Conjunctive Wildcard Search over Encrypted Data

Christoph Bösch, Richard Brinkman, Pieter Hartel, and Willem Jonker

University of Twente,
Enschede, The Netherlands

Abstract. Searchable encryption allows a party to search over encrypted data without decrypting it. Prior schemes in the symmetric setting deal only with exact or similar keyword matches. We describe a scheme for the problem of wildcard searches over encrypted data to make search queries more flexible, provide a security proof for our scheme and compare the computational, communication and space complexity with existing schemes. We develop an efficient scheme, using pseudorandom functions and Bloom filters, that supports wildcard searches over encrypted data. The scheme also supports conjunctive wildcard searches, efficient and secure updates and is more efficient than previous solutions. Besides, our construction is independent of the encryption method of the remote data and is practical to use in real world applications.

Key Words. Searchable Encryption, Bloom filter, Wildcard.

1 Introduction

Nowadays, remote storage is ubiquitous and widely used for services like outsourcing data to reduce operational costs or private backups. To securely store outsourced data on an untrusted server, the data should be encrypted which makes it impossible for inside and outside attackers to access the data, but at the same time the data owner loses all searching capabilities. It is desirable to support (full) searching functionality on the server side, without decrypting the data, and thus, without any loss of data confidentiality. This is typically called *searchable encryption* (SE).

Over the last decade there has been active research in the symmetric [7, 8, 10, 16] and the public key setting [4, 6]. To construct efficient schemes we focus on searchable symmetric encryption (SSE), where the same client stores and retrieves encrypted documents. Prior SSE schemes support only exact keyword matches or similarity searches, where keyword similarity is measured in the Hamming or edit distance. To get more flexibility in the search queries, we create a new construction that supports wildcard searches over encrypted data, where a wildcard may represent any number of characters. We present a construction that supports conjunctive wildcard searches, where a conjunction is the union of any number of keywords.

Protocol. We consider a user $\mathcal{U}$ who stores a set of encrypted documents on an honest-but-curious [11] database server $\mathcal{S}$ that can be trusted to adhere to the protocol, but which tries to learn as much information as possible. $\mathcal{U}$ later wants to retrieve some of the documents containing a specific keyword. To do so, $\mathcal{U}$ first generates an index over his documents and then stores the index and the encrypted documents on the server. The index allows $\mathcal{U}$ to search the encrypted documents. To search for a specific keyword in the document collection, $\mathcal{U}$ creates a trapdoor for that keyword and sends this trapdoor to the server which then returns the result indicating which documents match the query and which not. $\mathcal{U}$ then decides which of the documents she wants to retrieve and sends the document ids to $\mathcal{S}$. The server returns the requested documents.

Related Work Searchable encryption can be achieved by using the works of Ostrovsky and Goldreich [12,14,15] on *oblivious RAMs* from 1990, which hide all information including the access pattern, from a remote server. Unfortunately the scheme is not efficient in practice. The scheme needs a logarithmic number of rounds of interaction for each read and write.

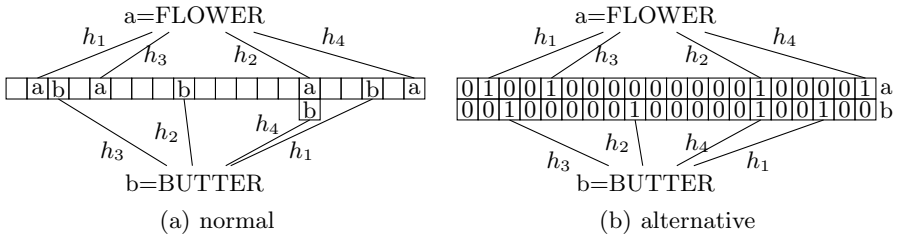
The first practical scheme for searching in encrypted data in the symmetric setting was proposed by Song et al. [16] in 2000. They use a special two-layered encryption construct which is known as a sequential scan. Unfortunately, the scheme is not secure against statistical analysis across multiple queries and can leak the positions of the queried keywords in a document. The scheme has to use fix-sized words and the complexity of the encryption and search is linear in the number of words. Also it is not compatible with existing file encryption standards and has to use their specific encryption method which can be used only for plaintext data and not for example on compressed data.

Some of the above problems are addressed by Goh [10] by introducing a Bloom filter index to each document. The index makes the scheme independent of the document encryption. Goh also introduced the formal indistinguishability against chosen keyword attack (IND-CKA) and a slightly stronger IND-CKA2 adversary model.

