New Techniques for Private Stream Searching

John Bethencourt* **Dawn Song*** **Brent Waters†**

February 2006
CMU-CS-06-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* Carnegie Mellon University.
† SRI International.

During this project, John Bethencourt and Dawn Song received support from the NSF and ARO. Brent Waters was supported in part by NSF CNS-0524252.

Keywords: private stream searching, private information retrieval, public key program obfuscation, systems of linear equations, Bloom filters, homomorphic encryption

Abstract

A system for private stream searching, introduced by Ostrovsky and Skeith [18], allows a client to provide an untrusted server with an encrypted search query. The server uses the query on a stream of documents and returns the matching documents to the client while learning nothing about the nature of the query. We present a new scheme for conducting private keyword search on streaming data which requires $O(m)$ server to client communication complexity to return the content of the matching documents, where m is the size of the documents. The required storage on the server conducting the search is also $O(m)$. Our technique requires some metadata to be returned in addition to the documents; for this we present a scheme with $O(m \log(t/m))$ communication and storage complexity. In many streaming applications, the number of matching documents is expected to be a fixed fraction of the stream length; in this case the new scheme has the optimal $O(m)$ overall communication and storage complexity with near optimal constant factors. The previous best scheme for private stream searching was shown to have $O(m \log m)$ communication and storage complexity. In applications where $\frac{t}{m} > m$, we may revert to an alternative method of returning the necessary metadata which has $O(m \log m)$ communication and storage complexity; in this case constant factor improvements over the previous scheme are achieved. Our solution employs a novel construction in which the user reconstructs the matching files by solving a system of linear equations. This allows the matching documents to be stored in a compact buffer rather than relying on redundancies to avoid collisions in the storage buffer as in previous work. We also present a unique encrypted Bloom filter construction which is used to encode the set of matching documents. In this paper we describe our scheme, prove it secure, analyze its asymptotic performance, and describe several extensions.

1 Introduction

The Internet currently has several different types of sources of information. These include conventional websites, time sensitive web pages such as news articles and blog posts, real time public discussions through channels such as IRC, newsgroup posts, online auctions, and web based forums or classified ads. One common link between all of these sources is that searching mechanisms are vital for a user to be able to distill the information relevant to him.

Most search mechanisms involve a client sending a set of search criteria to a server and the server performing the search over some large data set. However, for some applications a client would like to hide his search criteria, i.e., which type of data he is interested in. A client might want to protect the privacy of his search queries for a variety of reasons ranging from personal privacy to protection of commercial interests.

A naive method for allowing private searches is to download the entire resource to the client machine and perform the search locally. 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 to the unwillingness of a remote entity to disclose the entire resource to the client.

In many scenarios the documents to be searched are being continually generated and are already being processed as a stream by remote servers. In this case it would be advantageous to allow clients to establish persistent searches with the servers where they could be efficiently processed. Content matching the searches could then be returned to the clients as it arises. For example, Google News Alerts system [1] emails users whenever web news articles crawled by Google match their registered search keywords. In this paper we develop an efficient cryptographic system which allows services of this type while provably maintaining the secrecy of the search criteria.

Private Stream Searching Recently, Ostrovsky and Skeith defined the problem of "private filtering", which models the situations described above. They gave a scheme based on the homomorphism of the Paillier cryptosystem [19, 9] providing this capability [18]. First, a public dictionary of keywords D is fixed. To construct a query for the disjunction of some keywords $K \subseteq D$, the user produces an array of ciphertexts, one for each $w \in D$. If $w \in K$, a one is encrypted; otherwise a zero is encrypted. A server processing a document in its stream may then compute the product of the query array entries corresponding to the keywords found in the document. This will result in the encryption of some value c, which, by the homomorphism, is non-zero if and only if the document matches the query. The server may then in turn compute $E(c)^f = E(cf)$, where f is the content of the document, obtaining either an encryption of (a multiple of) the document or an encryption of zero.

Ostrovsky and Skeith propose the server keep a large array of ciphertexts as a buffer to accumulate matching documents; each $E(cf)$ value is multiplied into a number of random locations in the buffer. If the document matches the query then c is non-zero and copies of that document will be placed into these random locations; otherwise, $c = 0$ and this step will add an encryption of 0 to each location, having no effect on the corresponding plaintexts. A fundamental property of their solution is that if two different matching documents are ever added to the same buffer location then we will have a collision and both copies will be lost. If all copies of a particular matching document are lost due to collisions then that document is lost, and when the buffer is returned to the client, he will not be able to recover it.

To avoid the loss of data in this approach one must make the buffer sufficiently large so that this event does not happen. This requires that the buffer be much larger than the expected number of required documents. In particular, Ostrovsky and Skeith show that a given probability of successfully obtaining all matching documents may be obtained with a buffer of size $O(m \log m)$,[1] where m is the number of matching documents. While effective, this scheme results in inefficiency due to the fact that a significant portion of the buffer returned to the user consists of empty locations and document collisions.

Our Approach In this paper we present a new private stream searching scheme which achieves the optimal $O(m)$ communication from the server to the client and server storage overhead in returning the content of the matching documents, given any fixed probability of successfully retrieving all matching documents. Metadata required for the reconstruction of the documents is returned using a technique requiring $O(m \log(t/m))$ communication and storage. The latter technique also results in the optimal $O(m)$ complexity with near optimal constant factors in applications where each document matches the query with some probability, independent of the other documents. In applications where a fixed number of documents are expected to match, regardless of the stream length, a modification to our scheme produces the previous $O(m \log m)$ complexity, but with near optimal constant factor overhead. These results are based on the novel combination of a few techniques.

Like the approach of Ostrovsky and Skeith we give an encrypted dictionary and non-matching documents have no effect on the encrypted contents. However, rather than using a large buffer and attempting to avoid collisions, each matching document in our system is copied randomly over approximately half of the locations across the buffer. A pseudo-random function, g, whose key is shared by the client and server, will determine pseudo-randomly with probability $\frac{1}{2}$ whether the document is copied into a given location, where the function takes as inputs the document number (document number i is the ith document seen by the server) and buffer location. While any one particular buffer location will not likely contain any information about any matching document, with high probability all the information from all the matching documents can be retrieved from the whole system by the client given that the client knows the number of matching documents and that the number of matching documents is less than the buffer size. The client can do this by decrypting the buffer and then solving a linear system to retrieve the original documents. Finally, the server maintains a separate encrypted Bloom filter that efficiently keeps track of which document numbers were matched. The use of an efficient Bloom filter to keep track along with our method of storing documents allows us to store the encrypted documents in a much smaller buffer.

[1] Specifically, they define a correctness parameter γ and use a buffer of size $O(\gamma m)$. They show that a given success probability may be achieved with a γ that is $O(\log m)$.

