2006

New Constructions and Practical Applications for Private Stream Searching (Extended Abstract)

John Bethencourt
Carnegie Mellon University

Dawn Song
Carnegie Mellon University

Brent Waters
SRI International

New Constructions and Practical Applications for Private Stream Searching (Extended Abstract)

John Bethencourt
Carnegie Mellon University
bethenco@cs.cmu.edu

Dawn Song
Carnegie Mellon University
dawnsong@cmu.edu

Brent Waters
SRI International
bwaters@csl.sri.com

Abstract

A system for private stream searching allows a client to retrieve documents matching some search criteria from a remote server while the server evaluating the request remains provably oblivious to the search criteria. In this extended abstract, we give a high level outline of a new scheme for this problem and an experimental analysis of its scalability. The new scheme is highly efficient in practice. We demonstrate the practical applicability of the scheme by considering its performance in the demanding scenario of providing a privacy preserving version of the Google News Alerts service.

1 Introduction

Sources of information on the Internet include conventional websites, time sensitive web pages such as news articles and blog posts, newsgroup posts, online auctions, and web based forums or classified ads. To make use of these resources we need search mechanisms that distill the information relevant to each user. Normally, such mechanisms require the user to provide a server with a query such as a textual keyword that the server will compare against the documents in some large data set. This model becomes problematic for applications in which the user would like to hide the search criteria. A user might want to protect the privacy of his search queries for a variety of reasons, including protection of commercial interests and personal privacy. Such privacy issues were brought into the spotlight in 2005 when the U.S. Department of Justice subpoenaed records of search terms from popular web search engines.

Trivially, search privacy may be obtained by downloading the entire remote resource to the client machine and performing the search locally. However, this is typically infeasible due to the large size of the data to be searched, the limited bandwidth between the client and a remote entity, or the unwillingness of a remote entity to disclose the entire resource to the client.

Many of the listed information sources may be considered streams of documents which are being continually generated and processed one-by-one by remote servers. In these cases, it would be advantageous to allow clients to establish persistent searches with the servers, where the data can be efficiently processed. Content matching the search criteria can then be returned to the clients. For example, the Google News Alerts system [1] emails users whenever web news articles crawled by Google match their registered search keywords.

In this extended abstract, we present initial results on an efficient new cryptographic system which allows services of this type while maintaining the secrecy of the search criteria. The new scheme improves on both the asymptotic complexity and practical performance of the previous best solution, making realistic applications feasible. In Section 4, we demonstrate this through the concrete example of Google News Alerts, providing a description of how to apply our scheme in that context along with an analysis based on actual application data. A full treatment of the proposed scheme including detailed algorithms, complexity and correctness analysis, security proofs, and several extensions is now available in a technical report [2].

Related Work There are several problems related to private searching, including searching on encrypted data (in this case the data is encrypted and the query is unencrypted) [3, 4], single-database private information retrieval (PIR) [5, 6], and oblivious transfer [7, 8, 9], with the most closely related problem being PIR. The recent work of Ostrovsky and Skeith [10] was the first to directly address the private searching problem as defined above and only requires communication dependent on the number of matching documents (unlike previous PIR schemes). A drawback of their scheme is that it has steep resource requirements that limit its practical application for many of the scenarios described above. Additionally, their scheme a the keywords of each query to be selected from a public, unencrypted dictionary. In many applications, including a user's search keywords in the public dictio-

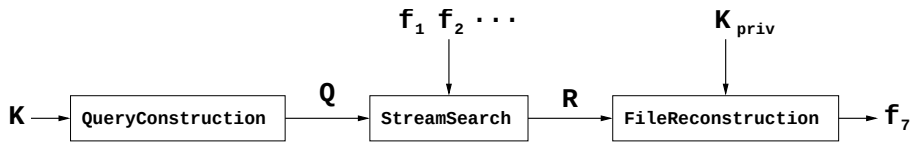

Figure 1. Model for the private searching problem.

nary will already reveal too much information about the client's interests.