Chang and Mitzenmacher [7] developed two index schemes, similar to Goh [10], using pre-build dictionaries. Their search schemes are independent of the encryption method and use one index per document.

Curtmola et al. [8] propose new adversarial models for searchable encryption: a non-adaptive and an adaptive one. They construct two schemes which are provably secure in these new models. The first scheme (SSE-I) is only secure against non-adaptive adversaries, but more efficient than the second scheme (SSE-II), which is also secure against adaptive adversaries.

Our Contribution. In this paper we present the first conjunctive wildcard search scheme in the symmetric setting. The scheme is proven secure against adaptive adversaries.



(a) normal (b) alternative

Fig. 1. Bloom filter with storage.

Structure. The rest of the paper is organized as follows. Section 2 describes the building blocks necessary for our constructions. We summarize the security definitions from Curtmola et al. [8] in Section 3. For sake of simplicity we explain an easy basic search scheme in Section 4.1, before we describe our new masked index scheme in Section 4.2. We then present the wildcard add-on for our search scheme in Section 5. Section 6 analyses the security of our masked scheme and in Section 7 we take a look at the efficiency of the constructions. We conclude the paper in Section 8.

2 Preliminaries

Bloom filters. A Bloom filter (BF) [3] is a data structure which is used to answer set membership queries. It is represented as an array of b bits which are initially set to 0. In general the filter uses r independent hash functions h_t, where $h_t : \{0,1\}^* \rightarrow [1,b]$ for $t \in [1,r]$, each of which maps a set element to one of the b array positions. For each element e in the set $\mathcal{S} = \{e_1, \ldots e_m\}$ the bits at positions $h_1(e), \ldots, h_r(e)$ are set to 1. To check whether an element x belongs to the set $\mathcal{S}$, we check if the bits at positions $h_1(x), \ldots, h_r(x)$ are set to 1. If so, x is considered a member of set $\mathcal{S}$. Bloom filters have a possibility of false positives, because the positions of an element may have been set by one or more other elements. With appropriate parameters the false positive probability can be reduced to a desired error rate.

Instead of r different hash functions we use a single HMAC-SHA1 [2, 13] function with r different and independent keys to create a trapdoor. This allows only legitimate users in possession of the keys to construct the correct Bloom filter and thus to add and search documents on a server.

In our constructions we use a Bloom filter with storage, as introduced by Boneh et al. [5]. Figure 1 shows two different versions of a Bloom filter with storage. Our constructions use the type (b).

Pseudorandom generators. A pseudorandom bit generator $g : \{0,1\}^\alpha \rightarrow \{0,1\}^\beta$ is a deterministic algorithm which, given a seed of length α, outputs a binary sequence of length $\beta \gg \alpha$ that is computationally indistinguishable from a random string.

Notation. Throughout this paper we use the following notation. Let $\mathcal{D}$ be a document collection $\mathcal{D} = \{d_{id_1}, \ldots, d_{id_n}\}$, consisting of n documents. The size of a document d_{id} is denoted $|d_{id}|$, where id is a unique document identifier. Each document d_{id} consists of a set of words W_{id}. Let $\Delta_{id} = u(d_{id})$ be a dictionary of distinct words in a document d_{id}. The function $u(\cdot)$ extracts the unique words of a document. The number of distinct words per document is denoted by $|\Delta_{id}|$. We refer to $\mathcal{D}(w)$ as all the document ids containing word w and the sequence $\mathcal{D}(w_1), \ldots, \mathcal{D}(w_c)$ as the access pattern of a client.

3 Definitions

Index scheme. Our index schemes consist of the following four algorithms:

- $\mathsf{Keygen}(s)$: Given a security parameter s, Keygen outputs the master private key $\mathcal{K}$. This algorithm is run by the client.
- $\mathsf{BuildIndex}(\mathcal{K}, \mathcal{D})$: Given the master key $\mathcal{K}$ and a document collection $\mathcal{D}$, the algorithm outputs an index $\mathcal{I}$. This algorithm is run by the client.
- $\mathsf{Trapdoor}(\mathcal{K}, w)$: Given the key $\mathcal{K}$ and a keyword w, $\mathsf{Trapdoor}$ outputs the trapdoor T_w for w. This algorithm is run by the client.
- $\mathsf{SearchIndex}(T_w, \mathcal{I})$: Given a trapdoor T_w for word w and the index $\mathcal{I}$, the algorithm outputs a bit string which indicates the matched documents. This algorithm is run by the server.

3.1 Security definitions

We use the security definitions for searchable symmetric encryption (SSE) from Curtmola et al. [8] which we summarize in this section. For detailed information we refer to the original paper [8].