1.1 Related Work

Private searching may be viewed as the flip side of searching on encrypted data [21, 3, 11]; in this case the data is unencrypted and the query is encrypted. Goh applied Bloom filters in a way that allows a server to store encrypted-searchable data in a more efficient manner.

However, searching on encrypted data is quite different from private searching. In the problem of searching on encrypted data the data is hidden from the server, while in private searching the data is known to the server and the client's queries must remain hidden. Private searching is actually most closely related to the topics of single-database private information retrieval [8, 15, 5, 6] and oblivious transfer [17, 16]. One incompatibility between previously proposed PIR schemes and the present problem is that PIR schemes have thus far required communication dependent on the size of the entire database rather than the size of the portion retrieved. In some streaming settings, a private searching scheme with communications independent of the size of the stream or database is desirable. Another difference between the PIR and private search settings is that most PIR constructions model the database to be searched as a long bitstring and the queries as indices of bits to be retrieved. In contrast, the system proposed in this paper and that of Ostrovsky and Skeith allow queries based on a search for keywords within text. Both these schemes may also retrieve pieces of data by index, however. The text associated with a block of data in the database against which queries are matched is arbitrary, so by simply including strings of the form "blocknumber:1", "blocknumber:2", ... in the text associated with each block of data, they may be explicitly retrieved by appropriate queries. There has been some consideration of search or retrieval by keyword rather than index in the PIR literature [7, 14, 10], but none of these systems has communication dependent only on the size of the data retrieved rather than some function of the length of the database or stream. In [2] we experimentally analyzed the performance of our system in the setting of a realistic application, comparing it with the scheme of Ostrovsky and Skeith.

2 Definitions and Preliminaries

In this section we describe the problem of private searching and make appropriate definitions. We also briefly review Paillier's cryptosystem and the definition of a pseudo-random function family.

2.1 Problem Definition

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[2] 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

[2] We use the name "file" as a general term for the data chunk that is to be returned. The type of data will vary by application.

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 stream search scheme. A scheme for private stream search scheme consists of the following three algorithms.

`QueryConstruction` **(λ, ϵ, m, K)** The `QueryConstruction` 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 files to retrieve 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}, $Q, f_1, \ldots, f_t, W_1, \ldots, W_t$)** The `StreamSearch` 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 $\vec{f} = (f_1, f_2, \ldots, f_t)$ and corresponding sets of keywords that describe each file $\vec{W} = (W_1, \ldots, W_t)$. Normally each set W_i is 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 after processing some number of files t, which is chosen by the client. The choice of t is application dependent and should ensure that no more than m matching documents are likely to be found.

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

To define privacy for a private stream search scheme, consider the following game between a challenger and an adversary. The adversary gives the challenger two sets of keyword strings K_0, K_1. The challenger then flips a coin β, runs the `QueryConstruction` ($\lambda, \epsilon, m, K_\beta$), and gives the public key and the encrypted query Q to the adversary. The adversary then outputs a guess β'. We say that an adversary has advantage ϵ if $|\mathrm{P}\,(\beta = \beta') - \frac{1}{2}| \geq \epsilon$

Definition 1. We say that a private searching scheme is semantically secure if for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ is negligible in the security parameter, λ.

We establish that the proposed system satisfies this definition in Section 4.2.

2.2 Preliminaries

Paillier's Cryptosystem We now provide a brief review of the most important features of the Paillier cryptosystem. The Paillier cryptosystem is a public key cryptosystem; as in RSA the public key n is the product of two large primes. The factorization of n is the private key. In this paper the encryption of a plaintext m with the public key (there is only one public key in use in this paper, the one generated by the client when constructing a private search) is denoted $E(m)$, and the decryption of a ciphertext c with the private key is denoted $D(c)$. Plaintexts are represented by elements of the group $\mathbb{Z}_n$ and ciphertexts are represented by elements of the group $\mathbb{Z}_{n^2}^*$, so $E : \mathbb{Z}_n \rightarrow \mathbb{Z}_{n^2}^*$ and $D : \mathbb{Z}_{n^2}^* \rightarrow \mathbb{Z}_n$. Note that ciphertexts are twice as large as plaintexts.[3]

The key property of the Paillier cryptosystem upon which the entire system is based is its homomorphism. For any $a, b \in \mathbb{Z}_n$, it is the case that $D(E(a) \cdot E(b)) = a + b$. That is, multiplying ciphertexts has the effect of adding the corresponding plaintexts. This allows one to perform rudimentary computations on encrypted values. Our construction may be adapted to use any public key, homomorphic cryptosystem, but for concreteness, we assume the use of the Paillier cryptosystem throughout the rest of the paper.

Pseudo-Random Functions In our construction we use a pseudo-random function family $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \rightarrow \{0, 1\}$. Roughly speaking, G will take in a key k and two integers and output a pseudo random bit. We let $g = G_k$ where $k \xleftarrow{R} \mathcal{K}_G$.

The security of a pseudo-random function family $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \rightarrow \{0, 1\}$ is defined by the following game between a challenger and an adversary $\mathcal{A}$. A challenger chooses a random key $k \xleftarrow{R} \mathcal{K}_G$ and lets $g = G_k$. The challenger then flips a binary coin β. At this point the adversary submits to make oracle queries to the challenger over the domain. If $\beta = 0$ the challenger will respond by evaluating the function g on the input, whereas if $\beta = 1$ it will respond with random bit to all new queries, while giving the same response if the same query is asked twice. Finally, the adversary outputs a guess β'. We define the adversary's advantage in this game as:

$$\mathsf{Adv}_{\mathcal{A}} = |\Pr[\beta = \beta'] - 1/2|$$

We say that a pseudo random function is $(\omega_t, \omega_q, \epsilon)$- secure if no ω_t time adversary, that makes at most ω_q oracle queries, has advantage at greater than ϵ.

3 New Construction

We now describe the algorithms of the new private search scheme and give an analysis of complexity and security properties. In the following explanations, we defer discussion of several special failure cases to the next subsection.

[3] This property of inflating messages by encrypting them is improved in Damgård-Jurik generalization of the Paillier cryptosystem [9]. In their scheme the plaintext and ciphertext spaces are $\mathbb{Z}_{n^s}$ and $\mathbb{Z}_{n^{s+1}}^*$ for any $s \in \{1, 2, \ldots\}$. However, the constraints in this paper are likely to make the original situation of $s = 1$ preferable.

Algorithm: `QueryConstruction`
Input: Set of keywords K.
Output: Query array $Q = (E(q_1), E(q_2), \ldots, E(q_{|D|}))$, public key n.