2 Problem Definition

In this section we review the problem of private searching. In a private searching scheme a client will create an encrypted query for the set of keywords that he is interested in. The client will give this encrypted query to the server. The server will then run a search algorithm on a stream of files while keeping an encrypted buffer storing information about files for which there is a keyword match. The encrypted buffer will then be returned to the client (periodically) to enable the client to reconstruct the files that have matched his query keywords. We call a file a *matching file* if it matches at least one keyword in the set of keywords that the client is interested in. The key aspect of a private searching scheme is that a server is capable of conducting the search even though it does not know which set of keywords the client is interested in. We now formally describe a private search scheme. A scheme for private stream search consists of the following three algorithms; their relationship is depicted in Figure 1.

`QueryConstruction` $(\lambda, \epsilon, m, K)$: This algorithm is run by a client to prepare an encrypted list of keywords that he would like the server to search for. The algorithm takes as input a security parameter λ, a correctness parameter ϵ, an upper bound on the number of files expected to match m, and an unencrypted set of strings K that are to be used as the search keywords. The algorithm outputs a public key K_{pub}, a private key K_{priv}, and an encrypted query Q. The client then sends K_{pub}, Q to the server. The correctness parameter ϵ may be used to select various algorithm parameters to ensure that up to m files will be correctly retrieved with high probability. These additional parameters are also sent to the server.

`StreamSearch` $(K_{pub}, Q, f_1, \ldots, f_t, W_1, \ldots, W_t)$: This algorithm is run by a server to perform a private keyword search on behalf of the client on a stream of files. The algorithm takes as input an encrypted query Q, a public key, K_{pub}, and a stream of files $f_1, f_2, \ldots, f_t$ and corresponding sets of keywords that describe each file $W_1, W_2, \ldots, W_t$. For each $i \in \{1, \ldots, t\}$, the set W_i is normally derived from the corresponding file f_i as a preprocessing step. The algorithm produces a buffer of encrypted results R, which is sent back to the client. Note that each pair (f_i, W_i) is processed independently when it becomes available, updating the buffer R with the results so far. It is not necessary to have the entire stream available before beginning processing.

`FileReconstruction` (K_{priv}, R): This is used to extract the set of matching files from the returned encrypted buffer. The algorithm takes as input the private key K_{priv} and a buffer of encrypted results R. It outputs the set of matching files $\{ f_i \mid |K \cap W_i| > 0 \}$.

3 New Construction (Outline)

Here we provide an outline of the proposed construction; for the full description refer to [2]. The implementation is built around the homomorphism of the Paillier cryptosystem [11, 12], namely, the fact that for any plaintexts a, b, it is the case that $D(E(a) \cdot E(b)) = a + b$, where E denotes encryption and D denotes decryption. That is, multiplying ciphertexts has the effect of adding the corresponding plaintexts. Our system could use any other semantically secure, asymmetric, additively homomorphic cryptosystem, but for concreteness we consider the use of Paillier in our performance analysis.

First we describe the `QueryConstruction` algorithm, which takes a set of strings K as search keywords and produces an encrypted query Q. First, the client produces an array of ciphertexts $Q = (q_1, q_2, \ldots, q_{\ell_Q})$ of length ℓ_Q initialized to encryptions of zero $E(0)$. Then for each $w \in K$, we replace $q_{h(w)}$ with $E(1)$, where $h : \{0,1\}^* \to \{1, \ldots, \ell_Q\}$ is a hash function used to map each keyword to a location in the array Q.

Now the server may use Q to process its file stream according to the `StreamSearch` algorithm. To process the file f_i containing keywords W_i, the server computes $\prod_{w \in W_i} q_{h(w)} = E(c)$, where c is defined to be $|W_i \cap K|$. The server then computes the modular exponentiation $E(c)^{f_i} = E(cf_i)$. Note that if the file does not match the query $c = 0$, then $E(cf_i)$ is an encryption of zero. We consider the possibility of "spurious matches" when $c \neq 0$ for a non-matching file due to hash collisions

in Section 4.1. In order to accumulate files matching the query, the server keeps a results buffer which is an array of ciphertexts, all initialized to encryptions of zero. The value $E(cf_i)$ is multiplied into random locations in the array, effectively adding the value cf_i to the plaintext already stored in each of the locations. If the file does not match, this does not affect the contents of the buffer. In this way, the locations in the buffer accumulate linear combinations of matching files. The server also uses the value $E(c)$ to update two small auxiliary encrypted buffers containing metadata about which files have matched. This process is described in [2] and omitted from this high level overview. Upon completing a period of searching, the server returns the main data buffer and the auxiliary buffers to the client.