Before stating the security definition for semantic security for SSE, we introduce three auxiliary notions: the *history*, the *view* and the *trace*.

History. The *history* defines the user input to the scheme. It is an interaction between the client and the server, which is determined by a document collection and a set of words that the client wishes to search for (and that we wish to hide from the adversary).

Definition 1. (History). *A* history H_u*, is an interaction between a client and a server over* u *queries, consisting of a document collection* $\mathcal{D}$ *and the keywords* w_i *used for* u *consecutive search queries. The partial history* H_u^τ *of a given history* $H_u = (\mathcal{D}, w_1, \ldots, w_u)$*, is the sequence* $H_u^\tau = (\mathcal{D}, w_1, \ldots, w_\tau)$*, where* $\tau \leq u$*.*

View. The server's *view* consists of all the information the server can gather during a protocol run. In particular, the view will consist of the index (of the document collection) and the trapdoors (of the queried words). It will also contain some additional common information, such as the number of documents in

the collection and their ciphertexts. However the view should not reveal any information about the history besides the outcome and the pattern of the searches. Let $\mathcal{I}$ be the index of a document collection generated under key K, and T_{w_i}, $1 \leq i \leq u$, be the trapdoors for the words w_i queried in H_u.

Definition 2. (View). *Let $\mathcal{D}$ be a collection of n documents and $H_u = (\mathcal{D}, w_1, \ldots, w_u)$ be a history over u queries. An adversary's* view *of H_u under secret key K is defined as*

$$V_K(H_u) = (id_1, \ldots, id_n, E(d_{id_1}), \ldots, E(d_{id_n}), \mathcal{I}, T_{w_1}, \ldots, T_{w_u}) .$$

The partial view $V_K^\tau(H_u)$ of a history H_u under secret key K is the sequence

$$V_K^\tau(H_u) = (id_1, \ldots, id_n, E(d_{id_1}), \ldots, E(d_{id_n}), \mathcal{I}, T_{w_1}, \ldots, T_{w_\tau}) ,$$

where $\tau \leq u$.

Note that K refers only to the secret key for the SSE scheme and not to the encryption key of the documents.

Trace. The *trace* consists of exactly the information we are willing to leak or that the server is allowed to learn. This information includes the identifiers of the documents that contain each query word in the history and information that describes which trapdoors in the view correspond to the same underlying words in the history. The encrypted documents are also stored on the server, so the document sizes and identifiers will be leaked. We add also the sequence $(\mathcal{D}(w_1), \ldots, \mathcal{D}(w_n))$ which denotes the *access pattern* of a client and the *search pattern* Π_u of a client as any information that can be derived from knowing whether two arbitrary searches were performed for the same word or not to the trace. More formally, Π_u can be thought of as a symmetric binary matrix where $\Pi_u[i, x] = 1$ if $w_i = w_x$, and 0 otherwise, for $1 \leq i, x \leq u$.

Definition 3. (Trace). *Let $\mathcal{D}$ be a collection of n documents and $H_u = (\mathcal{D}, w_1, \ldots, w_u)$ be a history over u queries. The* trace *of H_u is the sequence*

$$Tr(H_u) = (id_1, \ldots, id_n, |d_{id_1}|, \ldots, |d_{id_n}|, \mathcal{D}(w_1), \ldots, \mathcal{D}(w_u), \Pi_u) .$$

Semantic security. We now present the simulation-based definition for semantic security from Curtmola et al. [8]. We assume that the client initially stores a number of documents and afterwards performs an arbitrary number of search queries. For all queries $0 \leq \tau \leq u$, we require the simulator, given only a partial trace of the history, to simulate the adversary on a partial view of the same history.

Definition 4. (Adaptive Semantic Security for SSE). *An SSE scheme is adaptively semantically secure if for all $u \in \mathbb{N}$ and for all (non-uniform) probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a (non-uniform) probabilistic polynomial-time algorithm (the simulator) $\mathcal{S}$ such that for all traces Tr_u of*

length u, all polynomially samplable distributions $\mathcal{H}_u$ over $\{H_u : Tr(H_u) = Tr_u\}$ (i.e., the set of histories with trace Tr_u), all functions $f : \{0,1\}^m \to \{0,1\}^{v(m)}$ (where $m = |H_u|$ and $v(m) = \mathsf{poly}(m)$), all $0 \leq y \leq u$ and all polynomials p and sufficiently large s:

$$|Pr\left[\mathcal{A}\left(V_K^\tau(H_u)\right) = f(H_u^\tau)\right] - Pr\left[\mathcal{S}\left(Tr(H_u^\tau)\right) = f(H_u^\tau)\right]| < \frac{1}{p(s)},$$

where $H_u \xleftarrow{R} \mathcal{H}_u, K \leftarrow \mathsf{Keygen}(s)$, and the probabilities are taken over $\mathcal{H}_u$ and the internal coins of Keygen*, $\mathcal{A}$, $\mathcal{S}$ and the underlying* BuildIndex *algorithm.*