Generate a Paillier key pair n, K_{priv}.

```
for  i := 1, 2, ..., |D| :
    if  w_i ∈ K :
        q_i := 1
    else :
        q_i := 0
    Q[i] := E(q_i)
```

Figure 1: The algorithm for setting up an encrypted query.

3.1 Client's `QueryConstruction` Procedure

Figure 1 gives the algorithm for producing the encrypted query, `QueryConstruction`. A public dictionary of potential keywords

$$D = \{w_1, w_2, \ldots, w_{|D|}\}$$

is assumed to be available. Constructing the encrypted query for some disjunction of keywords $K \subseteq D$ then proceeds as in the scheme of Ostrovsky and Skeith. The client generates a key pair, then for each $i \in 1, \ldots, |D|$, defines $q_i = 1$ if $w_i \in K$ and $q_i = 0$ if $w_i \notin K$. The values $q_1, q_2, \ldots, q_{|D|}$ are encrypted (rerandomizing each encryption) and put in the array $Q = (E(q_1), E(q_2), \ldots, E(q_{|D|}))$, which forms the final encrypted query. In Section 5.2 we give an alternative form for the encrypted queries which eliminates the public dictionary D. The client then sends Q and the public key n to the server.

3.2 Server's `StreamSearch` Procedure

Figure 2 gives the full algorithm run by the server, `StreamSearch`. In addition to the public key and Q, the client may provide the server with the parameter t, the number of files to process before returning the results, and the parameters ℓ_F, ℓ_I, and k, which affect correctness and performance (see below and Section 4.1).

State The server must maintain three buffers as it processes the files in its stream. These buffers are hereafter referred to as the *data buffer,* the *c-buffer,* and the *matching-indices buffer* and denoted F, C, and I respectively. Each of these is an array of elements from the ciphertext space $\mathbb{Z}^*_{n^2}$, with F and C of length ℓ_F and I of length ℓ_I. For simplified notation here and in subsequent explanations, we assume that each document is at most n bits and therefore fits within a single plaintext in $\mathbb{Z}_n$. For longer documents requiring s elements of $\mathbb{Z}_n$, we would let F be an $\ell_F \times s$ array and subsequent operations involving a file updating F are performed blockwise.

Algorithm: `StreamSearch`
Input: Q, n, number of files to process t, sequence of files $f_1, \ldots, f_t$ with corresponding keyword sets $W_1, \ldots W_t$, size of data buffer ℓ_F, size of matching indices buffer ℓ_I, number of hash functions k.
Output: Data buffer F, coefficients buffer C, matching indices buffer I.

Initialize F and C as ℓ_F element arrays and I as an ℓ_I element array of members of $\mathbb{Z}^*_{n^2}$. Initialize each element of F, C, and I to $E(0)$.

```
for i := 1, 2, ..., t :
    c := E(0)
    for w_j ∈ W_i :
        c := c · Q[j] mod n^2

    e := c^{f_i} mod n^2
    for j := 1, 2, ..., ℓ_F :
        if g(i, j) = 1 :
            F[j] := F[j] · e mod n^2
            C[j] := C[j] · c mod n^2

    for j := 1, 2, ..., k :
        ℓ := h_j(i) mod ℓ_I
        I[ℓ] := I[ℓ] · c mod n^2
```

Figure 2: The algorithm for running the private search.

The data buffer will store the matching files in an encrypted form which can then be used by the client to reconstruct the matching files. In particular, the data buffer will contain a system of linear equations in terms of the content of the matching files in an encrypted form. This system of equations will later be solved by the client to obtain the matching files.

The c-buffer stores in an encrypted form the number of keywords matched by each matching file. We call the number of keywords matched for a file the *c-value* of the file. The c-buffer will be used in reconstruction of the matching files from the data buffer by the client. As in the case of the data buffer, the c-buffer stores its information in the form of a system of linear equations. The client will later solve the system of linear equations to reconstruct the c-values.

The matching-indices buffer is an encrypted Bloom filter that keeps track of the indices of matching files in an encrypted form. More precisely, the matching-indices buffer will be a encrypted representation of some set of indices $\{\alpha_1, \ldots, \alpha_r\}$ where $\{\alpha_1, \ldots, \alpha_r\} \subseteq \{1, \ldots, t\}$. Here r is the number of files which end up matching the query.

Each of these buffers begins with all its elements initialized to encryptions of zero. We now detail how they are updated as each file is processed.

Processing Steps To process the ith file f_i, the server takes the following steps.
Step 1: Compute encrypted c-value. First, the server looks up the query array entry $Q[j]$ corresponding to each word w_j found in the file. The product of these entries is then computed. Due to the homomorphic property of the Paillier cryptosystem, this product is an encryption of c-value of the file, i.e., the number of distinct members of K found in the file. That is,

$$\prod_{w_j \in W_i} Q[j] = E\left(\sum\nolimits_{w_j \in W_i} q_j\right) = E\left(c_i\right)$$

where W_i is the set of distinct words in the ith file and c_i is defined to be $|K \cap W_i|$. Note in particular that $c_i \neq 0$ if and only if the file matches the query.

Step 2: Update data buffer. The server computes $E\left(c_i f_i\right)$ using the homomorphic property of the Paillier cryptosystem.

$$E\left(c_i\right)^{f_i} = E\left(c_i f_i\right) = \begin{cases} E\left(c_i f_i\right) & \text{if } f_i \text{ matches the query} \\ E\left(0\right) & \text{otherwise.} \end{cases}$$

The server multiplies the value $E\left(c_i f_i\right)$ into a subset of the locations in the data buffer according to the following procedure. Let G be a family of pseudo-random functions that map $\mathbb{Z} \times \mathbb{Z}$ to $\{0, 1\}$. Randomly select $g \stackrel{R}{\leftarrow} G$ (this should be done once upon initialization and the same g used for all files). The algorithm multiplies $E\left(c_i f_i\right)$ into each location j in the data buffer where $g(i, j) = 1$. Suppose for example we are updating the third location in the data buffer with the second file. Assume that first file was also multiplied into this location, i.e., $g(1, 3) = g(2, 3) = 1$. Each of the two files may or may not match the query. Suppose in this example that f_1 matches the query, but f_2 does not. Before processing f_2 we have that $D\left(F[3]\right) = c_1 f_1$. After multiplying in $E\left(c_2 f_2\right)$, $D\left(F[3]\right) = c_1 f_1 + c_2 f_2$. But $c_2 = 0$ since f_2 does not match, so it is still the case that $D\left(F[3]\right) = c_1 f_1$ and the data buffer is effectively unmodified. This mechanism allows the data buffer to accumulate linear combinations of matching files while discarding all non-matching files.
Step 3: Update c-buffer. The value $E\left(c_i\right)$ is multiplied into each of the locations in the c-buffer in a similar fashion as $E\left(c_i f_i\right)$ was used to update the data buffer. In particular, the server multiplies the value $E\left(c_i\right)$ into each location j in the c-buffer where $g(i, j) = 1$.