Possessing the private key, the client is able to use the `FileReconstruction` algorithm to reconstruct the files that matched their query. First, they decrypt the main data buffer to obtain the plaintext of each entry, which is a linear combination of some of the matching files. With the help of some information from the two metadata buffers, the client is then able to establish a system of linear equations which may be be solved for the content of the matching files. If the number of files that matched the query exceeds the number of places in the main data buffer, however, the buffer has "overflowed" and the files cannot be recovered. Thus the user must establish an upper bound m on the number of files they expect to match and specify the size of the main data buffer accordingly, perhaps allowing some extra space if the number of files which will match is uncertain. This is done in the `QueryConstruction` algorithm, with the client passing the desired parameters to the server for use in `StreamSearch`.

Apart from the possibility of too many files matching the query, there are a couple of other scenarios in which the files may be unrecoverable. In [2] we give a detailed analysis of these cases, demonstrating that their probability diminishes exponentially with linear increases to the buffer size and giving upper bounds on the buffer size necessary to bound the failure probability below some ϵ. In practice the new system requires near minimal overhead to achieve a high probability of success. This is in contrast to the scheme of Ostrovsky and Skeith, which is quite demanding; this is considered in Section 4. One additional difference between our proposed scheme and that of Ostrovsky and Skeith is the absence of a predetermined keyword dictionary D. In most situations, providing a dictionary of all keywords one could possibly be searching for is a serious security limitation. Many of the strings a user may want to search for are obscure (e.g., names of particular people or other proper nouns) and including them in D would already reveal too much information. Since the size of encrypted queries is proportional to $|D|$, it is not feasible to fill D with, say, every person's name, much less all proper nouns.

r	s_q	optimized s_q
0.1	1.3 MB	0.3 MB
0.01	13.1 MB	3.6 MB
0.001	132.8 MB	36.6 MB

Table 1. Size of the encrypted query necessary to achieve a given spurious match rate before and after optimizations.

4 Practical Performance Analysis

We now consider the case of making a private version of Google's News Alerts service [1] using the new construction. According to the Google News website, their web crawlers continuously monitor approximately 4,500 news websites. These include major news portals such as CNN along with many websites of newspapers, local television stations, and magazines. In this setting, we analyze four aspects of the resources necessary for a private search: the size of the query sent to the server (s_q), the size of the storage buffers kept by the server while running the search and eventually transmitted back to the client (s_b), the time for the server to process a single file in its stream (t_p), and the time for the client to decrypt and recover the original matching files from the information he receives from the server (t_r).

4.1 Query Space

If we assume a 1024-bit Paillier key, then the encrypted query Q is $256\ell_Q$ bytes, since each element from the set of ciphertexts $\mathbb{Z}^*_{n^2}$ is $\frac{\lceil \log_2 n \rceil}{4}$ bytes, where n is the public modulus. The smaller ℓ_Q is, the more files will spuriously match the query. Specifically, we obtain the following formula for the the probability r that a non-matching file f_i will nevertheless result in a non-zero corresponding $E(c)$ (rearranged on the right to solve for ℓ_Q).

$$r = 1 - \left(1 - \frac{|K|}{\ell_Q}\right)^{|W_i|} \qquad \ell_Q = \frac{|K|}{1 - (1-r)^{\frac{1}{|W_i|}}}$$

We performed a sampling of the news articles linked by Google News and found that the average distinct word count is about 540 per article. This produces the false positive rates for several query sizes listed in Table 1. The first column specifies a rate of spurious matches r and the second column gives the size s_q of the minimal Q necessary to achieve that rate for a single keyword search. Additional keywords increase s_q

