Private Stream Search at the Same Communication Cost as a Regular Search

Matthieu Finiasz and Kannan Ramchandran
A stream search consists in filtering data according to a set of keywords:

- the data is a stream (it could also be a database)
  - every piece of data is treated independently
- the filtering is done externally
  - you retrieve only the matching data.

A typical scenario is Google Alerts:

- get an alert for each new page matching your interests.

Private Stream Search does this without revealing the keywords (your interests) to the filtering server.
Private Stream Search
Why is it useful?

- Protect your privacy:
  - Google Alerts,
  - web search in general...
- Protect your financial interests:
  - when searching for patents,
    → reveals what your company is focusing on.
- Global surveillance systems:
  - search for keywords in emails.
- But PSS is only worth it if it is efficient:
  - no one is ready to lose efficiency for privacy...
To preserve privacy, the user sends a masked query:
- a public list of possible keywords is needed,
- the query is an encrypted selection of keywords.

The server filters according to the encrypted query:
- all documents/all keywords are treated symmetrically,
- it accumulates matches in an encrypted data buffer,
- only the user can extract the matches.

PSS requires computations on encrypted data:
- this is possible using homomorphic encryption.
Paillier’s encryption function $E$ has the properties:

$$E(m + m') = E(m)E(m'),$$
$$E(c \times m) = E(m)^c.$$ 

→ possible to perform operations on encrypted values.

A fully homomorphic scheme would also allow to compute $E(m \times m')$ from $E(m)$ and $E(m')$.

Paillier’s cryptosystem is semantically secure:

× impossible to distinguish $E(0)$ and $E(1)$. 

Homomorphic Encryption
Paillier’s Cryptosystem
Requirements for this scheme:
- a public dictionary of keywords $\Omega = \{k_1, \ldots, k_{|\Omega|}\}$,
- the users asks OR queries on words of $\Omega$,
- a database/stream of $t$ documents $(f_1, \ldots, f_t)$,
- the users has an estimate of the number $m$ of matches.

We consider an example with:

$$\Omega = \text{dog brown cat black bird white}$$

$${f_1 = "the dog is black", f_3 = "the bird is white", f_2 = "the cat is white", f_4 = "the bird is black"}$$
The user wants to query “cat OR white”,
he computes a tuple $Q$ of $\epsilon(0)$ and $\epsilon(1)$ accordingly.
The First PSS Scheme
Query Execution

\[ \Omega = \text{dog brown cat black bird white} \]
\[ Q = \varepsilon(0) \varepsilon(0) \varepsilon(1) \varepsilon(0) \varepsilon(0) \varepsilon(1) \]

SERVER

buffer \( B \)

- The server prepares a response buffer \( B \),
- the matches will be accumulated in \( B \).
\[ \Omega = \text{dog brown cat black bird white} \]

\[ Q = \mathcal{E}(0) \quad \mathcal{E}(0) \quad \mathcal{E}(1) \quad \mathcal{E}(0) \quad \mathcal{E}(0) \quad \mathcal{E}(1) \]

\[ f_1 = \text{"the dog is black"} \quad \mathcal{E}(0+0) \]

For every document \( f_i \), the server computes:

\[ \mathcal{E}(c_i) = \prod_{k,j \in f_i} q_j. \]

\( c_i \) is the number of matching keywords in \( f_i \).
**The First PSS Scheme**

**Query Execution**

\[ \Omega = \text{dog brown cat black bird white} \]

\[ Q = \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \]

\[ f_1 = "\text{the dog is black}" \]

\[ \mathcal{E}((0+0)f_1) \]

\[ \text{buffer } B \]

\[ \times \text{For every document } f_i:\]

\[ \times \text{the server “adds” } \mathcal{E}(c_i)f_i = \mathcal{E}(c_if_i) \text{ randomly in } B. \]
The server repeats this for all documents.
\[ \Omega = \text{dog brown cat black bird white} \]

\[ Q = \varepsilon(0) \varepsilon(0) \varepsilon(1) \varepsilon(0) \varepsilon(0) \varepsilon(1) \]

\[ f_3 = \text{"the bird is white"} \]

\[ \varepsilon((0+1)f_3) \]

\[ \varepsilon(2f_2) \varepsilon(0f_1) \varepsilon(1f_3) \varepsilon(1f_3) \varepsilon(2f_2) \varepsilon(0f_1) \]