Step 4: Update matching-indices buffer. The server then multiplies $E\left(c_i\right)$ further into a fixed number of locations in matching-indices buffer. This is done using essentially the standard procedure for updating a Bloom filter. Specifically, we use k hash functions $h_1, \ldots, h_k$ to select the k locations where $E\left(c_i\right)$ will be added. For optimal efficiency, the client should select the parameter k as $\lfloor \frac{\ell_I \log 2}{m} \rfloor$, where m is the number of files they expect to retrieve [4]. The locations of the matching-indices buffer that a matching file i is multiplied into are take to be $h_1(i), h_2(i), \ldots, h_k(i)$. Again, if the f_i does not match, $c_i = 0$ so the matching-indices buffer is effectively unmodified.

After completing the aforementioned steps for a fixed number of files t in its stream, the server sends its three buffers back to the client. Also, the server should return the function g.

```
Algorithm:  FileReconstruction
    Input:  F, C, I, k.
   Output:  The matching files f_{α'_1}, f_{α'_2}, ..., f_{α'_r}.

Decrypt each element of F, C, and I to obtain F', C', and I'.

    β := 0
    for  i := 1, 2, ..., t :
        for  j := 1, 2, ..., k :
            ℓ := h_j(i) mod ℓ_I
            if  I'[ℓ] = 0 :  next  i
        β := β + 1
        α_β := i

    if  β > ℓ_F :
        output  "Error, overflow.",  exit
    while  β < ℓ_F :
        β := β + 1
        α_β := pick({1, ..., t} \ {α_1, α_2, ..., α_{β-1}})

    A := [ g(α_i, j) ]_{i:=1,2,...,ℓ_F; j:=1,2,...,ℓ_F}

    if  A is singular :
        output  "Error, singular matrix.",  exit
    c⃗ := A^{-1} · C'
    {α'_1, α'_2, ..., α'_r} = {α_1, α_2, ..., α_{ℓ_F}} \ { α_i | c_{α_i} = 0 }
    for  i ∈ { α_i | c_{α_i} = 0 } :
        c_{α_i} := 1

    f⃗ := diag(c⃗)^{-1} · A^{-1} · F'
    output  f_{α'_1}, f_{α'_2}, ..., f_{α'_r}
```

Figure 3: The algorithm for recovering the matching files after the completion of a private search.

3.3 Client's `FileReconstruction` Procedure

Figure 3 gives the algorithm run by the client upon completion of the private search and receipt of the three buffers F, C, and I, `FileReconstruction`.

Step 1: Decrypt buffers. The client first decrypts the values in the three buffers using the Paillier decryption algorithm with its private key K_{priv}, obtaining decrypted buffers F', C', and I'.

Step 2: Reconstruct matching indices. For each of the indices $i \in \{1, 2, \ldots, t\}$, the client computes $h_1(i), h_2(i), \ldots, h_k(i)$ and checks the corresponding locations in the decrypted matching-indices buffer; if all these locations are non-zero, then i is added to the list $\alpha_1, \alpha_2, \ldots, \alpha_\beta$ of potential matching indices. Note that if $c_i \neq 0$, then i will be added to this list. However, due to the false positive feature of Bloom filters, we may obtain some additional indices. Now we may check for overflow, which occurs when the number of false positives plus the number of actual matches r exceeds ℓ_F. At this point if $\beta < \ell_F$, we continue to add indices to the list until it is of length ℓ_F. Here the function pick denotes the operation of selecting an arbitrary member of a set. Note that we will not run out of indices since $t \geq \ell_F$.

Step 3: Reconstruct c-values of matching files. Given our superset of the matching indices $\{\alpha_1, \alpha_2 \ldots, \alpha_{\ell_F}\}$, the client next solves for the values of $c_{\alpha_1}, c_{\alpha_2}, \ldots, c_{\alpha_{\ell_F}}$. This is accomplished by solving the following system of linear equations for $\vec{c}$,

$$A \cdot \vec{c} = C' \tag{1}$$

where A is the matrix with the i,jth entry set to $g(\alpha_i, j)$, C' is the vector of values stored in the decrypted c-buffer, and $\vec{c}$ is the column vector $(c_{\alpha_i})_{i=1,\ldots,\ell_F}$.[4] Now the exact set of matching indices $\{\alpha'_1, \alpha'_2 \ldots, \alpha'_r\}$ may be computed by checking whether $c_{\alpha_i} = 0$ for each $i \in \{1, \ldots, \ell_F\}$. Before proceeding, we replace all zeros in the vector $\vec{c}$ with ones.

As an example of Step 3, suppose there are four spots in the decrypted c-buffer (i.e., $\ell_F = 4$), seven files are processed, and we have established the following list of potentially matching indices: $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} = \{1, 3, 5, 7\}$. Then given

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \quad C' = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 3 \end{pmatrix}$$

we may compute

$$\begin{aligned} c_{\alpha_1} &= c_1 = 1 \\ c_{\alpha_2} &= c_3 = 2 \\ c_{\alpha_3} &= c_5 = 1 \\ c_{\alpha_4} &= c_7 = 0 \quad . \end{aligned}$$

We then see that there were three matching files ($r = 3$): f_1, f_3, and f_5.

Step 4: Reconstruct matching files. Finally, the content of the matching files $f_{\alpha'_1}, f_{\alpha'_2}, \ldots, f_{\alpha'_r}$ may be determined by solving the linear system

$$A \cdot \mathrm{diag}(\vec{c}) \cdot \vec{f} = F' \tag{2}$$

where

$$\mathrm{diag}(\vec{c}) = \begin{pmatrix} c_1 & 0 & \cdots \\ 0 & c_2 & \\ \vdots & & \ddots \end{pmatrix} \quad .$$

We directly compute $\vec{f} = \mathrm{diag}(\vec{c})^{-1} \cdot A^{-1} \cdot F'$. Note that $\mathrm{diag}(\vec{c})$ is never singular because we previously ensured that no zeros appear in $\vec{c}$. The content of the matching files appears as

[4] The possibility of the matrix A being singular is considered in the next section.

$f_{\alpha'_1}, f_{\alpha'_2}, \ldots, f_{\alpha'_r}$; the other entries in $\vec{f}$ will be zero. Continuing the example above (and making up a value of F'), this corresponds to solving the following equations

$$\begin{aligned} f_1 + f_5 &= 32 \\ f_1 + 2f_3 + f_7 &= 32 \\ f_1 + f_7 &= 10 \\ 2f_3 + f_5 &= 44 \;, \end{aligned}$$

thereby determining that $f_1 = 10$, $f_3 = 11$, and $f_5 = 22$ (and $f_7 = 0$, but this value is ignored).