4 Constructions

In this section we describe two index based constructions similar to Goh [10]. We first introduce the basic construction, which stores a keyed Bloom filter index on an untrusted server. The second construction stores a masked index on the server side. We refer to our basic search scheme as (B) and to our masked index search scheme as (M). Both constructions use a Bloom filter per document. The index can thus be represented as an $n \times b$ binary matrix where n is the number of documents and b the size of a single Bloom filter in bits. We use the words index and matrix interchangeably.

4.1 The Basic Index Scheme

To create a searchable index, we use one Bloom filter per document. We insert all distinct words of a document d_{id} in its Bloom filter BF_{id} by applying the HMAC-SHA1 function r times with r independent keys on each distinct word. All the BFs and the encrypted documents are then stored on the server.

To search in the database a trapdoor is required. This trapdoor for finding a specific keyword w in the database is derived by applying r times HMAC-SHA1 on the keyword to search for. The outcome of each HMAC-SHA1 denotes a specific position in a Bloom filter. After receiving the trapdoor, the server looks up the columns of the index specified in the trapdoor, handles them as bit strings and computes the bitwise AND on the columns. The resulting bit string indicates a match with a 1 and a non-match with a 0.

Our construction consists of the following four algorithms:

$\mathsf{Keygen}(s)$: Given a security parameter s, generate a secret master key $K = \{k_1, \ldots, k_r\}$, consisting of r independent secret keys.

$\mathsf{Trapdoor}(K, w)$: Given the key $K = \{k_1, \ldots, k_r\}$ and a word w, calculate the positions $p_t = h_{k_t}(w)$ for $t \in [1, r]$ in a Bloom filter and output the trapdoor $T_w = \{p_1, \ldots, p_r\}$, where $p_t \in [1, b]$.

$\mathsf{BuildIndex}(K, \mathcal{D})$: The input is the master secret key K and a document collection $\mathcal{D}$ comprising of a set of n documents.

1. For each $id \in [1, n]$, create the list of unique words $\Delta_{id} = u(d_{id})$ and compute for each word $w_i \in \Delta_{id}$:
   (a) the trapdoor: $T_{w_i} = \{p_1, \ldots, p_r\}$
   (b) and set the bits at the positions T_{w_i} in BF_{id} to 1.
2. Output the index $\mathcal{I} = (\mathrm{BF}_1, \ldots, \mathrm{BF}_n)^T$.

We define $\mathcal{I}[p]$ as the column vector $[\mathrm{BF}_{id}[p]]_{id \in [1,n]}$ of the matrix $\mathcal{I}$.

$\mathsf{SearchIndex}(T_w, \mathcal{I})$: Given the trapdoor $T_w = \{p_1, \ldots, p_r\}$ for word w and the index $\mathcal{I}$, take the set of columns $\{\mathcal{I}[p_t]\}_{t \in [1,r]}$ of the matrix $\mathcal{I}$. Consider each column as a bit string and output the bitwise AND of the columns.

It is easy to see, that this construction is vulnerable to correlation attacks which leak the similarity of documents upfront. This is because each word is represented by the same r positions in all Bloom filters. Another disadvantage of Bloom filters is the fact, that the number of 1's is dependent on the number of entries, in this case the number of distinct keywords per document. As a consequence, the scheme gives a good guess on the number of keywords in each document. To conceal this information we can use padding, where we add random strings to a documents distinct word list, so that the number of entries per BF is equal. To gain a higher level of security we mask the index before it is stored on an untrusted server as seen in the next section.

4.2 The Masked Index Scheme

To mask the index we use a pseudorandom generator $g(K_G, p, id)$ which takes a secret generator key K_G and the exact position of the bit to mask in the matrix (p, id) as input.

Our construction consists of the following four algorithms:

$\mathsf{Keygen}(s)$: Given a security parameter s, generate a secret master key $K = \langle K_H, K_G \rangle$, with $K_H = \{k_t\}_{t \in [1,r]}$ being r independent keys to compute the HMAC and $K_G \in \{0,1\}^*$ the key for the pseudorandom generator.