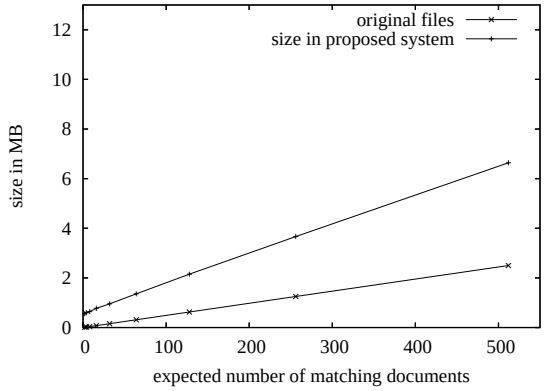

Figure 2. Server to client communication after a period of searching in the new system.

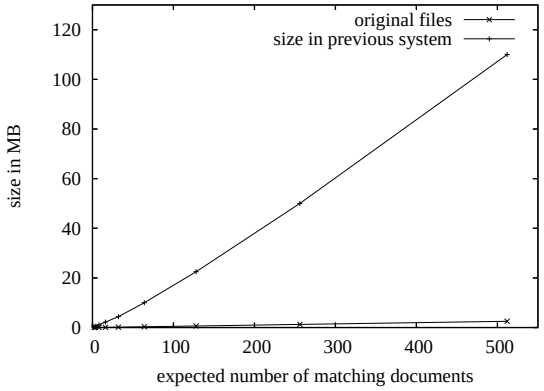

Figure 3. Server to client communication after a period of searching in Ostrovsky-Skeith.

proportionally (e.g., $|K| = 2$ would double the value of s_q). It should be apparent that this is a significant cost; in fact, it turns out that s_q is the most significant component in the total resource usage of the system under typical circumstances.

Two measures may be taken to reduce this cost. First, note that the majority of distinct words occurring in the text of a news article are common English words that are not likely to be useful search terms. Given this observation, the client may specify that the server should ignore the most commonly occurring words when processing each file. A cursory review of the 3000 most common English words (based on data from the British National Corpus [13]) confirms that none are likely to be useful search terms. Ignoring those words reduces the average distinct word count in a news article to about 200.

The second consideration in reducing s_q is that a smaller Paillier key may be acceptable. While 1024 bits is generally accepted to be the minimum public modulus secure for a moderate time frame (e.g., as required by the standards ANSI X9.30, X9.31, X9.42, and X9.44 and FIPS 186-2) [14], it is important to realize that a compromise of the Paillier key would not immediately result in the revelation of K. Instead, it would allow the adversary to mount a dictionary attack, checking potential members of K against Q. Since a string not in K that is checked against Q will match anyway with probability $\frac{|K|}{\ell_Q}$, an attacker may also need some prior knowledge about potential search terms if they are to gain useful information about K. Without any such knowledge, checking a very large set of potential words against Q would result in too many false positives to be useful. Given this consideration, if the client decides a smaller key length is acceptable, s_q will be reduced. The third column in Table 1 gives the size of the encrypted query using a 768-bit key and pruning out the 3000 most common English words from those searched.

Despite the significant cost of s_q in our system, the cost to obtain a comparable level of security is likely much greater in the system of Ostrovsky and Skeith. In that case $s_q = 256|D|$, where $|D|$ is the set of all possible keywords that could be searched. In order to reasonably hide $K \subseteq D$, $|D|$ may have to be quite large. For example, if we wish to include names of persons in K, in order to keep them sufficiently hidden we must include many names with them in D. If D consists of names from the U.S. population alone, s_q will be over 70 GB.

4.2 Storage Buffers Space

We now turn to the size of the buffers maintained by the server during the search and then sent back to the client. This cost, s_b, is both a storage requirement of the server conducting the search and a communication requirement at the end of the search. Let the length of the main data buffer maintained by the server be ℓ_D, the length of the two metadata buffers (taken together) be ℓ_M, and the length of a file be ℓ_f (measured as the required number of plaintexts from $\mathbb{Z}_n$). The server then stores $s_b = 256\,\ell_D\ell_f + 256\,\ell_M$ bytes (using a 1024-bit key).

