New Techniques for Private Stream Searching

JOHN BETHENCOURT and DAWN SONG
Carnegie Mellon University
and
BRENT WATERS
SRI International

16

A system for private stream searching, introduced by Ostrovsky and Skeith, 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 an upper bound on the size of the documents. The required storage on the server conducting the search is also $O(m)$. The previous best scheme for private stream searching was shown to have $O(m \log m)$ communication and storage complexity. 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. This technique requires a small amount of metadata to be returned in addition to the documents; for this the original scheme of Ostrovsky and Skeith may be employed with $O(m \log m)$ communication and storage complexity. We also present an alternative method for returning the necessary metadata based on a unique encrypted Bloom filter construction. This method requires $O(m \log(t/m))$ communication and storage complexity, where t is the number of documents in the stream. In this article we describe our scheme, prove it secure, analyze its asymptotic performance, and describe a number of extensions. We also provide an experimental analysis of its scalability in practice. Specifically, we consider its performance in the demanding scenario of providing a privacy preserving version of the Google News Alerts service.

Categories and Subject Descriptors: E.3 [**Data Encryption**]: public key cryptosystems

General Terms: Performance, Security

During this project, J. Bethencourt and D. Song received support from the NSF and ARO. B. Waters was supported by NSF CNS-0749931, CNS-0524252, CNS-0716199; the U.S. Army Research Office under the CyberTA Grant No. W911NF-06-1-0316; and the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001.

Authors' emails: J. Bethencourt; email: bethenco@cs.cmu.edu; D. Song; email: dawnsong@cmu.edu; B. Waters; email: bwaters@cls.sri.com.

Additional Key Words and Phrases: Bloom filter, public key program obfuscation, private information retrieval, private stream searching

ACM Reference Format:
Bethencourt, J., Song, D., and Waters, B. 2009. New techniques for private stream searching. ACM Trans. Inf. Syst. Secur. 12, 3, Article 16 (January 2009), 32 pages.
DOI = 10.1145/1455526.1455529. http://doi.acm.org/10.1145/1455526.1455529.

1. INTRODUCTION

The Internet currently has several different types of sources of information. These include conventional Web sites, time sensitive Web pages such as news articles and blog posts, 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 (e.g., a textual keyword) to a server and the server performing the search over some large data set. However, for some applications a client would like to hide their search criteria, that is, which data they are interested in. A client might want to protect the privacy of their search queries for a variety of reasons ranging from personal privacy to protection of commercial interests. 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.

The sensitivity of search terms was highlighted again in 2006 when AOL Research released a database of about 20 million search queries on the Internet, revealing a great deal of compromising information about 658,000 AOL users [Arrington 2006]. Although the database had been anonymized by replacing each username with a unique integer, in many cases it was possible to completely determine the real world identity of a user based on the content of their queries. Note that other privacy preserving technologies such as mix-based anonymizers [Dingledine et al. 2004] do not solve this problem, since the search terms alone are enough to compromise the user's privacy.

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 set to be searched, the limited bandwidth between the client and the remote host, or to the unwillingness of the other party 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 these cases it is advantageous to allow clients to establish persistent searches with the servers where they can be efficiently processed. Content matching the searches can then be periodically returned to the clients. For example, the Google News Alerts system emails users whenever Web news articles crawled by Google match their registered search keywords. As in the case of the AOL search data, it is not hard to imagine queries which could be privacy sensitive. For example, a high-ranking government staff person may wish to be immediately notified if certain information has leaked and is being publicly reported.

However, the search terms they register with the Google News Alerts service may hint at or reveal the very information they sought to protect. Again, anonymizing systems such as Tor [Dingledine et al. 2004] do nothing to solve this problem.

In this article we develop an efficient cryptographic system which allows services of this type while provably maintaining the secrecy of the search criteria. We go on to evaluate its practical applicability using a private version of the Google News Alerts service as an example application.

1.0.1 *Private Stream Searching.* Recently, Ostrovsky and Skeith defined the problem of "private filtering," which corresponds to the type of scenario described above [Ostrovsky and Skeith 2005]. In the same article, they gave a scheme based on the homomorphism of the Paillier cryptosystem [Paillier 1999; Damgård and Jurik 2001] providing this capability. First, a public dictionary of keywords D is fixed. To construct a query for the disjunction[1] 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 nonzero 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 nonzero 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, a collision will result 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, they 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 $O(m \log m)$,[2] where m is an upper bound on the number of matching documents. While effective, this scheme results in inefficiency due to the fact that a significant

[1]They also mention an extension allowing a single conjunction; this extension may also be applied to the scheme presented in this article. However, support for general logical formulas is not possible without doubly homomorphic encryption (see Ostrovsky and Skeith [2007] Corollary 3.6).
[2]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 $O(\log m)$.

Table I. For m Matches in a Stream of t Documents, the New Scheme Retrieves the Bulk Content of the Documents with Linear Overhead. Two Alternatives Are Available for Retrieving the Necessary Metadata. Both Incur Communication Depending Only on the Number of Documents and Not Their Lengths

Private Stream Searching Scheme	Storage and Comm. (For Bulk Content)	Storage and Comm. (For Metadata)	Client Reconstruction Time
Ostrovsky-Skeith 2005	$O(m \log m)$	$O(m \log m)$	$O(m \log m)$
Our scheme (simple metadata)	$O(m)$	$O(m \log m)$	$O(m^{2.376})$
Our scheme (Bloom filter)	$O(m)$	$O(m \log(t/m))$	$O(m^{2.376} + t \log(t/m))$

portion of the buffer returned to the user consists of empty locations and document collisions.