$\mathsf{Trapdoor}(K_H, w)$: Given the key $K_H = \{k_1, \ldots, k_r\}$ and a word w, output the trapdoor $T_w = \{p_1, \ldots, p_r\}$, where $p \in [1, b]$.

$\mathsf{BuildIndex}(K, \mathcal{D})$: The input is the master secret key K and a document collection $\mathcal{D}$ comprising of a set of n documents.

1. For each $id \in [1, n]$, create the list of unique words $\Delta_{id} = u(d_{id})$ and compute for each word $w_i \in \Delta_{id}$:
   (a) the trapdoor: $T_{w_i} = \{p_1, \ldots, p_r\}$
   (b) and set the bits at the positions T_{w_i} in BF_{id} to 1.
2. Create the index $\mathcal{I} = (\mathrm{BF}_1, \ldots, \mathrm{BF}_n)^T$.
3. For each position $\mathrm{BF}_{id}[p]$, with $p \in [1, b]$, compute $g(K_G, p, id)$ and create the masked index

$$\mathcal{M}[p][id] = \{\mathcal{I}[p][id] \oplus g(K_G, p, id)\}\,.$$

4. Output the masked index $\mathcal{M}$.

$\mathsf{SearchIndex}(T_w, \mathcal{M})$: Given the trapdoor $T_w = \{p_1, \ldots, p_r\}$ for word w and the masked index $\mathcal{M}$, send the set of columns $\{\mathcal{M}[p_t]\}_{t\in[1,r]}$ of the matrix $\mathcal{M}$ to the client. The client computes the set of decrypted columns

$$\mathcal{I}[p_t][id] = \{\mathcal{M}[p_t][id] \oplus g(K_G, p_t, id)\}_{t\in[1,r]}\,,$$

and outputs the bitwise AND of the columns. The resulting bit vector indicates the matched documents.

This interactive construction allows a user to decide which documents from the list of matched documents she wants to retrieve. By having a two-round protocol the scheme becomes more flexible. This is comparable with an internet search, where the search engine gives a list of results and the user can decide, which sites to download/visit. Most of the times not all of the matching documents are interesting for a user. By downloading only the desired documents, instead of all the matched documents, we do not produce unnecessary traffic. This is important for mobile users with limited bandwidth or expensive data usage fees.

4.3 Properties

Boolean queries. As an additional feature our scheme supports conjunctive search queries, which means a boolean AND combination of two or more keywords. This is done by sending the union of several trapdoors to the server, which then sends back the result associated to those keywords. The resulting bit string indicates the documents including all of the searched words.

Secure updates. Our search scheme supports efficient and secure updates on a document collection $\mathcal{D}$, in the sense that the client is able to Add and Delete documents from the database. A document is added by simply running the BuildIndex algorithms with the new documents as input. The resulting index can then be appended to the existing index stored on a server. To delete a document in our constructions, the document and the corresponding row of the index can be deleted from the server.

Adding a document can be done with the following algorithm:

$\mathsf{Add}(K, \mathcal{D})$: This Algorithm is equal to BuildIndex. The resulting index is appended to the existing index.

Deleting a document in the unmasked index scheme can be done with the following algorithm:

$\mathsf{Delete}(id)$: Given a document id, delete d_{id} and BF_{id} from the server.

Complexity. The complexity of an update operation (add, delete) depends on the number of documents processed. The communication overhead is $O(n)$, where n is the number of documents $\mathcal{U}$ wants to add. Per document $\mathcal{U}$ has to transfer the Bloom filter of size b bit to the server. To delete a document in our schemes, $\mathcal{U}$ can simply delete the document and the corresponding row from the index by sending the *id* to the server.

Security. During an update operation, $\mathcal{U}$ reveals only the number of documents processed. The newly added BFs look like random strings.

5 Wildcard Add-on

In this section we introduce a simple wildcard add-on that can be used with most search schemes. The main idea behind our wildcard search is to pre-process the words that will be inserted into the index: for each distinct word we create all the wildcardified variants of the word as shown in Algorithm 1. The individual characters of a word are denoted by $w_i[j]_{j\in[1,\lambda]}$ where $w_i[x : y]$ denotes the characters x to y.

For example, the keyword `flower` will be represented in a single wildcard scheme as {`flower, *flower, flower*, *lower, ..., flowe*, *ower, ..., flow*, *wer, f*er, fl*r, flo*, *er, f*r, fl*, *r, f*`}. Thus all possible variations of the word are created.

The number of all these single wildcard combinations per distinct word is computed by

$$\left(\sum_{j=1}^{\lambda} j\right) + 2 = \frac{\lambda(\lambda+1)}{2} + 2,$$

where λ is the length of the word w_i.