4 Analysis

4.1 Correctness and Complexity

In this section, we give the correctness and complexity analysis of our scheme. In particular, we will show that given a desired success probability bound $1-\epsilon$, if the number of matching documents is at most m, then by using communication and storage overhead $O(m\log(t/m))$, our scheme will enable the user to correctly reconstruct all the matching documents from a stream of t documents with probability at least $1-\epsilon$.

In order to perform the analysis to demonstrate the above point, we first analyze the different failure cases where the user will fail to reconstruct the matching documents. From the reconstruction procedure, we can see that the client fails to reconstruct the matching files when the two systems of linear equations $A \cdot \vec{c} = C'$ (Eq. 1) and $A \cdot \mathrm{diag}(\vec{c}) \cdot \vec{f} = F'$ (Eq. 2) cannot be correctly solved. This failure only happens in two cases:

1. The matrix A is singular. In this case, we will not be able to compute A^{-1} and solve the system of linear equations.
2. There are more than $\ell_F - r$ false positives when the set of matching indices is computed using the Bloom filter. In particular, if in Step 2 in the `FileReconstruction` procedure, the number of matching indices β reconstructed from the Bloom filter I' is greater than ℓ_F, then we have more variables than the number of linear equations and thus we will not be able to solve the system of linear equations $A \cdot \vec{c} = C'$.

We show below that by picking the parameters ℓ_F and ℓ_I correctly, we can guarantee that the probability of the above two failure cases can be bounded to be below ϵ. We demonstrate this by proving the following three lemmas.

Lemma 1. *For a given $0 < \epsilon < 1$, there exists $n = o(\log(1/\epsilon))$, such that for any $n' > n$, an $n' \times n'$ random $(0,1)$-matrix is singular with probability at most ϵ.*

Proof. Note that an $n \times n$, random (0,1)-matrix is singular with negligible probability in n. This was first conjectured by Erdös and proven in the 60's by J. Komlós [13]. The specific bound has since been improved several times, recently reaching $O\left(\left(\frac{3}{4} + o(1)\right)^n\right)$ [12, 22, 23]. Thus, it is easy to see that the above lemma holds. □

Lemma 2. *Let $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \rightarrow \{0,1\}$ be a $(\omega_t, \omega_q, \epsilon/8)$-secure pseudo-random function family. Let $g = G_k$, where $k \xleftarrow{R} \mathcal{K}_G$. Let $\ell_F = o(\log(1/\epsilon))$ such that an $\ell_F \times \ell_F$ random $(0,1)$-matrix is singular with probability at most $\epsilon/4$. Then the matrix*

$$A = \Big[\, g(i,j) \,\Big]_{\substack{i=1,\ldots,\ell_F \\ j=1,\ldots,\ell_F}}$$

is singular with probability at most $\epsilon/2$.

Intuitively, this lemma bounds the failure probability that the matrix A is singular. We provide the proof in Appendix B. Additionally, we note that for a given constant ϵ the size of the ℓ_F will be linear in m.

Lemma 3. *Given $\ell_F > m + 8\ln(2/\epsilon)$, let $\ell_I = O(m\log(t/m))$, and assume the number of matching files is at most m out of a stream of t. Then the probability that the number of reconstructed matching indices β is greater than ℓ_F is at most $\epsilon/2$.*

Given the false positive rate of a Bloom filter, the proof is straightforward; we provide it in Appendix C. Together, Lemma 2 and Lemma 3 provide the primary result:

Theorem 1. *If $\ell_F = o(\log(1/\epsilon)) + O(m)$, $\ell_F > m + 8\ln(2/\epsilon)$, $\ell_I = O(m\log(t/m))$, $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \rightarrow \{0,1\}$ is a $(\omega_t, \omega_q, \epsilon/8)$-secure pseudo-random function family, then when the number of matching files is at most m in a stream of t, our scheme guarantees that the client can correctly reconstruct all matching files with probability at least $1 - \epsilon$.*

Proof. By Lemma 2, the probability that the matrix A is singular is at most $\epsilon/2$. By Lemma 3, the probability that the reconstruction of the matching indices will yield more than ℓ_F matching indices is at most $\epsilon/2$. Since these are the only two failure cases as explained earlier, the total failure probability, the probability that the client would fail to reconstruct the matching files, is at most ϵ. □

4.2 Security

The security of the proposed system according to Definition 1 is straightforward. Intuitively, since the server is only provided with an array of encryptions of ones and zeros, the scheme should be as secure as the underlying cryptosystem.

Theorem 2. *If the Paillier cryptosystem is semantically secure, then the proposed private searching scheme is semantically secure according to Definition 1.*

In Appendix D we provide a proof. The proof is straightforward and proceeds as in the case of Ostrovsky and Skeith. Note that this establishes security based on the decisional composite residuosity assumption, since that was used to prove the security of the Paillier cryptosystem.

5 Extensions

Here we describe several extensions to the proposed system which provide additional features or vary performance tradeoffs.

5.1 Bloom Filter Space Saving

For security it will generally be necessary to use a modulus n of at least 1024 bits (e.g., as required by the standards ANSI X9.30, X9.31, X9.42, and X9.44 and FIPS 186-2) [20]. The fact the c-values will never approach 2^{1024} reveals that the Bloom filter I is in fact mostly wasted space. A simple technique can be used to reclaim some of this space. If we assume that the sums of c-values appearing in each location in I will be less than 2^{16}, for example, we may use each group element to represent $\frac{n}{16}$ array entries. In the case of $n = 1024$, this reduces the size of I by a factor of 64. When we need to multiply a value $E(c)$ into the Bloom filter in the `StreamSearch` algorithm, we use the following technique. To multiply it into the ith location in I, we let $i_1 = \lfloor \frac{i}{64} \rfloor$ and $i_2 = i \bmod 64$. Then we compute

$$I[i_1] := I[i_1] \cdot E(c)^{2^{16 i_2}}$$

which has the result of shifting c into the i_2th 16-bit block within the group element in $I[i_1]$. After the client decrypts I, they may simply break up each element into 64 regions of 16 bits. This space savings comes at an additional computation cost, however. The server will need to perform k additional modular exponentiations for each file it processes.

5.2 Hashing Keywords

In some applications, the predetermined set of possible keywords D may be unacceptable. 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 may not be feasible to fill D with, say, every person's name, much less all proper nouns.