1.0.2 *Our Approach.* In this article 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. Some metadata must be returned, for which we may use the original scheme with $O(m \log m)$ communication and storage. When considering unit length (i.e., one 1024-bit group element) documents, our scheme therefore shares the same overall $O(m \log m)$ communication complexity as Ostrovsky-Skeith. However, because our scheme decouples the logarithmic communication factor from the document length, we strictly improve the communication complexity for longer documents.

We also present an alternative technique for returning the metadata requiring $O(m \log(t/m))$ communication and storage, where t is the total number of documents searched. This latter technique results in the optimal $O(m)$ overall complexity with near optimal constant factors in applications where each document matches the query with some probability, independent of the other documents. One disadvantage of the latter technique is a step in reconstructing the matching documents on the client with $O(t \log(t/m))$ time complexity, introducing a dependency on the overall stream length. However, this step consists only of computing a series of hash values, which is greatly outweighed by other costs in practice. These efficiency improvements and trade-offs are summarized in Table I.

The new results are based on the combination of several novel techniques. Like the approach of Ostrovsky and Skeith we use 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 pseudorandom function, g, the key of which is shared by the client and server, will determine pseudorandomly 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 sufficient information to reconstruct any one 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.

To do so, the client must obtain a list of the indices of the documents in the stream which matched the query. The first method for accomplishing this (hereafter termed the *simple metadata* construction) is based on the original Ostrovsky-Skeith construction. To employ the alternative method (hereafter termed the *Bloom filter* construction), the server maintains a separate encrypted Bloom filter that efficiently keeps track of which document numbers were matched. The Bloom filter construction provides a compact way of representing the set indices of matching documents and normally requires much less space than the simple metadata construction.

1.1 Related Work

Private searching may be viewed as the flip side of searching on encrypted data [Song et al. 2000; Boneh et al. 2004; Goh 2003], 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 (PIR) [Chor et al. 1995; Kushilevitz and Ostrovsky 1997; Cachin et al. 1999; Chang 2004] and oblivious transfer [Naor and Pinkas 1999; Lipmaa 2005]. 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, both the system proposed in this article 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 constructions supporting retrieval by keyword rather than block index in the PIR literature [Chor et al. 1997; Kurosawa and Ogata 2004; Freedman et al. 2005], 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.

Portions of the work presented in this article have previously appeared as an extended abstract [Bethencourt 2006a] and tech report [Bethencourt 2006b].

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, the definition of a pseudorandom function family, and Bloom filters. Appendix A provides a reference for the terms and variables defined throughout this section and the rest of the article; the reader is encouraged to consult it if unclear.

2.1 Problem Definition

In a private searching scheme a client will create an encrypted query for the set of keywords that they are 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[3] 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 its 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 or which files match those keywords. We now formally describe a private stream searching scheme. A scheme for private stream searching consists of the following three algorithms.

2.1.1 *QueryConstruction*$(\lambda, \epsilon, m, K)$. The `QueryConstruction` algorithm is run by a client to prepare an encrypted list of keywords that they 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 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.

2.1.2 *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

[3]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. Also, when presenting our scheme we initially consider fixed length files; an extension in Section 5.3 relaxes this restriction without affecting the scheme's complexity.

algorithm updates a buffer of encrypted results R after processing each file and eventually sends it back to the client.

2.1.3 *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 searching 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 probabilistic, polynomial time (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 4.4.

Note that the security definition for private stream searching necessitates that the server return the same amount of data regardless of how many files matched the query. If this were not the case, the server could easily mount a dictionary attack using the StreamSearch algorithm to determine the exact query keywords. As a result, like all other proposals for private stream searching, our scheme requires an a priori upper bound m on the number of files to retrieve (or the total length of the files when using the extension of Section 5.3).

2.2 Preliminaries

2.2.1 *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 $K_{pub} = n$ is the product of two large primes. The factorization of n is the private key. In this article the encryption of a plaintext m with the public key (there is only one public key in use in this article, 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 $E : \mathbb{Z}_n \to \mathbb{Z}^*_{n^2}$ and $D : \mathbb{Z}^*_{n^2} \to \mathbb{Z}_n$. Note that ciphertexts are twice as large as plaintexts.[4]

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

[4]This property of inflating messages by encrypting them is improved in a generalization of the Paillier cryptosystem [Damgård and Jurik 2001]. 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 article are likely to make the original situation of $s = 1$ preferable in practice.

$D(E(a) \cdot E(b)) = a+b$. 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 article.

2.2.2 *PseudoRandom functions.* In our construction we use a pseudorandom function family $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$. Roughly speaking, G will take in a key k and two integers and output a pseudorandom bit. We let $g = G_k$ where $k \xleftarrow{R} \mathcal{K}_G$.

The security of a pseudorandom function family $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$ 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:

$$\mathsf{Adv}_{\mathcal{A}} = \left| \mathrm{P}\left(\beta = \beta'\right) - \frac{1}{2} \right|$$

We say that a pseudorandom function is $(\omega_t, \omega_q, \epsilon)$-secure if no ω_t time adversary, that makes at most ω_q oracle queries, has advantage greater than ϵ. As explained in Section 4, the "security" of the pseudorandom function family employed in our scheme is actually only necessary to prove correctness properties. Privacy is unaffected.

