letter in the original text corresponds to one posting for each chunk stream unless the letter is at a distance $< l$ from the end of the record. Our choice of an index implies that we use more storage space, but gain in the speed of processing searches. Besides, the information retrieval community has developed efficient ways of compressing the list of postings. Assume that simple compression leads to using x bytes per posting. Good compression schemes use from less than 4 to about 5

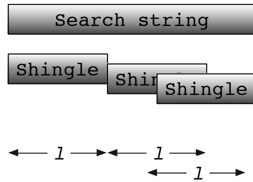

Fig. 2: Shingling of a large search string into groups of l symbols.

bits per posting, giving us a value of x of around 0.5 [16]. Since compression benefits from having not large differences in the value of one posting to the next, and since the chunk streams have some similarity to random bit strings, we can orientate ourselves more towards the lower end for the number of bits stored per posting. Each site stores one posting for every character in the text. The keys (0000 0000 to 1111 1111 or 0000 0000 00 to 1111 1111 11 for shingle size 10) do not need to be stored explicitly but are an offset into the list of postings lists. Thus, each site stores x times the size of the original file. If alternatively we only store the chunk streams themselves, each site stores 1/8$^{\text{th}}$ of the original file written in ASCII.

D. Performing searches

To perform a search for a substring, we first use the binning to encode the search string in exactly the same manner in which we obtained the chunk streams to obtain k binary search strings for each chunk stream. We then divide each binary search string into shingles – groups of l contiguous bits – and use the index to find positions in each site, where the string is located. In more detail, if $b_0, b_1, b_2, \ldots, b_{n-2}, b_n$ is a binary search string, then the first shingle is $b_0, b_1, \ldots, b_{l-1}$, the second $b_l, b_{l+1}, \ldots b_{2l-1}$ *etc.* If l does not divide n, then we have a last shingle $b_{n-l-1}, b_{n-l}, \ldots, b_n$ that partially overlaps the previous one as depicted in Figure 2. We then use the lists of postings at a site to obtain the positions of the binary search string within the chunk stream. If the first shingle of the binary search string starts at position x, then the second shingle (unless it overlaps with the first) starts at position $x+l$, the third shingle (unless it is the last one and overlaps with the second) starts at position $x+2l$ *etc.* For the last shingle, the offset to x needs to be adjusted. This is done by going through the index in a similar manner in which the "and" operation works in information retrieval. All sites send the list of the positions of the binary search string to the client. There, we form the intersection of these lists. The intersection only contains positions where the compressed version of the search string occurs in the compressed version of the original.

If the search string is smaller than l, we can append wild card characters to create a single shingle of size l. We create the binary search string and then generate all binary strings that have the binary search string as a prefix. We then combine the postings of all these strings as the local result. These results can have a high rate of false positives, but the "and" operation at the client will remove most of these false positives.

E. Example searches

Assume that we want to search for the substring "a univers" in the example of Figure 1. The search string is first lossily compressed by the bin numbers, giving "407663153". The binary search string at Site 0 "001001111", which is shingled as "0010", "0111" and "1111". In our small example, the first shingle is found at positions 3, 12, and 16. If one of them is a true position of the binary search string, then the second shingle would be found at position 7, 16, and 20, respectively. Since the second shingle is only at position 20, only 16 is remains possible for the location of the binary search string. Since the third shingle starts at one character removed from the second shingle, "1111" has to be at position 21, which is indeed the case. Therefore, according to Site 0, the search string can possibly only occur in position 16. Since the results from the other two sites also contain this value, this is the only position that possibly can contain the search string.

Assume now that we want to search for the substring "too" in the text of Figure 1. The lossy compression yields search string "255". At Site 0, we look for the binary search string "011", which we complete to "011*". Accordingly, we look for the positions of "0110" and "0111". This gives list [20,28]. At Site 1, we look for the binary search string "100". Accordingly, we combine the postings of "1000" and "1001", yielding [6, 10, 14, 21, 28]. At Site 2, we look for "011", giving [4, 17, 28]. The lists are intersected at the client, giving the only possible position of 28, which is indeed a correct position.