Algorithm 1 Algorithm for Wildcardifying Words

Input: A word w_i
Output: Ω_i: all wildcardified versions of the word w_i

```
wild_carded_words = [w_i, *w_i, w_i*]
for wild_card_size = 1 to wordLength do
    for v = 1 to (wordLength - wild_card_size +1) do
        wild_carded_words.append(w_i[1 : v] + * + w_i[v + wild_card_size : λ])
    end for
end for
```

For our wildcard search scheme we insert not only the keywords, but all the wildcardified versions of a word into the index. Hence the scheme transforms the problem of a wildcard search into a lookup for an exact match. The scheme still supports conjunctive search queries.

Example 1. A search for the word chin* will return all the document ids containing a word starting with chin*, like china, chinatown, chinaware, chinchilla, chine, chinese, chinked, chinless, ...

Multiple wildcards. If desired, it is possible to add multiple wildcards at the cost of more pre-processing and server storage space. Thus it is possible, to add the support for two or more wildcards (e.g., `*owe*` or `f*o*r`).

6 Security Proof

We now provide the security proof for our masked index scheme. At this point we do not take updates and conjunctive queries into account. HMAC-SHA1 is used as the hash function for the Bloom filter. Bellare [1] proved, that HMAC is a pseudorandom function.

Theorem 1. *If h and g are secure pseudorandom functions, our masked search scheme described in Section 4.2 including the wildcard add-on explained in Section 5 is an adaptively secure SSE scheme.*

Proof. Let $u \in \mathbb{N}$, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. We describe a probabilistic polynomial-time simulator $\mathcal{S}$ such that for all polynomially-bounded functions f and all distributions $\mathcal{H}_u$, $\mathcal{S}$ can simulate the partial view of an adversary $\mathcal{A}(V_K^\tau(H_u))$ given only the trace of a partial history $Tr(H_u^\tau)$ for all $0 \leq \tau \leq u$ with probability negligibly close to 1. For all $0 \leq \tau \leq u$, we show that $\mathcal{S}(Tr(H_u^\tau))$ can generate a simulated view $\mathfrak{V}_u^\tau$ that is indistinguishable from $V_K^\tau(H_u)$. Let

$$Tr(H_u^\tau) = (id_1, \ldots, id_n, |d_{id_1}|, \ldots, |d_{id_n}|, b, \mathcal{D}(w_1), \ldots, \mathcal{D}(w_\tau), \Pi_\tau).$$

be the trace of an execution after τ search queries and let H_u be a history consisting of u search queries such that $Tr(H_u) = Tr_u$. The simulator $\mathcal{S}$ works as follows:

With the information from the trace, $\mathcal{S}$ chooses n random values $R_1, \ldots, R_n$ such that $|R_i| = |d_i|$ for all $i = 1, \ldots, n$. $\mathcal{S}$ also includes the document identifiers, known from the trace, in the partial view. Then the simulator $\mathcal{S}$ generates a simulated index $\mathfrak{M} = (B_1, \ldots, B_n)^T$ with random $B_i \in \{0,1\}^b$, for $i \in [1, n]$. $\mathfrak{M}$ will be included in all partial views $\mathfrak{V}_u^\tau$ used to simulate $\mathcal{A}$. Next $\mathcal{S}$ simulates the trapdoor for query $\tau, (1 \leq \tau \leq u)$ in sequence. If $\Pi_\tau[j, \tau] = 1$ for some $1 \leq j < \tau$ set $\mathfrak{T}_\tau = \mathfrak{T}_j$. Otherwise $\mathcal{S}$ picks a random value rnd, calculates $p_t = h_{k_t}(rnd)$ for $t \in [1, r]$ and sets $\mathfrak{T}_\tau = \{p_1, \ldots, p_r\}$, such that for $1 \leq j < \tau, \mathfrak{T}_\tau \neq \mathfrak{T}_j$. Then $\mathcal{S}$ constructs for all τ a simulated view

$$\mathfrak{V}_u^\tau = (id_1, \ldots, id_n, E(R_1), \ldots, E(R_n), \mathfrak{M}, \mathfrak{T}_1, \ldots, \mathfrak{T}_\tau)\,,$$

and eventually outputs $\mathcal{A}(\mathfrak{V}_u^\tau)$. We now claim that $\mathfrak{V}_u^\tau$ is indistinguishable from

$$V_K^\tau(H_u) = (id_1, \ldots, id_n, E(d_{id_1}), \ldots, E(d_{id_n}), \mathcal{M}, T_1, \ldots, T_\tau)\,.$$