2.2.3 *Bloom filters.* A Bloom filter [Bloom 1970] is a space-efficient data structure for storing a set of keys that has several unique features. First, rather than allowing direct enumeration of the keys stored, a Bloom filter only supports querying to determine if a given key is present. Second, while queries for a key that has been previously stored will always succeed, a query for a key which has not been previously stored will also succeed with some small, configurable probability. This false-positive inducing "lossiness" allows Bloom filters to achieve extremely compact storage.

A Bloom filter may be implemented as a vector of ℓ bits $v_1, v_2, \ldots v_\ell \in \{0, 1\}$, all initially zero, and a collection of k hash functions $h_i : \{0, 1\}^* \to \{1, 2, \ldots \ell\}$, $i \in \{1, 2, \ldots k\}$. To insert a key $x \in \{0, 1\}^*$, we set $v_{h_1(x)} = v_{h_2(x)} = \cdots v_{h_k(x)} = 1$. To query for a key y, we check whether $v_{h_i(y)} = 1$ for all $i \in \{1, 2, \ldots k\}$ and return true if so. If $v_{h_i(y)} = 0$ for some $i \in \{1, 2, \ldots k\}$, we return false. Based on the number of keys one expects to store and a desired false positive rate, optimal values for ℓ and k may be selected [Broder and Mitzenmacher 2005].

3. NEW CONSTRUCTIONS

We now describe the algorithms of the new private search scheme and give an analysis of complexity and security properties. For ease of exposition, we first

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)
```

Fig. 1. The algorithm for setting up an encrypted query.

describe the version of the scheme using the Bloom filter construction, then give the modifications necessary to employ the simple metadata construction. Additionally, we defer discussion of several special failure cases to the next section.

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, regardless of whether the simple metadata construction or Bloom filter construction will be used. 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 (independently randomizing 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 (Bloom Filter Construction)

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 parameters ℓ_F, ℓ_I, and k, which affect correctness and performance (see below and Section 4).

3.2.1 *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

Algorithm: `StreamSearch`
Input: Q, n, 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, pseudo-random function $g : \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$.
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  :
     /* Step 1 */
     c := E(0)
     for  w_j ∈ W_i  :
          c := c · Q[j] mod n^2

     /* Steps 2 and 3 (in parallel) */
     e := c^{f_i} mod n^2   /* now c = E(c_i) and e = E(c_i f_i) */
     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

     /* Step 4 */
     for  j := 1, 2, ..., k  :
          ℓ := h_j(i) mod ℓ_I
          I[ℓ] := I[ℓ] · c mod n^2
```

Fig. 2. The algorithm for running the private search, using the Bloom filter construction.

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 $\lfloor \log_2 n \rfloor$ 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.

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 during 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 an 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, which may be computed by the server using the client's public key.[5] We now detail how they are updated as each file is processed.

3.2.2 *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 the c-value of the file, that is, the number of distinct members of K found in the file. That is,

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

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 $E(c_i f_i)$ using the homomorphic property of the Paillier cryptosystem.

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

The server multiplies the value $E(c_i f_i)$ into a subset of the locations in the data buffer according to the following procedure. Let G be a family of pseudorandom functions that map $\mathbb{Z} \times \mathbb{Z}$ to $\{0, 1\}$. Randomly select $g \xleftarrow{R} G$ (this should be done once upon initialization and the same g used for all files). The algorithm multiplies $E(c_i f_i)$ 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 the first file was also multiplied into this location, that is, $g(1, 3) = g(2, 3) = 1$. 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 $D\left(F[3]\right) = c_1 f_1 \bmod n$. After multiplying in $E(c_2 f_2)$, $D\left(F[3]\right) = c_1 f_1 + c_2 f_2 \bmod n$. But $c_2 = 0$ since f_2 does not match, so it is still the case that $D\left(F[3]\right) = c_1 f_1 \bmod n$ and the data buffer is effectively unmodified.

[5]Since these values need not be individually randomized, it actually suffices to initialize each to the value one, which is a valid Paillier encryption of zero under any public key.

This mechanism causes the data buffer to accumulate linear combinations of matching files while discarding all nonmatching files.

Note that, as shown in Figure 2, the server multiplies ciphertexts modulo n^2; this results in the underlying plaintexts being added modulo n. Naturally, when several files are added modulo n, the result will "wrap around" and be mapped back into $\mathbb{Z}_n$. It is important to realize that this does not result in a loss of essential information or pose any problem to the scheme. Provided there are as many (independent) linear equations as file blocks, the value of each file block will be uniquely determined, and the client will be able to correctly recover each of the files using the `FileReconstruction` algorithm.

Step 3: Update c-buffer. The value $E(c_i)$ is multiplied into each of the locations in the c-buffer in a similar fashion as $E(c_i f_i)$ was used to update the data buffer. In particular, the server multiplies the value $E(c_i)$ into each location j in the c-buffer where $g(i, j) = 1$.

Step 4: Update matching-indices buffer. The server then multiplies $E(c_i)$ further into a fixed number of locations in the 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 $E(c_i)$ 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 [Broder and Mitzenmacher 2005]. 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 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.