\textbf{buffer B}

\textbf{The server repeats this for all documents.}
$\Omega = \text{dog brown cat black bird white}$

$Q = \mathcal{E}(0) \ \mathcal{E}(0) \ \mathcal{E}(1) \ \mathcal{E}(0) \ \mathcal{E}(0) \ \mathcal{E}(1)$

$f_4 = \text{"the bird is black"}$

$\mathcal{E}((0+0)f_4)$

$\mathcal{E}(0f_4) \ | \ \mathcal{E}(2f_2) \ | \ \mathcal{E}(0f_1) \ | \ \mathcal{E}(1f_3) \ | \ \mathcal{E}(1f_3) \ | \ \mathcal{E}(2f_2) \ | \ \mathcal{E}(0f_4) \ | \ \mathcal{E}(0f_1)$

buffer $B$

※ The server repeats this for all documents.
\[ \Omega = \text{dog brown cat black bird white} \]
\[ Q = \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \]

\[ \text{buffer } B \]

\[ \mathcal{E}(0f_4) \mathcal{E}(2f_2) \mathcal{E}(0f_1) \mathcal{E}(1f_3) \mathcal{E}(2f_2) \mathcal{E}(0f_4) \mathcal{E}(0f_1) \]

\[ \times \text{ The user receives the encrypted buffer } B. \]
The First PSS Scheme

Extraction of Results

\[ \Omega = \text{dog brown cat black bird white} \]
\[ Q = \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \]

\[
\begin{array}{cccccc}
2f_2 & 0 & f_3 & f_3 + 2f_2 & 0 & 0 \\
\mathcal{E}(0f_4) & \mathcal{E}(0f_1) & \mathcal{E}(1f_3) & \mathcal{E}(1f_3) & \mathcal{E}(2f_2) & \mathcal{E}(0f_4) \\
\mathcal{E}(2f_2) & \mathcal{E}(0f_1) & \mathcal{E}(1f_3) & \mathcal{E}(2f_2) & \mathcal{E}(0f_4) & \mathcal{E}(0f_1) \\
\end{array}
\]

buffer \( B \)

\[ \text{The user receives the encrypted buffer } B, \]
\[ \text{he decrypts it.} \]
\[ \Omega = \text{dog brown cat black bird white} \]
\[ Q = E(0) \ E(0) \ E(1) \ E(0) \ E(0) \ E(1) \]

"the cat is white"  

"the bird is white"

\[ 2f_2 \quad \times \quad f_3 \quad \times \quad f_3 + 2f_2 \quad \times \quad \times \]

\[ \begin{array}{cccccc}
E(0f_4) & E(0f_1) & E(1f_3) & E(1f_3) & E(0f_4) & E(0f_1) \\
E(2f_2) & E(0f_1) & E(1f_3) & E(2f_2) & E(0f_4) & E(0f_1) \\
\end{array} \]

buffer \( B \)

\[ \times \text{The user receives the encrypted buffer } B, \]
\[ \times \text{he gets one document for each singleton}. \]
What can be improved in this scheme?

- **Computations:**
  - PSS requires one operation for each message,
  - difficult to improve,
    - requires more efficient homomorphic encryption.

- **Communications:**
  - the query is linear in the dictionary size,
    - fully homomorphic encryption could help,
  - the reply is linear in the buffer size,
    - the buffer size should be the number of matches.

- In the Ostrovsky-Skeith scheme, the size is $O(m \log m)$,
  - Bethencourt *et al.* and Danezis-Díaz improve this.
Our Contribution

- Take an information theory look at the problem:
  - the server computes $\mathcal{E}(c_if_i)$ an encrypted sparse vector
  - the problem is to compress it,
  - possible to compute it’s syndrome for any linear code
  - compatible with homomorphic encryption.