Therefore we state that for all $i \in [1,n]$, id_i in $V_K^0(H_u)$ and $\mathfrak{V}_u^0$ are identical and thus indistinguishable. Also, $E(\cdot)$ is a semantically secure encryption algorithm, thus $E(d_i)$ is indistinguishable from $E(R_i)$ of the same length. Given the BuildIndex algorithm, it is clear that $\mathfrak{M}$ is indistinguishable from $\mathcal{M}$. Otherwise one could distinguish between a random string B of size b and $[BF[p] \oplus g(K_G, p)]_{p \in [1,b]}$, the bitwise XOR of a Bloom filter of size b and the output of the pseudorandom generator $g(\cdot)$. It is easy to see that the trapdoors are indistinguishable, otherwise one could distinguish $h_k(\Omega)$ from $h_k(rnd)$. Thus, $\mathfrak{V}_u^\tau$ is indistinguishable from $V_K^\tau(H_u)$, for all $0 \leq \tau \leq u$. □

Updates. In our scenario, intermixing queries and updates is equivalent to first update and then query. Incremental updates can be aggregated to one BuildIndex over a larger document set. Thus we can proceed with the proof of Theorem 1.

Conjunctive queries. The above proof also holds if we search for a conjunctive set of words. Imagine that we substitute the words w_0 and w_1 with the two sets $(w_{0,1}, \ldots, w_{0,l})$ and $(w_{1,1}, \ldots, w_{1,l})$. Then proceed with the proof of Theorem 1.

Stronger security. Note that to gain a higher level of security it is possible to split the large Bloom filter into several smaller parts and store the parts on different non communicating servers. Another way of increasing the security by not revealing the access pattern is to store the index and the documents on two different non communicating servers.

7 Performance

We now consider the efficiency of our constructions where the efficiency is measured in terms of the computation, communication and space complexity.

Computational complexity. Table 2 shows the efficiency of our schemes compared to others in terms of computational complexity. The Trapdoor, BuildIndex and SearchIndex columns describe the computational complexity of the algorithms. The column Server gives the computation on the server side, whereas Client shows the computational complexity on the client.

The Trapdoor algorithm is a constant time operation. The BuildIndex algorithm has to process each distinct word per document. Thus the complexity is $O(n|\Delta|)$. Because the index is stored as a Bloom filter with storage (see Figure 1(b)), the SearchIndex algorithm is a simple table lookup and takes time $O(1)$. However the table lookup does not give us the result of the query. Thus after a basic index search the server has to compute a bitwise AND of the matrix columns labelled by the trapdoor, which is a $O(n)$ operation. With the masked index the server is not able to compute the result and has to send the matrix columns to the client, which is then able to decrypt the columns by an XOR operation and then performs the AND on the unmasked columns. Thus the client

computation is $O(n)$. Note that in both of our schemes the server and client computations are AND and/or XOR operations and thus are efficient. Table 2 shows that in the big O notation the scheme of Curtmola et al. is more efficient than our schemes but in practice n AND operations are more efficient than $|\mathcal{D}(w)|$ encryptions.

The scheme of Song et al. (SWP) [16] does not use an index. Thus the BuildIndex field is marked with a "-". The SearchIndex algorithm denotes the search through the encrypted documents. In the SWP scheme all the words per document have to be searched and so the complexity is $O(nq)$ where n is the number of documents and the q the number of words per document.

Communication and space complexity. Table 3 compares the space complexity of different search schemes. Index describes the storage space for the index on the server side. The Trapdoor column shows the size of a trapdoor and the Result column describes the size of the results that have to be transferred to the client. In both of our schemes the index can be seen as a $n \times b$-matrix, where n is the number of documents and b the size of the Bloom filter. Thus the server has to store nb bits, where b is a constant. The trapdoor has a constant size $O(1)$ and the size of the result vector is $O(n)$ because it is dependent on the number of documents in the database.

Example 2. Assume a user who wants to search for 1000 keywords. The average word length in the English language is 5 characters per word. Thus we end up with 17,000 wildcardified words. To achieve a false positive rate of 0.01, we set k = 6 and b = 153000. The resulting sizes for different document collections can be found in Table 1.

8 Conclusion

We examined the problem of wildcard searches over encrypted data in the symmetric setting and proposed a searchable encryption scheme similar to Goh [10] which supports wildcards that can be either any single character or a string of characters inside a word. The scheme also supports conjunctive search queries with any number of keywords. We proposed two variants of our scheme which differ in the security of the index and the communication overhead. The first scheme is more efficient in terms of computation and communication, while the second scheme is more secure in the sense that we leak less information about the index. Our masked scheme is proven secure against adaptive adversaries. Our schemes are more efficient than previous search schemes and are practical to use in real world applications.