3.3 Client's `FileReconstruction` Procedure (Bloom Filter Construction)

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

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 nonzero, 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

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

```
/* Step 1 */
for  i := 1, 2, ..., ℓ_F  :
     F'[i] := D(F[i])
     C'[i] := D(C[i])
for  i := 1, 2, ..., ℓ_I  :
     I'[i] := D(I[i])

/* Step 2 */
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}})

/* Step 3 */
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

/* Step 4 */
f⃗ := diag(c⃗)^{-1} · A^{-1} · F'
output  f_{α'_1}, f_{α'_2}, ..., f_{α'_r}
```

Fig. 3. The algorithm for recovering the matching files after the completion of a private search when using the Bloom filter construction.

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 $(c_{\alpha_i})_{i=1,\ldots,\ell_F}$.[6] 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 have been processed ($t = 7$), and from Step 2 we have established the following list of potentially matching indices: $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} = \{1, 3, 5, 7\}$. Further suppose that the matrix induced by the pseudorandom function g is

$$A = \Big(g(\alpha_i, j)\Big)_{\substack{i=1,2,\ldots,\ell_F \\ j=1,2,\ldots,\ell_F}} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} .$$

Then if the c-buffer decrypts to the column vector $C' = (2 \;\; 3 \;\; 1 \;\; 3)$, we may establish the following linear system, since $A \cdot \vec{c} = C'$.

$$\begin{aligned} c_{\alpha_1} + c_{\alpha_3} &= 2 \\ c_{\alpha_1} + c_{\alpha_2} + c_{\alpha_4} &= 3 \\ c_{\alpha_1} + c_{\alpha_4} &= 1 \\ c_{\alpha_2} + c_{\alpha_3} &= 3 \; . \end{aligned}$$

Solving, we obtain

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

The seventh file is ignored since it only appeared due to a Bloom filter false positive, so we see that there were three actual matching files ($r = 3$): f_1, f_3, and f_5.

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

Step 4: Reconstruct matching files. Continuing in our description of the general algorithm with Step 4, 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

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

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, at the end of Step 3, we replace all zeros in $\vec{c}$ with ones. The content of the matching files appears as $f_{\alpha'_1}, f_{\alpha'_2}, \ldots, f_{\alpha'_r}$; the other entries in $\vec{f}$ will be zero.

Continuing the example started in the description of Step 3, suppose the data buffer decrypts to F' = (32 32 10 44). Of course, these are artificially small values; in reality they would typically be 1024 bits each. Then we may solve the following system

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

to determine that $f_1 = 10$, $f_3 = 11$, and $f_5 = 22$ (and $f_7 = 0$, but this value is ignored since $c_7 = 0$).

Keep in mind that the linear equations for the file blocks and c-values are modulo n; that is, the values appearing the decrypted buffers F' and C' were computed modulo n as explained in the description of the `StreamSearch` algorithm. The above example was shown using standard arithmetic for simplicity, but a system of linear equations modulo n may be solved in the same way. The original values of each c_i and f_i will be recovered as expected.

3.4 Simple Metadata Construction

Now that we have defined the version of the scheme incorporating the (more complex) Bloom filter construction, we may easily describe the differences between this version of the scheme and the variant using the simple metadata construction. In applications where the expected number of matching documents is fixed and independent of the stream length, this latter variant is preferable since it does not require communication and storage dependent on the stream length. To produce this effect, we abandon the Bloom filter 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 Ostrovsky and Skeith [2005].

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.

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

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. There are now no false positives for streams of any length.

4. ANALYSIS

In this section, we analyze the correctness and complexity of both variants of our scheme and prove their security.

4.1 Computational Complexity

The running time of the first client side algorithm, `QueryConstruction`, is $O(|D|)$. This is exactly the same as in Ostrovsky-Skeith, in which the encrypted queries take the same form. More precisely, `QueryConstruction` requires $|D|$ exponentiations and $|K| \leq |D|$ multiplications. For large dictionaries, this is a significant cost in both our scheme and Ostrovsky-Skeith; Section 5.2 presents an extension to our scheme which can greatly reduce this cost.

When using the Bloom filter construction, the `StreamSearch` algorithm has running time $O(|W_i| + s \cdot m + \log(t/m))$ when processing the ith file. Recall that W_i is the set of keywords associated with that file and s is the number of plaintext blocks required to store the contents of a file. With the simple metadata construction, the `StreamSearch` algorithm runs in time $O(|W_i| + s \cdot m + \log(m))$. In either case, however, only s exponentiations are required; the rest of the computation results from multiplications.

The `FileReconstruction` algorithm runs in time $O(s \cdot m + m^{2.376} + t\log(t/m))$ when using the Bloom filter construction or $O(s \cdot m + m^{2.376})$ with the simple metadata construction. Note that the $s \cdot m$ term is necessary to simply output the results. The $m^{2.376}$ term corresponds to solving a system of linear equations [Coppersmith and Winograd 1990], and the $t\log(t/m)$ term is the time to check each possible document index against the Bloom filter. Asymptotically, these running times are neither strictly better nor strictly worse than the $O(s \cdot m \log m)$ file reconstruction time with Ostrovsky-Skeith. In practice, however, we find that file reconstruction is far faster using either of the new schemes; this is considered in detail in Section 6.4.