In such applications an alternative form of encrypted query may be used. Eliminating D, we allow K to be any finite subset of Σ^*, where Σ is some alphabet. Now in `QueryConstruction`, we pick a length ℓ_Q for the array Q and initialize each element to $E(0)$. Then for each $w \in K$, we use a hash function $h : \Sigma^* \to \{1, \ldots, \ell_Q\}$ to select a location $h(w)$ in Q and set $Q[h(w)] := E(1)$. As before we rerandomize each encryption. To process the ith file in `StreamSearch`, the server may now compute $E(c_i) = \prod_{w \in W_i} Q[h(w)]$. The rest of the scheme is unmodified. Using this extension, it is possible for a file f_i to spuriously match the query if there is some word $w' \in W_i$ such that $h(w') = h(w)$ for some $w \in K$. The possibility of such false positives is the key disadvantage of this approach.

An advantage of this alternative approach, however, is that it is possible to extend the types of possible queries. Previously only disjunctions of keywords in D were allowed, but in this case a limited sort of conjunction of strings may be achieved. To support queries of the form "w_1 w_2"

where $w_1, w_2 \in \Sigma^*$, we change the way each W_i is derived from the corresponding file f_i. In addition to including each word found in the file f_i, we include all adjacent pairs of words in W_i (note that this approximately doubles the size of W_i). It is easy to imagine further extensions along these lines. In particular, it is possible to match against binary data by simply including blocks of the contents of f_i in W_i.

5.3 Stream Length Independence

In applications where the expected number of matching documents is fixed and independent of the stream length, a modification to the scheme allows communication and storage independent of the stream length as well. To produce this effect, we abandon the Bloom filter based construction used in the matching-indices buffer and instead use the Ostrovsky Skeith construction to store the matching indices. We briefly describe this technique below; for details (including an analysis of collision detection) refer to [18].

Let $\ell_I = \gamma m$, where γ is selected based on the desired error bound ϵ. Fix a set of hash functions $h_1, h_2, \ldots, h_\gamma$. Also, let each entry in the matching-indices buffer I be a pair of ciphertexts in $\mathbb{Z}^*_{n^2}$ rather than a single ciphertext. To update I when processing the ith file in `StreamSearch`, compute the following.

$$
\begin{aligned}
&\texttt{for}\ \ j := 1, 2, \ldots, \gamma\ : \\
&\quad \ell := h_j(i) \bmod \ell_I \\
&\quad I[\ell][1] := I[\ell][1] \cdot c \bmod n^2 \\
&\quad I[\ell][2] := I[\ell][2] \cdot c^i \bmod n^2
\end{aligned}
$$

To recover the set of matching indices in `FileReconstruction`, the client decrypts each pair of entries in I. When a pair $I'[k][1]$ and $I'[k][2]$, $k \in \{1, \ldots \ell_I\}$ is non-zero (and not a collision), the client may recover the index of a matching file as $i = I'[k][2]/I'[k][1]$.

When using this technique, the c-buffer is omitted. We may set $\ell_F = m$; otherwise, the data buffer is used as before. All parameters are now selected based only on m and ϵ without regard to t, and there are no false positives for streams of any length. The analysis in [18] demonstrates that the probability of an overflow in the new matching-indices buffer may be bounded below ϵ with $\gamma = O(\log m + \log(1/\epsilon))$, producing an overall communication and storage complexity of $O(m \log m)$. Note that our scheme still produces a constant factor improvement over the original scheme of Ostrovsky and Skeith in this case. If each file requires s plaintext blocks (i.e., is of length ns bits), then we reduce communication and storage by a factor of approximately s. This is accomplished by retrieving the bulk of the content through the efficient data buffer and only retrieving document indices through the less efficient matching-indices buffer.

5.4 Arbitrary Length Files

In applications where the files are expected to vary significantly in length, an unacceptable amount of space may be wasted by setting an upper bound on the length of the files and padding smaller files to that length. Here we describe a modification to the scheme which eliminates this source of inefficiency by storing each block of a file separately.

In this extension `QueryConstruction` takes two upper bounds on the matching content. We let m_1 be an upper bound on the number of matching files and m_2 be an upper bound on the total length of the matching files, expressed in units of Paillier plaintext blocks. As before, the c-buffer is of length $O(m_1)$ and the matching-indices buffer is of length $O(m_1 \log(t/m_1))$ (or, using the alternative construction given in Section 5.3, $O(m_1 \log m_1)$). The data buffer is now set to length $O(m_2)$, and each entry in the data buffer is now a single ciphertext rather than an array fixed to an upper bound on the length of each file. We introduce a new buffer on the server called the *length buffer,* which is an array L set to length $O(m_1)$. Intuitively, the length buffer will be used to store the length of each matching file, and the data buffer will now be used to store linear combinations of individual blocks from each file rather than entire files.

We briefly describe how this is accomplished in more concrete terms. Replace the corresponding portion of `StreamSearch` with the following, where $\ell_C = O(m_1)$ is the length of the c-buffer and length buffer, $\ell_F = O(m_2)$ is the length of the data buffer, $\hat{g} : \mathbb{Z}^3 \to \{0,1\}$ is an additional pseudo-random function, d_i is the length of the ith file in the stream, and the d_i blocks of the file are denoted $f_{i,1}, f_{i,2}, \ldots, f_{i,d_i}$.

$$
\begin{array}{l}
e := c^{d_i} \bmod n^2 \\
\texttt{for}\ \ j := 1, 2, \ldots, \ell_C\ : \\
\quad \texttt{if}\ \ g(i,j) = 1\ : \\
\qquad C[j] := C[j] \cdot c \bmod n^2 \\
\qquad L[j] := L[j] \cdot e \bmod n^2 \\
\\
\texttt{for}\ \ j_1 := 1, 2, \ldots, d_i\ : \\
\quad e := c^{f_{i,j_1}} \bmod n^2 \\
\quad \texttt{for}\ \ j_2 := 1, 2, \ldots, \ell_F\ : \\
\qquad \texttt{if}\ \ \hat{g}(i, j_1, j_2) = 1\ : \\
\qquad\quad F[j_2] := F[j_2] \cdot e \bmod n^2
\end{array}
$$

The client may use a modified version of `FileReconstruction` to recover the matching files. As before, the matching-indices buffer I is used to determine a superset of the indices of matching files, and a matrix A of length ℓ_C is constructed based on these indices using g. The vector $\vec{c}$ is again computed as $\vec{c} := A^{-1} \cdot C'$. The client next computes the lengths of the matching files as $\vec{d} := \mathrm{diag}(\vec{c})^{-1} \cdot A^{-1} \cdot L'$. If $\sum_i d_i > \ell_F$, the combined length of the files is greater than the prescribed upper bound and the client aborts. Otherwise, the data buffer now stores a system of $\ell_F \geq m_2$ linear equations in terms of the individual blocks of the matching files. Briefly, the blocks may be recovered by constructing a new matrix $\hat{A}$, filling its entries by evaluating $\hat{g}$ over the indices of the blocks of the matching files. The blocks of the matching files are then computed as $\vec{f} := \mathrm{diag}(\vec{c}\,')^{-1} \cdot \hat{A}^{-1} \cdot F'$, where $\vec{c}\,'$ is as $\vec{c}$ but with the ith entry repeated d_i times.