- We propose two different approaches:
  - Using Reed-Solomon codes,
    - allows a “zero-error” guarantee (if $m$ is known),
    - computationally heavy at the server,
  - Using irregular LDPC codes,
    - gives optimal asymptotic performances.
Using LDPC codes

- To obtain an optimal reply size:
  - the user should only get the documents,
  - changing their order should not change the syndrome.

- Each document defines its own parity check column:
  - use the document as a seed to a PRNG,
  - use the PRNG to generate a “random” LDPC column.

⚠️ This can’t be done in a standard communication, we define the code from the values of the error.
- Use a PRNG to generate LDPC columns.
\[
Q = \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1) \mathcal{E}(0) \mathcal{E}(0) \mathcal{E}(1)
\]

\[
\mathcal{E}(0 f_1) \mathcal{E}(1 f_2) \mathcal{E}(1 f_3) \mathcal{E}(0 f_4) \mathcal{E}(1 f_5) \mathcal{E}(0 f_6) \mathcal{E}(0 f_7) \mathcal{E}(1 f_8)
\]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Compute the encrypted sparse vector \( \mathcal{E}(c_i f_i) \) as before.
\[ Q = \begin{bmatrix} \mathcal{E}(0) & \mathcal{E}(0) & \mathcal{E}(1) & \mathcal{E}(0) & \mathcal{E}(0) & \mathcal{E}(1) \\ \mathcal{E}(0f_1) & \mathcal{E}(1f_2) & \mathcal{E}(1f_3) & \mathcal{E}(0f_4) & \mathcal{E}(1f_5) & \mathcal{E}(0f_6) & \mathcal{E}(0f_7) & \mathcal{E}(1f_8) \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \times \]

\[ = \begin{bmatrix} \mathcal{E}(0) \\ \mathcal{E}(f_2) \\ \mathcal{E}(f_2+f_3) \\ \mathcal{E}(0) \\ \mathcal{E}(f_3+f_8) \\ \mathcal{E}(f_5+f_8) \\ \mathcal{E}(f_5+f_8) \\ \mathcal{E}(f_2) \\ \mathcal{E}(f_3+f_5) \end{bmatrix} \]

- Compute its syndrome and send it to the user.
The user first decrypts the buffer.
For each singleton, he can generate its column.
He can remove it completely from the buffer.
This uncovers new singletons.
They can again be stripped from the buffer.
PSS with LDPC codes

\[ \begin{align*}
E(f_2) & \rightarrow f_3 + f_5 \\
E(f_3 + f_5) & \rightarrow f_8 \\
E(f_8) & \rightarrow f_2 + f_3 + f_5 + f_8 \\
E(0) & \rightarrow f_3 + f_5 + f_8
\end{align*} \]
All documents were recovered when the buffer is 0.
The whole algorithm is independent of the stream size, the buffer size depends only on the number of matches.

Computationally very efficient:
- for the server, one “encryption” per document,
- for the user, one decryption per buffer position
  \[\Rightarrow\] the rest of the decoding is also linear.

We have full control on the column distribution,
- possible to use constant weight,
  \[\Rightarrow\] not optimal asymptotically,
- possible to use irregular LDPC codes,
  \[\Rightarrow\] use work of Luby, Mitzenmacher, Shokrollahi for asymptotic analysis.
Simulations for buffers of sizes 100, 1000 and 10000:
- for 100, constant weight is as good as irregular,
- we see the asymptotic limitation of constant weight
  at least a ratio 1.22 between buffer/matches.
Our new Private Stream Search scheme:
- compared to the Ostrovsky-Skeith scheme → same computational cost, better communication,
- compared to a non-private search → same asymptotic communication cost, additional computations (especially for the server)

Is it practical?
- probably too expensive using Paillier’s encryption → lighter homomorphic encryption (lattice based?).
- practical from a communication point of view.

Would any search engine want to use it?