F. Wildcard searches

There are two main types of wildcard searches, one where the wildcard represents a single, unknown character (which is trivially to implement in our system) and one where the wildcard represents a known or unknown number of characters. It is possible to even use these type of searches if the determined parts of the search string make up completely or almost shingles. We leave a detailed description to future work, though the idea is similar to the one used by Litwin and colleagues [14].

III. Enhanced Security Version

As each chunk stream has one bit per symbol in the original text, its information content lies below, but close to the information content of English text. We can increase security by only storing half a bit in a chunk stream for each symbol. We simply break each chunk stream into two by storing the even bits at one site and the odd bits at another site. The resulting increase in security is paid for by more complex searches.

A. Dividing chunk streams

If we have a chunk stream $b_0, b_1, b_2, \ldots$, we define two smaller chunk streams, an even chunk stream $b_0, b_2, b_4 \ldots$ and an odd chunk stream $b_1, b_3, b_5, \ldots$. The even and the odd chunk stream are stored at two different sites. In order to allow fast processing, we create as before a postings index that takes bit strings of length l as vocabulary and associates to them the

Original 1001000111011000101010001011100011101010

Site A 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1

Site B 0 1 0 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0

Site A		Site B	
000	1	000	6,7,8,9,10,17
001	2	001	11,14
010	3,5	010	0,12,15
011	7,11,15	011	2,
100	0	100	5,13,16
101	4,6,10,14	101	1,
110	9,13,	110	4,
111	8,12,16,17	111	3,

Fig. 3: Breaking a chunk stream into an even and an odd part.

occurrences of the vocabulary bit string in the smaller (even or odd) chunk stream. The value of l needs to be half the previous value for l in order to maintain the same search precision as in the basic variant.

B. *Defining searches*

After breaking all chunk streams into even and odd components, we perform a search using shingles of length $2l$. A given search string shingle is first converted into the three chunk streams, and then is divided into an even and an odd half. The indices at the even and the odd site are consulted for the postings, which we then translate into positions into the original data. If the search string is to be found at an even position $2p$ in the text, then the even part of the search string occurs at position p in the even stream and the odd part at the same position p in the odd chunk stream. If the search string is to be found at an odd position $2p+1$ in the text, then the odd part of the search string begins at position p in the even chunk stream and the even part of the search string at position $p+1$ in the odd chunk stream.

C. *Example*

Assume that we want to find "100010" in the chunk stream in Figure 3. We break the search string into even and odd parts to obtain "101" and "000" respectively. The index for Site A gives positions 4, 6, 10, and 14 for the even part in the even chunk stream. The index for Site B gives positions 6, 7, 8, 9, 10, and 17 for the odd part. We can conclude that the search string is to be found at offset 12 in the original chunk stream. Since "000" only appears at position 1 in the even part and "101" at position 1 in the odd part, there are no further occurrences.

If we search for "000101" in Figure 3, we also break the search string into even and odd parts to obtain "000" and "011" respectively. If we look for the even part at Site A and the odd part at Site B, we get the two incompatible lists 1 and 2 corresponding to positions 2 and 5 in the original text. If we look for the odd part of the search string at Site A and for the even part at Site B, we get 7, 11, 15 and 6, 7, 8, 9, 10, 17, respectively. These correspond to positions 14, 22, 30 and 13, 15, 17, 19, 21, 35. Therefore, we have an occurrence of the original search string starting at offset 13 in the original chunk stream.

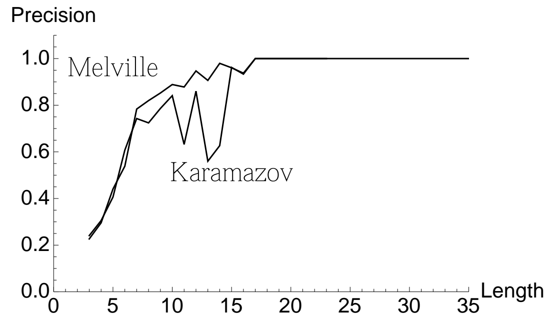

Fig. 4: Precision of 1000 random word searches in the two samples.