Using this extension, space may be saved if the matching files are expected to vary in size. Some information about the number expected to match and their total size is still needed to set up the query, but the available space may now be distributed arbitrarily amongst the files.

6 Conclusion

The primary contribution of our scheme is the improvement of server storage and server to client communication complexity from $O(m \log m)$ in the size of the matching files to $O(m \log(t/m))$. In the common streaming case of each document matching independently from other documents, this results in the optimal $O(m)$ complexity, with near optimal constant factors. A practical analysis with problem parameters corresponding to a realistic application is given in [2], an extended abstract on the performance of this scheme. It is shown that in a typical scenario with a long stream, it is possible to avoid failure with probability over 0.99 while using communication (and server storage) $1.2m$, where m is the actual size of the matching files, before the factor of two inflation due to the Paillier cryptosystem. In contrast, we found the scheme of Ostrovsky and Skeith to result in storage and communication as high as $24m$ before the inflation due to Paillier. In applications where m is allowed to vary arbitrarily, independent of t, a modified version of our scheme returns to the $O(m \log m)$ communication and storage complexity. In this case constant factor improvements are made over the previous scheme of Ostrovsky and Skeith. Both versions of our scheme achieve the increased efficiency through a novel technique for efficiently spreading the matching documents throughout the buffer of results, the former also employing a unique encrypted Bloom filter construction. Finally, we proved correctness and security results for the scheme and noted some extensions.

References

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

[2] John Bethencourt, Dawn Song, and Brent Waters. New constructions and practical applications for private stream searching (extended abstract). In *IEEE Symposium on Security and Privacy*, May 2006.

[3] Boneh, Di Crescenzo, Ostrovsky, and Persiano. Public key encryption with keyword search. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT*, 2004.

[4] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[5] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT '99*, pages 402–414, 1999.

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

[7] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical report, 1997.

[8] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science FOCS'95*, pages 41–51, October 1995.

[9] 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.

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

[11] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/`.

[12] J. Kahn, J. Komlós, and E. Szemerédi. On the probability that a random ± 1 matrix is singular. *J. Amer. Math. Soc.*, 8(1):223–240, 1995.

[13] J. Komlós. On the determinant of (0,1)-matrices. *Studia Math. Hungarica*, 2:7–21, 1967.

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

[15] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS'97*, pages 364–373, 1997.

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

[17] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 245–254, New York, May 1999. Association for Computing Machinery.

[18] Rafail Ostrovsky and William Skeith. Private searching on streaming data. In *CRYPTO*, August 2005.

[19] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT '99*, pages 223–238, 1999.

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

[21] 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.

[22] Terence Tao and Van H. Vu. On random ± 1 matrices: singularity and determinant. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, pages 431–440, 2005.

[23] Terence Tao and Van H. Vu. On the singularity probability of random bernoulli matrices. Eprint in submission to *J. Amer. Math. Soc.*, 2005.

A Terms and Notation

For easy reference, we provide a single list of the terms and variables introduced and defined throughout the text.

client the person or machine conducting a private search, i.e., generating a private query and eventually recovering the content that matched the query

server the person or machine carrying out the private search on the behalf of the client

n Paillier public key ($n = p_1 p_2$, where p_1 and p_2 are large, secret primes)

s an upper bound on the length of a file as a number of elements from $\mathbb{Z}_n$, i.e., if files are at most b bits, then $s = \lceil \frac{b}{\lceil \log_2 n \rceil} \rceil$

t number of files processed by the server before returning buffers to the client

ρ false positive rate of the Bloom filter I

D global dictionary of potential keywords

K the set of keywords forming the query

w_i the ith word in D

q_i the ith entry in the query array (before encryption), corresponds to w_i

f_i the ith file checked by the server

W_i the words present in or associated with the ith file[5]

c_i the number of distinct keywords matched by the ith file, i.e., $|K \cap W_i|$

m an upper bound on the number of files which may be retrieved

r the number of files which actually match the query

Q the encrypted query, an array of $|D|$ elements from $\mathbb{Z}^*_{n^2}$

F the data buffer, an array of ℓ_F elements, each of which is an array of s elements from $\mathbb{Z}^*_{n^2}$

C the coefficients buffer, an array of ℓ_F elements from $\mathbb{Z}^*_{n^2}$

I the matching indices buffer, an array of ℓ_I elements from $\mathbb{Z}^*_{n^2}$

k the number of hash functions to be used with the matching indices buffer, set to $\lfloor \frac{\ell_I \log 2}{m} \rfloor$

[5]In the case of text documents, this is essentially the file itself; in the case of binary files, this set of words may be metadata bundled with the file (e.g., the ID3 tag of an MP3 file).

B Proof of Lemma 2

Lemma 2. *Let $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \to \{0,1\}$ be a $(\omega_t, \omega_q, \epsilon/8)$-secure pseudo-random function family. Let $g = G_k$, where $k \overset{R}{\leftarrow} \mathcal{K}_{\mathcal{G}}$. Let $\ell_F = o(\log(1/\epsilon))$ such that an $\ell_F \times \ell_F$ random $(0,1)$-matrix is singular with probability at most $\epsilon/4$. Then the matrix*

$$A = \Big[\, g(i,j) \,\Big]_{\substack{i=1,\ldots,\ell_F \\ j=1,\ldots,\ell_F}}$$

is singular with probability at most $\epsilon/2$.

Proof. We know that an $\ell_F \times \ell_F$ random $(0,1)$-matrix is singular with probability at most $\epsilon/4$. However, in our scheme, A is not a random matrix, but a matrix constructed using the pseudo-random function g. Thus, we need the additional proof step to show that the matrix A we constructed using the pseudo-random function g also satisfies the non-singular property with overwhelming probability, otherwise, we could break the pseudo-random function. This proof step is as follows.

Now assume for contradiction that the matrix A is singular with probability greater than $\epsilon/2$. Then we show that we can construct an adversary $\mathcal{B}$ with $\mathsf{Adv}_{\mathcal{B}} > \epsilon/4$ with polynomial number of queries and polynomial time, and thus contradicting the original assumptions of G.