4.2 Correctness and Communication Complexity (Bloom Filter Construction)

We now consider correctness and communication complexity, beginning with the scheme employing the Bloom filter construction. 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 was proven in the 60's [Komlós 1967]. The specific bound has since been improved several times, recently reaching $O\left(\left(\frac{3}{4}+o(1)\right)^n\right)$ [Kahn et al. 1995; Tao and Vu 2005, 2007]. Thus, it is easy to see that the above lemma holds. □

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 pseudorandom function family. Let $g=G_k$, where $k\stackrel{R}{\leftarrow}\mathcal{K}_G$. 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 $\ell_F = o(\log(1/\epsilon)) + O(m)$, $\ell_F > m + 8\ln(2/\epsilon)$, $\ell_I = O(m \log(t/m))$, and $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \to \{0,1\}$ is a $(\omega_t, \omega_q, \epsilon/8)$-secure pseudorandom function family, then when the number of matching files is at most m in a stream of t, the new scheme using the Bloom filter construction 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. □

4.3 Correctness and Communication Complexity (Simple Metadata Construction)

We now consider the correctness and complexity in the case of using the simple metadata construction.

THEOREM 2. *If $\ell_F = o(\log(1/\epsilon)) + O(m)$, $\ell_F > m + 8\ln(2/\epsilon)$, $\ell_I = O(m(\log m + \log(1/\epsilon)))$, and $G : \mathcal{K}_G \times \mathbb{Z} \times \mathbb{Z} \to \{0,1\}$ is a $(\omega_t, \omega_q, \epsilon/8)$-secure pseudorandom function family, then when the number of matching files is at most m, the new scheme using the simple metadata construction guarantees that the client can correctly reconstruct all matching files with probability at least $1 - \epsilon$.*

PROOF. Briefly, the argument for Theorem 1 may be applied again, except that we no longer need Lemma 3. Instead, we refer to the analysis in Ostrovsky and Skeith [2005] that demonstrates that the probability of an overflow in the alternative matching-indices buffer may be bounded below ϵ with $\ell_I = \gamma m$ where $\gamma = O(\log m + \log(1/\epsilon))$, producing an overall communication and storage complexity of $O(m \log m)$. □

Note that our scheme still produces a constant factor improvement over Ostrovsky-Skeith in this case. If each file requires s plaintext blocks (i.e., is of length at most $s \cdot \lfloor \log_2 n \rfloor$ bits), then we reduce communication and storage by a factor of approximately $\log(sm)$ for large files. 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.

4.4 Security

The security of the proposed scheme (in both variants) 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 3. *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. Since the proof depends only on the form of the encrypted query, the variant of the scheme which will be used is irrelevant. Note that this theorem establishes security based on the decisional composite residuosity assumption (DCRA), since the Paillier cryptosystem has been proven semantically secure based on that assumption [Paillier 1999].

5. EXTENSIONS

Here we describe a number of extensions to the proposed system which provide additional features.

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) [Silverman 2001]. If the Bloom filter construction is used, the fact that the c-values will never approach 2^{1024} reveals that most of its space is wasted. A simple technique allows improved usage 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, it may simply break up each element into 64 regions of 16 bits. However, this space savings comes at an additional computation cost. The server will need to perform k additional modular exponentiations for each file it processes.

5.2 Hashing Keywords

In some applications, a 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.

See Section 6.1 for a discussion of the size and computation time corresponding to various query sizes in practical settings.

5.3 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. For convenience, we describe it in terms of the version of the scheme employing the Bloom filter; applying this technique to the other variant is straightforward.

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 3.4, $O(m_1 \log m_1)$). 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 pseudorandom function, d_i is the length of the ith file in the stream, and the d_i blocks of the file are denoted $f_{i,1}, f_{i,2}, \ldots, f_{i,d_i}$.

$e := c^{d_i} \bmod n^2$
`for` $j := 1, 2, \ldots, \ell_C$:
  `if` $g(i, j) = 1$:
    $C[j] := C[j] \cdot c \bmod n^2$
    $L[j] := L[j] \cdot e \bmod n^2$

`for` $j_1 := 1, 2, \ldots, d_i$:
  $e := c^{f_{i,j_1}} \bmod n^2$
  `for` $j_2 := 1, 2, \ldots, \ell_F$:
    `if` $\hat{g}(i, j_1, j_2) = 1$:
      $F[j_2] := F[j_2] \cdot e \bmod n^2$

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 $\vec{d} := \text{diag}(\vec{c})^{-1} \cdot A^{-1} \cdot L'$. 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 $\vec{f} := \text{diag}(\vec{c}\,')^{-1} \cdot \hat{A}^{-1} \cdot F'$, 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 among the files.

5.4 Merging Parallel Searches

Another extension makes multiple server, distributed searches possible. Suppose a collection of servers each have their own stream of files. The client wishes to run a private search on each of them, but does not wish to separately download and decipher a full size buffer of results from each. Instead, the client wants the servers to remotely merge their results before returning them.

This can be accomplished by simply having each server separately run the search algorithm, then multiplying together (element by element) each of the arrays of resulting ciphertexts. This merging step can take place on a single collecting server, or in a hierarchical fashion. A careful investigation of the algorithms reveals that the homomorphism ensures the result is the same as it would be if a single server had searched the documents in all the streams. Care

must be taken, however, to ensure the uniqueness of the document indices across multiple servers. This can be accomplished by, for example, having servers prepend their IP address to each document index. Also, it is of course necessary for the buffers on each server to be of the same length.

Note that if the client splits its query and sends it to each of the remote servers, a different set of keywords may be used for each stream. Alternatively, a server may split a query to be processed in parallel for efficiency without the knowledge or participation of the client.