TABLE II: Frequency of ones

Chunk Stream	Frequency	Chunk Stream	Frequency
Melville		Dostoyevsky	
0	46.62%	0	48.49%
1	47.58%	1	49.08%
2	47.35%	2	49.49%

IV. Evaluation

For our evaluations, we choose two novels from the Project Gutenberg, Herman Melville's Moby Dick and Garnett's translation of Fyodor Dostoyevsky's The Brothers Karamazov.

A. *Precision*

We first measure the precision of searches for longer search strings. The precision is defined as the proportion of actual positions of the search string over the number of reported positions. A precision of 1 is therefore ideal: each reported occurrence is indeed a true occurrence. We treated the lines of the novels as records. We generated a list of 1000 random words in the novel and compared the number of true occurrences with the number of occurrences in the compressed text to obtain the proportion of times that a record supposedly with a hit does into contain one. The result is given in Figure 4. Basically, if the word size is more than about 10, then the precision is reasonably high, and becomes 1 if the word is long enough.

We then investigated the precision at individual chunk streams. This time, we selected randomly 1000 substrings of a given length and tabulated the number of times that these substrings appeared, the number of times that the compressed text had the compressed version of the substring, and the number of times that a chunk stream had a corresponding occurrence. We then calculated the four precision values for each substring length, one for the precision of searches in the compressed text and three for the precision of searches in the bit streams. Figure 5 gives the result. First, we notice that precision at the chunk streams behaves as is to be expected, starting to become high when we look for substrings of

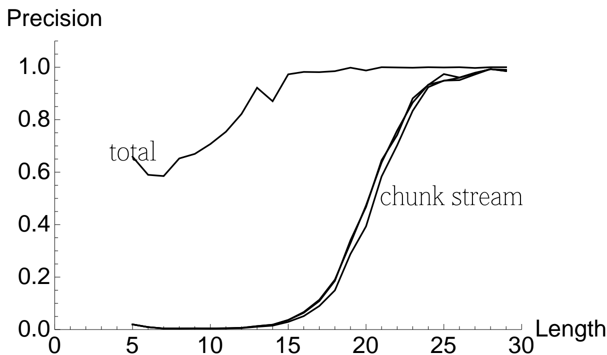

Fig. 5: Precision searching for 1000 random existing substring searches in Melville.

TABLE III: Frequency of blocks of 2 bits in a chunk stream

Dostoyevsky						
	Stream 0		Stream 1		Stream 2	
Block	Obs	Theor	Obs	Theor	Obs	Theor
00	26.89%	26.53%	23.91%	25.93%	24.93%	25.51%
01	24.24%	24.98%	26.73%	24.99%	25.50%	25.00%
10	23.47%	24.98%	26.18%	24.99%	24.90%	25.00%
11	25.41%	23.52%	23.18%	24.09%	24.66%	24.49%
Melville						
	Stream 0		Stream 1		Stream 2	
Block	Obs	Theor	Obs	Theor	Obs	Theor
00	27.79%	27.48%	25.29%	27.48%	27.10%	27.72%
01	25.04%	24.94%	26.61%	24.94%	25.23%	24.93%
10	24.00%	24.94%	25.85%	24.94%	24.52%	24.93%
11	23.17%	22.64%	22.24%	22.64%	23.15%	22.42%

about 20 letters, since for bit strings of this size, a random coincidence becomes quite small given the overall size of the corpus. Second, we can see that the behavior of the chunk streams is not quite identical, in our case, chunk stream 3 almost always had lower precision. This indicates that the chunk streams are different from random bit stream in a statistically significant way.

B. Apparent randomness

A first test for randomness is the distributions of zeroes and ones in the chunk streams. As Table II shows, the number of ones is less than expected. This is a consequence that the frequency of letters in the two corpus is not the same as given in Lee [13]. We developed an alternative binning for the Melville corpus, but the distribution could not be made even because the white spaces make up more than 1/8th of all letters. In itself, a slight discrepancy from a 50-50 distribution of zeroes and ones in a string is not an argument against considering the string essentially random. Much more important is the possibility to predict the next bit given a sequence of various bits. If we measure the frequency of 2-bit blocks (Table III), we see that the distribution is significantly different from one predicted by assuming independence and the observed proportion of ones and zeroes. Clearly, a chunk stream cannot serve as a stand-in for a random number generator, but the differences are not dramatic.