The client will specify ℓ_D and ℓ_M based on the number of documents they expect their search to match in one period and the desired correctness guarantees. In the case of Google news, we may estimate that each of the 4,500 crawled news sources produces an average of 30 articles per day [1]. If the client retrieves the current search results four times per day, then the number of files processed in each period is $t = 33,750$. Now the client cannot know ahead of time how many articles will match their query, so they must make an estimate of m. Based on this estimate, they may try increasing ℓ_D and ℓ_M until the probability of an over-

	time to multiply	time to exponentiate
768-bit key	3.9 μs	6.2 ms
1024-bit key	6.3 μs	14.7 ms

Table 2. Benchmarks for arithmetic in $\mathbb{Z}_{n^2}^*$.

flow is bounded by some small ϵ. A detailed analysis demonstrating values of ℓ_D and ℓ_M that achieve this appears in [2]. In this extended abstract we instead consider empirical results on the necessary ℓ_D and ℓ_M in a particular application.

A range of desired values of m were considered and the results are displayed in Figure 2. In each case, ℓ_D and ℓ_M were selected so that the probability of an overflow is less than 0.01. Also, the spurious match rate r was taken to be 0.001, and the news articles were considered to be 5 KB in size (text only, compressed). Note that s_b is linear with respect to the size of the matching files. More specifically, the data displayed in Figure 2 reveals that it is about 2.4 times the size of the matching files. For comparison, the space stored by the server and returned to the client using the Ostrovsky-Skeith scheme for private searching in this scenario is shown in Figure 3.[1] Note that the graph differs in scale from Figure 2.

To summarize, in the proposed system s_b ranges from about 564 KB to about 6.63 MB when the expected number of matching files ranges from 2 to 512 and the overflow rate is held below 0.01. In the Ostrovsky-Skeith scheme, s_b would range from about 282 KB to 110 MB.

4.3 File Stream Processing Time

Next we consider the time t_p necessary for the server to process each file in its stream. This is essentially determined by the time necessary for modular multiplications in $\mathbb{Z}_{n^2}^*$ and modular exponentiations in $\mathbb{Z}_{n^2}^*$ with exponents in $\mathbb{Z}_n$. To roughly estimate these times, benchmarks were run on a modern workstation. The processor was a 64-bit, 3.2 Ghz Pentium 4. We used the GNU Multiple Precision Arithmetic Library (GMP), a library for arbitrary precision arithmetic that is suitable for cryptographic applications. The results are given in Table 2.

The first step carried out for in processing the ith file in the `StreamSearch` procedure is computing $E(c)$; this takes $|W_i| - 1$ multiplications. We again use

[1]The paper describing this system did not explicitly state a minimum buffer length for a given number of files expected to be retrieved and a desired probability of success, but instead gave a loose upper bound on the length. Rather than using the bound, we ran a series of simulations to determine exactly how small the buffer could be made while maintaining an overflow rate below 0.05.

m	t_p with 768-bit key	t_p with 1024-bit key
2	359 ms	600 ms
8	362 ms	600 ms
32	373 ms	603 ms
128	420 ms	617 ms
512	593 ms	669 ms

Table 3. The time necessary for the server to process a file.

$|W_i| = 540$ as described in Section 4.1. Computing $E(cf_i)$ requires ℓ_f modular exponentiations. The next step is updating the main data buffer and metadata buffers with these values. Although we have not given the details of this algorithm, it requires approximately $\frac{\ell_D \ell_f}{2}$ modular multiplications. The time necessary for both these steps is given for several values of m in Table 3. The majority of t_p is due to the ℓ_f modular exponentiations. Since the Ostrovsky-Skeith scheme requires the same number of modular exponentiations, the processing time for each file would be similar.

4.4 File Recovery Time

Finally, we consider the time necessary for the client to recover the original matching files after a period of searching, t_r. This time is composed of the time to decrypt the returned buffers and the time to setup and solve a system of linear equations, producing the matching documents. A decryption requires 1536 modular multiplications with a 1024-bit key and 1152 with a 768-bit key [12]. The times necessary to decrypt the buffers are given in Table 4. This time is typically less than a minute, but can take as long as five with many files.