6. PRACTICAL PERFORMANCE ANALYSIS

To better assess the applicability of the new private stream searching scheme in practical scenarios, we now give a detailed analysis of a realistic application. Specifically, we consider the case of making a private version of Google's News Alerts service[7] using the new construction. According to the Google News Web site, their Web crawlers continuously monitor approximately 4,500 news Web sites. These include major news portals such as CNN along with many Web sites 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 it receives from the server (t_r). Due to the potential sensitivity of search keywords, we will not use a public dictionary and we instead assume the use of the hashing extension described in Section 5.2.

We assume that the client is able to estimate an upper bound m on the number of files in the stream of t that will match the query. In situations where m may be more difficult to estimate or bound, an alternative method for selecting it may be used, at the expense of a small loss in privacy. Specifically, the server may assist the client in selecting m by computing and returning encrypted c-values for a series of files during some initial monitoring period. After decrypting the c-values, the client will know exactly how many files matched their query during the monitoring period and use this information to select m before beginning the normal stream search. While one iteration of this process may provide the server with some information about possible keywords in the query, a full dictionary attack will not be possible due to the required client participation in decrypting any c-values.

6.1 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 probability θ

[7]See http://www.google.com/alerts.

Table II. Size of the Encrypted Query Necessary to Achieve a Given Spurious Match Rate Before and After Optimizations

θ	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

that a nonmatching file f_i will nevertheless result in a nonzero corresponding $E(c)$ (rearranged on the right to solve for ℓ_Q).

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

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 II. The first column specifies a rate of spurious matches θ 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 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[8]) 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) [Silverman 2001], in some applications only short term guarantees of secrecy may be required. Also, 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 (which will also yield many false positives). Given this consideration, if the client decides a smaller key length is acceptable, s_q will be reduced. The third column in Table II 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 to be much greater in the system of Ostrovsky and

[8]Information available at http://www.natcorp.ox.ac.uk.

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, s_q will be over 70 GB. It is important to emphasize, however, that the system is not truly stream length independent when using the keyword hashing technique. Checking longer streams may result in more false positives, but when using a public dictionary as in Ostrovsky and Skeith, no false positives are possible.

6.2 Storage Buffers Space

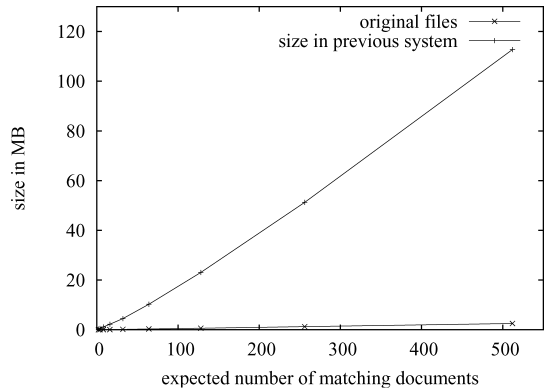
We now turn to the size of the buffers maintained by the server during the search and then sent back to the client. This cost, s_b, is both a storage requirement of the server conducting the search and a communication requirement at the end of the search. We assume fixed length files for this application rather than employing the extension of Section 5.3. In Bloom filter variant of the new scheme, to store the data buffer F, the c-buffer C, and the matching-indices buffer I, the server then uses

$$s_b = 256(s\ell_F + \ell_F + \ell_I)$$

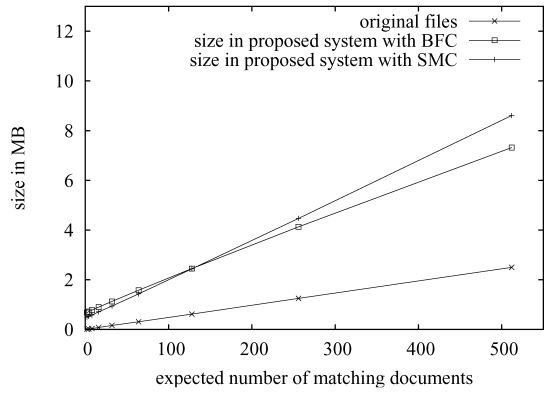
bytes, where s is the number of number of plaintexts from $\mathbb{Z}_n$ required to represent an article and we assume the use of a 1024-bit key.

The client will specify ℓ_F and ℓ_I 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. 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 instead estimate an upper bound m. Based on this estimate, the analysis in Section 4 may be used to select values for ℓ_F and ℓ_I that ensure the probability of an overflow is acceptably small. In these experiments, we determined the minimum values for ℓ_F and ℓ_I empirically.

A range of desired values of m were considered and the results are displayed in Figure 4(b). In each case, ℓ_F and ℓ_I were selected so that the probability of an overflow was less than 0.01. In computing this probability, we treated the number of documents which actually match the query as a binomial random variable with t trials and rate parameter m/t, as would be the case if each matches with some probability independent of the others. Also, the spurious match rate θ 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, Figure 4(b) reveals that s_b is about 2.5 times the size of the matching files. We also show the result of using the simple metadata construction with the new scheme, which performs about as well as the Bloom filter construction for small searches but becomes less efficient with larger numbers of documents. For comparison, the data stored



(a) Ostrovsky-Skeith



(b) New system

Fig. 4. Server to client communication after a period of searching in the previous scheme and in the proposed scheme, using both the Bloom filter construction (BFC) and simple metadata construction (SMC). Note the difference in scale.

by the server and returned to the client using the Ostrovsky-Skeith scheme for private searching in this scenario is shown in Figure 4(a).[9] Note that this graph differs in scale from Figure 4(b) by a factor of 10.