If we pass to the even / odd chunk streams, we find that the apparent randomness of the chunk streams has not improved

TABLE IV: Frequency of blocks of 2 bits in an even / odd chunk stream for Melville.

Even						
	Stream 0		Stream 1		Stream 2	
Block	Obs	Theor	Obs	Theor	Obs	Theor
00	26.39%	29.06%	26.29%	27.84%	24.79%	28.08%
01	27.52%	24.85%	26.47%	24.92%	28.19%	24.91%
10	26.30%	24.85%	25.91%	24.92%	28.18%	24.91%
11	19.79%	21.25%	21.33%	22.32%	18.18%	22.10%
Odd						
	Stream 0		Stream 1		Stream 2	
Block	Obs	Theor	Obs	Theor	Obs	Theor
00	26.22%	28.59%	26.29%	27.62%	24.15%	27.36%
01	27.25%	24.88%	26.26%	24.94%	28.16%	24.95%
10	26.67%	24.88%	26.26%	24.94%	28.60%	24.95%
11	19.86%	21.65%	21.63%	22.52%	19.09%	22.74%

by dividing them. Table IV clearly shows this just using blocks of two bits. The total number of characters processed (which is slightly different from the total number of characters in the text because of how we treat end of lines) is 581618 and 571672 respectively.

V. Related Work

Rabin proposed in 1989 his Information Dispersal Algorithm (IDA) to distribute a file amongst several nodes in a distributed system [18]. It turns out that IDA is based on a erasure correcting code [17]. Krawczyk was the first to consider the problem of validation of the shards into which a file is split [12]. This problem is related to secret splitting [23], [4]. Since Krawczyk's work, numerous schemes provide for checking of remote data [1], [7], [20], [22]. Since IDA is based on a linear transformation (in the sense of an algebraic homomorphism), it is possible to search in IDA chunks [19]. However, IDA and similar schemes either use encryption of the shards into which they split a file or provide no security.

Searching in encrypted data is a well-established important problem. Any fully homomorphic encryption scheme provides this capability. Even though the recent, important advances in finding fully homomorphic encryption schemes [25], [8], [9], [10], usable homomorphic encryption is still far in the future.

Song's *et al.* proposed to encrypt every word of a document independently to allow for efficient keyword searches and a number of authors have provided better schemes for keyword searches. Goh uses a structure based on Bloom filters [11] and Chang and Mitzenmacher use a similar index that is secure with a stronger definition [6]. More recent improvements are those by Boneh *et al.* using asymmetric cryptography [5], Bellare *et al.* [3] and v. Liesdonk *et al.* [26], where the main effort is preventing traffic analysis. To our best knowledge, only our previous work attempts to allow searches not based on key-words [21].

A main difference between these recent works and dispersion is the grade of security provided. Dispersal uses a security based on the statistical impossibility of reconstruction complete data contents from a single site and this is simply not secure enough for many applications. However, the costs of cryptography (for example of storing keys securely) can be

high and some data only needs protection at the "confidential" security level, which is what dispersal can provide.

VI. Future Work

The greatest drawback to the work presented here is the lack of evaluation to other data domains such as audio data. For example, an anti-mafia police surveillance might generate thousands of hours worth of data from phone calls, but would need to process the information fresh if new clues would arise. A typical search then might be for all mentions of a certain gangster monicker in all tapped phone calls during a certain time period. This example shows the need for additional search capabilities, since representing the phone call as a string of phonemes will not be a completely accurate process. One only has to think about local accents (such as the pronunciation "nucelar" for "nuclear" in Texan) and background noise as sources of inaccuracies in the presentation.

Easy porting of our method to another data domain requires the automatic generation of the lossy compression scheme and a bin number assignment that minimizes the differences between a chunk stream and a random bit string. The scheme needs to generate chunk streams that lack information to reconstruct the original stream of symbols (the phonemes in our example).