To do so, we play the following game. We flip a coin $\theta \in \{0,1\}$ with a half and half probability, the adversary $\mathcal{B}$ is given one of two worlds in which he can make a number of queries to a given oracle. If $\theta = 1$, $\mathcal{B}$ is given world one, where $g = G_k$, $k \overset{R}{\leftarrow} \mathcal{K}_{\mathcal{G}}$, and the oracle responds to a query (i,j) with $g(i,j)$. If $\theta = 0$, the adversary $\mathcal{B}$ is given world two, where the oracle responds to a query (i,j) by picking a random function R mapping (i,j) to $\{0,1\}$, i.e., by flipping a coin $b \in \{0,1\}$ with a half and half probability and returning b (using a table of previous queries to ensure consistency). After a series of queries, the adversary $\mathcal{B}$ guesses which world he is in. The adversary $\mathcal{B}$ makes his guess using the following strategy: First, the adversary $\mathcal{B}$ constructs a matrix A by querying the oracle for all (i,j) where $i \in \{1,\ldots,\ell_F\}$ and $j \in \{1,\ldots \ell_F\}$; then the adversary $\mathcal{B}$ checks if A is singular. If yes, he guesses that he is in world one. If not, he guesses that he is in world two.

Thus, we can compute the advantage of such an adversary $\mathcal{B}$.

$$\mathsf{Adv}_{\mathcal{B}} = |\Pr[\mathcal{B}^g = 1] - \Pr[\mathcal{B}^R = 1]| = |1/2 \Pr[\text{A is singular}|\theta = 1] - 1/2 \Pr[\text{A is singular}|\theta = 0]| \,.$$

From the above assumptions, $\Pr[\text{A is singular}|\theta = 1] > \epsilon/2$, and $\Pr[\text{A is singular}|\theta = 0] < \epsilon/4$, thus $\mathsf{Adv}_{\mathcal{B}} > \epsilon/8$, contradicting the original assumptions of G. □

C Proof of Lemma 3

Lemma 3. *Given $\ell_F > m + 8\ln(2/\epsilon)$, let $\ell_I = O(m\log(t/m))$, and assume the number of matching files is at most m, the probability that the number of reconstructed matching indices β is greater than ℓ_F is at most $\epsilon/2$.*

Proof. The number of reconstructed matching indices β equals to the number of truly matching files plus the number of false positives from the reconstruction using the Bloom filter. Thus, we need to bound this number of false positives to be at most $\ell_F - m$.

The false positive rate ρ of the Bloom filter storing m entries is as follows [4].

$$\rho = \left(\frac{1}{2}\right)^{\frac{\ell_I \log 2}{m}} \tag{3}$$

Thus, the expectation of the number of false positives is ρt. For simplicity, let's set $\rho t = (\ell_F - m)/2$. Thus $\ell_I = m(\log 2)^{-2}\log(\frac{2t}{\ell_F - m})$. Since ℓ_F is set to be linear in m, with $\ell_I = O(m\log(t/m))$ the expected number of false positives can be bounded far from ℓ_F.

Moreover, we can model the number of false positives with a Bernoulli random variable X with rate parameter ρ and approximate it with a Gaussian centered at the expected number of false positives. From Chernoff bounds, we can derive that $\Pr[X > \ell_F - m] < \exp(-(\ell_F - m)/8)$. Thus, with $\ell_F > m + 8\ln(2/\epsilon)$, we can show that this probability is bounded by $\epsilon/2$. Thus, we show that the above lemma holds. □

D Proof of Theorem 2

Here we provide a proof of the semantic security of the proposed private searching system assuming the semantic security of the Paillier cryptosystem. The proof is simple; in fact it proceeds in the same way as the proof of semantic security in Ostrovsky and Skeith's scheme [18]. The same proof applies whether we are using encrypted queries of the original form proposed by Ostrovsky and Skeith or the hash table queries we propose as an extension.

Theorem 2. *If the Paillier cryptosystem is semantically secure, then the proposed private searching scheme is semantically secure according to Definition 1.*

Proof. We assume there is an adversary $\mathcal{A}$ that can play the game described in Definition 1 with non-negligible advantage ε in order to show that we then have non-negligible advantage in breaking the security of the Paillier cryptosystem.

First we initiate a game with the Paillier challenger, receiving public key n. We choose plaintexts $m_0, m_1 \in \mathbb{Z}_n$ to be simply $m_0 = 0$ and $m_1 = 1$. We return them to the Paillier challenger who secretly flips a coin β_1 and sends us $E\left(m_{\beta_1}\right)$.

Now we initiate a game with $\mathcal{A}$ and send them the modulus n, challenging them to break the semantic security of the private searching system. They send us two sets of keywords, K_0 and K_1. We flip a coin β_2 and construct the query Q_{β_2} by passing K_{β_2} to `QueryConstruction`. Next we replace all the entries in Q_{β_2} which are encryptions of one with $E\left(m_{\beta_1}\right)$, re-randomizing each time by multiplying by a new encryption of zero. Note that with probability one half, $\beta_1 = 0$ and Q_{β_2} is a query that searches for nothing. In this case β_2 has no influence on Q_{β_2} since Q_{β_2} consists solely of uniformly distributed encryptions of zero. Otherwise, Q_{β_2} searches for K_{β_2}.

Next we give Q_{β_2} to $\mathcal{A}$. After investigation, $\mathcal{A}$ returns their guess β_2'. If $\beta_2' = \beta_2$, we let the guess for our challenge be $\beta_1' = 1$ and return it to the Paillier challenger. Otherwise we let $\beta_1' = 0$ and send it to the Paillier challenger.

Since $\mathcal{A}$ is able to break the semantic security of the private searching system, if $\beta_1 = 1$ the probability that $\beta_2' = \beta_2$ is $\frac{1}{2} + \varepsilon$, where ε is a non-negligible function of the security parameter n. If $\beta_1 = 0$, then $\mathrm{P}\left(\beta_2' = \beta_2\right) = \frac{1}{2}$, since β_2 was chosen uniformly at random and it had no bearing on the choice of β_2'. Now we may compute our advantage in our game with the Paillier challenger as follows.

$$\begin{aligned}
\mathrm{P}\left(\beta_1' = \beta_1\right) &= \mathrm{P}\left(\beta_1' = 1 | \beta_1 = 1\right)\frac{1}{2} + \mathrm{P}\left(\beta_1' = 0 | \beta_1 = 0\right)\frac{1}{2} \\
&= \left(\frac{1}{2} + \varepsilon\right)\frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \\
&= \frac{1}{2} + \frac{\varepsilon}{2}
\end{aligned}$$

Since ε is non-negligible, so is $\frac{\varepsilon}{2}$. □