To summarize, in the proposed system s_b ranges from about 680 KB to about 7.3 MB when the expected number of matching files ranges from two to 512 and the overflow rate is held below 0.01. In the Ostrovsky-Skeith scheme, s_b would range from approximately 280 KB to 110 MB.

[9]The article 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.

Table III. The Time Necessary for the Server to Process a File

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

6.3 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. With 768-bit keys, multiplications and exponentiations took 3.9μs and 6.2 ms respectively. With 1024-bit keys, the times increased to 6.3μs and 14.7 ms.

The first step in processing the ith file in the `StreamSearch` procedure is computing $E(c)$; this takes $|W_i| - 1$ multiplications. We again use $|W_i| = 540$ as described in Section 6.1. Computing $E(cf_i)$ requires s modular exponentiations. Updating the data buffer requires an average of $s \cdot \frac{\ell_F}{2}$ modular multiplications, updating the c-buffer requires another $\frac{\ell_F}{2}$ multiplications, and updating the matching-indices buffer requires $k = \lfloor \frac{\ell_I \log 2}{m} \rfloor$ multiplications. The time necessary for these steps is given for several values of m in Table III. The majority of t_p is due to the s modular exponentiations. Since the Ostrovsky-Skeith scheme requires the same number of modular exponentiations, the processing time for each file would be similar.

6.4 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 [Damgård and Jurik 2001]. The times necessary to decrypt the buffers are given in the third column of Table IV. This time is typically less than a minute, but can take as long as five with many files.

The most straightforward way to solve the system of linear equations is by performing LUP decomposition over $\mathbb{Z}_n$. Although LUP decomposition of an $n \times n$ matrix is expensive ($\Theta(n^3)$), practical cases are quite feasible. A naive implementation resulted in the benchmarks shown in the fourth column of Table IV. The total time to recover the matching files is given in the final column of Table IV.

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

Table IV. Time (in Seconds and Minutes) Necessary to Recover the Original Documents from the Returned Results

key length	m	decryption time	inversion time	total time
768	2	14 s	<0.1 s	14 s
768	8	15 s	<0.1 s	15 s
768	32	23 s	<0.1 s	23 s
768	128	54 s	1.4 s	55 s
768	512	2.7 m	1.8 m	4.5 m
1024	2	23 s	<0.1 s	23 s
1024	8	26 s	<0.1 s	26 s
1024	32	38 s	<0.1 s	38 s
1024	128	1.4 m	21 s	1.8 m
1024	512	4.4 m	2.9 m	7.3 m

buffer size and resulting reduction in decryption time. 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.

7. CONCLUSION

The primary contribution of our scheme is achieving the optimal linear overhead in returning the bulk content of matching files to the client. Our scheme also requires either $O(m \log m)$ or $O(m \log(t/m))$ space to return some metadata, depending on the variant used. In the common streaming case of each document matching independently from other documents, the latter variant results in the optimal $O(m)$ complexity, with near optimal constant factors. With the former variant, significant 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 latter also employing a unique encrypted Bloom filter construction. Finally, we proved correctness and security results for the scheme and noted some extensions.

We have also given a detailed example demonstrating the new techniques in 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. Several times per day, they download approximately 500 KB to 7 MB of new search results, allowing up to approximately 500 articles per time interval. After receiving the encrypted results, the client's PC spends under a minute recovering the original files, or up to approximately seven minutes if many files are retrieved. To provide the searching service, the server keeps approximately 500 KB to 7 MB of storage for the client and spends roughly 500 ms processing each new article it encounters. In this scenario, the previous scheme would require up to twelve times

the communication and take up to four times as long for the client to recover the results.

APPENDIX A. TERMS AND NOTATION

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

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

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

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, that is, the set of keywords available for use in queries

K. set of keywords forming a query

w_i. ith word in D

q_i. ith entry in the query array (before encryption), associated with w_i

f_i. ith file processed by the server

W_i. words present in or associated with the ith file[10]

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

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

r. number of files which actually match the query

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

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

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

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

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

s_q. size of the query sent to the server

s_b. size of the storage buffers kept by the server while running the search and eventually transmitted back to the client

[10]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).

t_p. time for the server to process a single file in its stream

t_r. time for the client to decrypt and recover the original matching files from the information it receives from the server

APPENDIX 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 pseudorandom function family. Let* $g = G_k$*, where* $k \overset{R}{\leftarrow} \mathcal{K}_G$*. 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 $\epsilon/2$*.*

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 pseudorandom function g. Thus, we need the additional proof step to show that the matrix A we constructed using the pseudorandom function g also satisfies the non-singular property with overwhelming probability, otherwise, we could break the pseudorandom 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 $\mathcal{B}$ (relative to the pseudorandom function family G) with $\mathsf{Adv}_{\mathcal{B}} > \epsilon/8$ with polynomial number of queries and polynomial time, 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 it can make a number of queries to a given oracle. If $\theta = 1$, $\mathcal{B}$ is given world one, where $g = G_k$, $k \overset{R}{\leftarrow} \mathcal{K}_G$, 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 $\{0,1\}$, that is, 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 it is in. The adversary $\mathcal{B}$ makes its 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, it guesses that it is in world one. If not, it guesses that it is in world two.

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

$$\begin{aligned}\mathsf{Adv}_{\mathcal{B}} &= \left|\mathrm{P}\left(\mathcal{B}^g = 1\right) - \mathrm{P}\left(\mathcal{B}^R = 1\right)\right| \\ &= \left|\frac{1}{2}\mathrm{P}\left(\text{A is singular}|\theta = 1\right) - \frac{1}{2}\mathrm{P}\left(\text{A is singular}|\theta = 0\right)\right| .\end{aligned}$$

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

APPENDIX 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 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 $\epsilon/2$.*

PROOF. The number of reconstructed matching indices β equals 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 [Broder and Mitzenmacher 2005].