The time to setup and solve the system of linear equations is dominated by the time necessary to invert two matrices of size ℓ_D. Inverting an $n \times n$ matrix through Gaussian elimination requires $\frac{1}{3}n^3 + n^2 - \frac{1}{3}n$ multiplications; this could likely be improved with sparse matrix techniques. As shown in Table 5, the time for matrix inversions is small for all but the largest cases.

Although the time spent in matrix inversions is a significant additional cost of the new scheme over Ostrovsky-Skeith, it is more than offset by the reduced buffer size. In Ostrovsky-Skeith, the times to decrypt the buffer returned to the client in this scenario range from 6.79 seconds for $m = 2$ to 45.5 minutes for $m = 512$, using a 768-bit key. With a 1024-bit key, the buffer decryption times range from 10.8 seconds to 1.21 hours.

m	decryption time with 768-bit key	decryption time with 1024-bit key
2	14s	23s
8	15s	26s
32	23s	38s
128	51s	1.4m
512	2.7m	4.4m

Table 4. Time (in seconds and minutes) necessary to decrypt the buffers.

m	inversion time with 768-bit key	inversion time with 1024-bit key
2	0.1 s	0.2 s
8	0.2 s	0.3 s
32	0.8 s	1.3 s
128	0.2 m	0.3 m
512	7.1 m	11.5 m

Table 5. Time (in seconds and minutes) necessary to invert the matrices.

5 Conclusion

Our system for private stream searching allows a range of applications not previously practical. In particular, we have considered the case of conducting a private search on essentially all news articles on the web as they are generated, estimating this number to be 135,000 articles per day. In order to establish the private search, the client has a one time cost of approximately 10 MB to 100 MB in upload bandwidth, based on various tradeoffs. Several times per day they download about 500 KB to 7 MB of new search results, allowing up to about 500 articles per time interval. After receiving the encrypted results, the client spends under a minute recovering the original files, or up to about 15 minutes if many files were retrieved. This performance would be typical of a desktop PC; a mobile device would be capable of handling a somewhat less demanding scenario. To provide the searching service, the server keeps about 10 MB to 100 MB of storage for the client and spends roughly 500 ms processing each new article it encounters. These costs are comparable to many free services currently available on the web (e.g., email and webhosting), so it is likely the private searching service could be provided for free. With high probability, the client will successfully obtain all articles matching their query, and in any case the server will remain provably oblivious to nature of their search.

Most of the parameters of this scenario (e.g., the number of distinct articles generated per day, the number of distinct words per file, the size of a file, etc.) are probably less than one or two orders of magnitude different than for the other online searching situations mentioned in Section 1 (such as blog posts, USENET, online auctions). We expect our techniques to be applicable to many of these searching applications. The complete algorithms for the private searching scheme are presented along with complexity analysis and formal security proofs in [2].

References

[1] The Google news alerts service. Information available at `http://www.google.com/alerts`.

[2] John Bethencourt, Dawn Song, and Brent Waters. New techniques for private stream searching. Technical Report CMU-CS-06-106, Carnegie Mellon University, March 2006.

[3] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[4] Boneh, Di Crescenzo, Ostrovsky, and Persiano. Public key encryption with keyword search. In *Eurocrypt*, 2004.

[5] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science (FOCS)*, pages 364–373, 1997.

[6] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *Information Security and Privacy (ACISP)*, 2004.

[7] Kaoru Kurosawa and Wakaha Ogata. Oblivious keyword search. *Journal of Complexity*, 20, 2004.

[8] Freedman, Ishai, Pinkas, and Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference (TCC)*, volume 2, 2005.

[9] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. pages 314–328, 2005.

[10] Rafail Ostrovsky and William Skeith. Private searching on streaming data. August 2005.

[11] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, pages 223–238, 1999.

[12] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136, 2001.

[13] The British national corpus. Oxford University Computing Services. Information available at `http://www.natcorp.ox.ac.uk`.

[14] Robert Silverman. A cost-based security analysis of symmetric and asymmetric key lengths. Technical report, RSA Laboratories, November 2001.