References

1. Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO '06, 26th*

Table 1. Number example of communication and space complexity. Parameters: Keywords to search = 1000, average word length = 5, FP-rate = 0.01, k = 6, b = 153000. CM: t = 2030, *depending on the number of matched documents. SSE: max = 500, p = 160.

Scheme	Documents	Index	Trapdoor	Result
Goh [10]	1000	18.24 MB	960 b	750 B
	5000	91.21 MB	960 b	3.66 kB
	10000	182.42 MB	960 b	7.32 kB
CM2 [7]	1000	3.9 MB	2542 b	*
	5000	19.5 MB	2542 b	*
	10000	39 MB	2542 b	*
SSE-2 [8]	1000	95.37 MB	80000 b	625 B
	5000	619.9 MB	80000 b	813 B
	10000	1.3 GB	80000 b	875 B
Our	1000	18.24 MB	108 b	750 B
	5000	91.21 MB	108 b	3.66 kB
	10000	182.42 MB	108 b	7.32 kB

Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006.

2. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.
3. Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.
4. Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public Key Encryption with Keyword Search. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.
5. Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith III. Public Key Encryption that Allows PIR Queries. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2007.
6. Dan Boneh and Brent Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In Salil P. Vadhan, editor, *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, volume 4392 of *Lecture Notes in Computer Science*, pages 535–554. Springer, 2007.
7. Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *In Proceedings of 3rd Applied Cryp-*

Table 2. Computational performance of different search schemes, where n is the number of documents in the database and q the number of words per document. The number of distinct words per document is denoted by $|\Delta|$ and $|\mathcal{D}(w)|$ denotes the number of documents containing the keyword w. The number of all distinct words in the database is denoted by $|W_{db}|$. The asterisk * denotes a bitwise AND and/or XOR. The two asterisks ** refer to the use of a so-called *FKS dictionary* introduced by Fredman et al. [9].

Scheme	Trapdoor	BuildIndex	SearchIndex	Server	Client
SWP [16]	$O(1)$	$O(nq)$	$O(nq)$	$O(nq)$	$O(\|\mathcal{D}(w)\|q)$
Goh [10]	$O(1)$	$O(n\|\Delta\|)$	$O(n)$	$O(n)$	$O(1)$
CM2 [7]	$O(\log \|W_{db}\|)$	$O(n\|\Delta\|)$	$O(n)$	$O(n)$	$O(1)$
SSE-1 [8]	$O(1)$	$O(n\|\Delta\|)$	$O(1)^{**}$	$O(\|\mathcal{D}(w)\|)$	$O(1)$
Our (M)	$O(1)$	$O(n\|\Delta\|)$	$O(1)$	$O(1)$	$O(n)^*$

Table 3. Communication and space complexity of different search schemes, where n is the number of documents in the database and $|\mathcal{D}(w)|$ the number of documents containing the keyword w. The total size of the plaintext document collection in units, where a unit is the smallest possible size for a word, is denoted by m and the number of all distinct words in the database is denoted by $|W_{db}|$.

Scheme	Index	Trapdoor	Result
SWP [16]	-	$O(1)$	$O(\|\mathcal{D}(w)\|)$
Goh [10]	$O(n)$	$O(1)$	$O(n)$
CM2 [7]	$O(n)$	$O(1)$	$O(n)$
SSE-1 [8]	$O(m) + O(\|W_{db}\|)$	$O(1)$	$O(\|\mathcal{D}(w)\|)$
Our (M)	$O(n)$	$O(1)$	$O(n)$

tography and Network Security Conference (ACNS), pages 442–455, 2005.
8. Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS '06: Proceedings of the 13th ACM conference on Computer and Communications Security*, pages 79–88, New York, NY, USA, 2006. ACM.
9. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM*, 31(3):538–544, 1984.
10. Eu-Jin Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003.
11. Oded Goldreich. Secure Multi-Party Computation. Working draft, October 2002.
12. Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
13. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC2104, Internet Engineering Task Force (IETF), Februar 1997.
14. Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing, 14-16 May 1990, Baltimore, Maryland, USA*, pages 514–523. ACM, 1990.
15. Rafail Ostrovsky. *Software Protection and Simulations on Oblivious RAMs*. PhD thesis, MIT, 1992.

16. Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–55, Washington, DC, USA, 2000. IEEE Computer Society.