VII. Conclusion

We have presented a method to disperse text based information into several chunk streams in a manner that allows arbitrary searches in the dispersed text. The information content of each chunk stream is too low for an adversary with only access to a single chunk stream to reconstruct the content. We evaluated the statistical properties of chunk streams using English language novels. In our test corpus, the search shows excellent precision if the search string is longer than about 10 letters. For larger arbitrary substrings, the precision at a single chunk stream becomes high if the substring is longer than 20 letters. We implemented a version in C that showed that searches in bit strings are too time consuming. We therefore advocate the present scheme that replaces the chunk stream with a complete index of the chunk stream. We also evaluated the security of the scheme. As was to be expected, after lossy compression and dispersion, the resulting chunk stream still shows a certain structure that make it easily distinguishable from a random bit stream. This is also evidences that on the average existing substring search, only few false positives are generated at the sides. To alleviate the security concerns, we propose to divide each of the three chunk streams into halves. The scheme is ready to be ported to other type of data of which audio recordings of human speech seems to be the most interesting.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 1, p. 12, 2011.

[2] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 557–591, 1989.

[3] M. Bellare, A. Boldyreva, and A. ONeill, "Deterministic and efficiently searchable encryption," in *Advances in Cryptology-CRYPTO 2007*. Springer, 2007, pp. 535–552.

[4] J. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *Proceedings on Advances in cryptology*. Springer-Verlag New York, Inc., 1990, pp. 27–35.

[5] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith III, "Public key encryption that allows pir queries," in *Advances in Cryptology-CRYPTO 2007*. Springer, 2007, pp. 50–67.

[6] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Applied Cryptography and Network Security*. Springer, 2005, pp. 442–455.

[7] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE, 2008, pp. 411–420.

[8] C. Fontaine and F. Galand, "A survey of homomorphic encryption for nonspecialists," *EURASIP J. Inf. Secur.*, vol. 2007, pp. 15:1–15:15, January 2007.

[9] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*, ser. STOC '09, 2009, pp. 169–178.

[10] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, ser. EUROCRYPT'11, 2011, pp. 129–148.

[11] E.-J. Goh *et al.*, "Secure indexes." *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[12] H. Krawczyk, "Distributed fingerprints and secure information dispersal," in *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '93, 1993, pp. 207–218.

[13] E. S. Lee. (1999) Essays about computer security. Centre for Communications Systems Research Cambridge, Cambridge. [Online]. Available: http://www.proselex.net/Documents/Essaay about Computer Security.pdf

[14] W. Litwin, R. Mokadem, P. Rigaux, and T. Schwarz, "Fast ngram-based string search over data encoded using algebraic signatures," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 207–218.

[15] M. Mahoney. Large text compression benchmark. Accessed: 2014-06-19. [Online]. Available: http://mattmahoney.net/dc/text.html

[16] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[17] F. Preparata, "Holographic dispersal and recovery of information," *Information Theory, IEEE Transactions on*, vol. 35, no. 5, pp. 1123–1124, 1989.

[18] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989.

[19] ——, "The information dispersal algorithm and its applications," in *Sequences*. Springer, 1990, pp. 406–419.

[20] T. Schwarz and E. L. Miller, "Store, forget, and check: Using algebraic signatures to check remotely administered storage," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE, 2006, pp. 12–12.

[21] T. Schwarz, P. Tsui, and W. Litwin, "An encrypted, content searchable scalable distributed data structure," in *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW '06)*, 2006, p. 18.

[22] M. A. Shah, M. Baker, J. C. Mogul, R. Swaminathan *et al.*, "Auditing to keep online storage services honest." in *Proceedings, Eight workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[23] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[24] C. E. Shannon, "Prediction and entropy of printed english," *Bell system technical journal*, vol. 30, no. 1, pp. 50–64, 1951.

[25] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proceedings of the 29st Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, 2010.

[26] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker, "Computationally efficient searchable symmetric encryption," in *Secure Data Management*. Springer, 2010, pp. 87–100.