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

Thus, the expectation of the number of false positives is ρt. For simplicity, let's set $\rho t = (\ell_F - m)/2$, which corresponds to the number of false positives filling about half the extra space on average. This choice is somewhat arbitrary, but it suffices to allow the proof to go through. So now $\ell_I = m(\log 2)^{-2}\log(\frac{2t}{\ell_F - m})$. 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 binomial 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 $\mathrm{P}(X > \ell_F - m) < \exp(-(\ell_F - m)/8)$. 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. □

APPENDIX D. PROOF OF THEOREM 3

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 [2005]. 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 in Section 5.2.

THEOREM 3. *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 nonnegligible advantage ε in order to show that we then have nonnegligible 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 $E(m_{\beta_1})$.

Now we initiate a game with $\mathcal{A}$ and send it the modulus n, challenging it to break the semantic security of the private searching system. The adversary sends 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 $E(m_{\beta_1})$, rerandomizing 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 its 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 $\mathrm{P}(\beta_2' = \beta_2) = \frac{1}{2}$, 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.

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

Since ε is nonnegligible, so is $\frac{\varepsilon}{2}$. □

REFERENCES

ARRINGTON, M. 2006. Aol proudly releases massive amounts of user search data. *TechCrunch News*.

BETHENCOURT, J., SONG, D., AND WATERS, B. 2006a. New constructions and practical applications for private stream searching (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy (SP'06)*.

BETHENCOURT, J., SONG, D., AND WATERS, B. 2006b. New techniques for private stream searching. Tech. rep. CMU-CS-06-106, Carnegie Mellon University.

BLOOM, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM 13*, 7, 422–426.

BONEH, D., CRESCENZO, G. D., OSTROVSKY, R., AND PERSIANO, G. 2004. Public key encryption with keyword search. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'04)*.

BRODER, A. AND MITZENMACHER, M. 2005. Network applications of bloom filters: A survey. *Internet Math. 1*, 4, 485–509.

CACHIN, C., MICALI, S., AND STADLER, M. 1999. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Texhniques (EUROCRYPT'99)*.

CHANG, Y.-C. 2004. Single database private information retrieval with logarithmic communication. In *Proceedings of the Symposium on Information Security and Privacy (ACISP'04)*.

CHOR, B., GILBOA, N., AND NAOR, M. 1997. Private information retrieval by keywords. Tech. rep. CS0917, Department of Computer Science, Technion.

CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. 1995. Private information retrieval. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'95)*.

COPPERSMITH, D. AND WINOGRAD, S. 1990. Matrix multiplication via arithmetic progressions. *J. Symb. Comput. 9*, 251–280.

DAMGÅRD, I. AND JURIK, M. 2001. A generalization, a simplification and some applications of Paillier's probabilistic public-key system. In *Proceedings of the Public Key Cryptography (PKC'01)*.

DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. 2004. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium (SECURITY'04)*.

FREEDMAN, ISHAI, PINKAS, AND REINGOLD. 2005. Keyword search and oblivious pseudorandom functions. In *Proceedings of the Theory of Cryptography Conference (TCC'05)*.

GOH, E.-J. 2003. Secure indexes. Cryptology ePrint Archive, Report 2003/216.

KAHN, J., KOMLÓS, J., AND SZEMERÉDI, E. 1995. On the probability that a random ± 1 matrix is singular. *J. Amer. Math. Soc. 8*, 1, 223–240.

KOMLÓS, J. 1967. On the determinant of (0,1)-matrices. *Studia Math. Hungarica 2*, 7–21.

KUROSAWA, K. AND OGATA, W. 2004. Oblivious keyword search. *J. Complex. 20*.

KUSHILEVITZ, E. AND OSTROVSKY, R. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'97)*.

LIPMAA, H. 2005. An oblivious transfer protocol with log-squared communication. In *Proceedings of the Information Security Conference (ISC'05)*.

NAOR, M. AND PINKAS, B. 1999. Oblivious transfer and polynomial evaluation. In *Proceedings of the Symposium on Theory of Computing (STOC'99)*.

OSTROVSKY, R. AND SKEITH, W. 2005. Private searching on streaming data. In *Proceedings of the Annual International Cryptology Conference (CRYPTO'05)*.

OSTROVSKY, R. AND SKEITH, W. 2007. Algebraic lower bounds for computing on encrypted data. Tech. rep. TR07-022, Electronic Colloquium on Computational Complexity.

PAILLIER, P. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptograhic Techniques (EUROCRYPT'99)*.

SILVERMAN, R. 2001. A cost-based security analysis of symmetric and asymmetric key lengths. Tech. rep. RSA Laboratories. Nov.

SONG, D. X., WAGNER, D., AND PERRIG, A. 2000. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'00)*.

TAO, T. AND VU, V. H. 2005. On random ± 1 matrices: singularity and determinant. In *Proceedings of the Symposium on Theory of Computing (STOC'05)*. 431–440.

TAO, T. AND VU, V. H. 2007. On the singularity probability of random bernoulli matrices. *J. Amer. Math. Soc. 20*, 3, 603–628.

Received July 2007; revised June 2008; accepted August 2008