Leakage-Abuse Attacks against Searchable Encryption

(Work in progress)

David Cash
Rutgers U

Paul Grubbs
Skyhigh Networks

Jason Perry
Rutgers U

Tom Ristenpart
Cornell Tech
searchable encryption: four algorithms

[Song-Wagner-Perrig], [Curtmola-Garay-Kamara-Ostrovsky], …

1. Encrypted index generation

cloud

nCeUKlK7GO5ew6mwpIra
ODusbskYvBj9GX0F0bNv
puxtuxKuEdbHVuYAd4mE
ULgyJmzHV03ar8RDpUE1
6TfEqihoa8WzcEoI8U8b
Q1BzLK368qufbMMH1GvN
5OVq2xtf2hDUpDig8I0
jyWyuOedYOvYq6XPqZc2
5tDHNCLv2DFJdcD9o4FD
searchable encryption: four algorithms

1. Encrypted index generation

2. Token generation

3. Search w/ token

4. Protocol to update documents

- [Song-Wagner-Perrig]
- [Curtmola-Garay-Kamara-Ostrovsky]
- [Kamara-Papamanthou-Roeder]
A typical basic SSE Approach

Inverted index:

<table>
<thead>
<tr>
<th>keyword</th>
<th>documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rutgers</td>
<td>4, 9,37</td>
</tr>
<tr>
<td>Admissions</td>
<td>9,37,93,94,95</td>
</tr>
<tr>
<td>Committee</td>
<td>8,37,89,90</td>
</tr>
<tr>
<td>Accept</td>
<td>4,37,62,75</td>
</tr>
</tbody>
</table>

"Encrypted" index:

<table>
<thead>
<tr>
<th>keyword</th>
<th>documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>45e8a</td>
<td></td>
</tr>
<tr>
<td>092ff</td>
<td></td>
</tr>
<tr>
<td>f61b5</td>
<td></td>
</tr>
<tr>
<td>cc562</td>
<td></td>
</tr>
</tbody>
</table>

- Search token is just key for a row
- Update protocol adds ciphertexts to correct rows

can encrypt id #’s with per-row keys,
SSE security definition (sketch) [Curtmola-Garay-Kamara-Ostrovsky]

**Def**: Scheme is \( L \)-secure if \( \forall A \ \exists S \) s.t. Real and Simulated games are indistinguishable.

---

**Real Game**
- Chooses key \( K \)
- Runs SE algs as requested by \( A \) (setup, search, etc)

**\((L, S)\)-Simulated Game**
- Queries answered by simulator \( S \) that only sees leakage determined by \( L \)
Example SSE Leakage $\mathcal{L}$

**Intuition:** Scheme is $\mathcal{L}$-secure $\implies$ Server only learns info determined by $\mathcal{L}$ when processing searches and updates.

Example $\mathcal{L}$ outputs (sketch):

- Size info of docs, searches, updates
- Access pattern: Serial numbers of documents returned on each search
- Update pattern: Depending on scheme, which leaks doc ids or hashes of keywords
What does $L$-secure mean in practice?

Messy question which depends on:

- **The documents**: number, size, type/content
- **The queries**: number, distribution, type/content
- **Data processing**: Stemming, stop word removal, etc
- **The updates**: frequency, size, type
- **Adversary’s knowledge**: of documents and/or queries
- **Adversary’s goal**: What exactly is it trying to do?

**Bad news:** Queries can be recovered by server who knows documents. [Islam-Kuzu-Kantarcioğlu]
Current Solution

me, et al.
This work: A Closer Look at Leakage

- Expansion of [Islam-Kuzu-Kantarcioğlu]:
  
  1. **Different leakage functions**: Weaker, practical instances
  
  2. **Different adversary goals**: Document recovery in addition to query recovery
  
  3. **Different adversary knowledge**: (full, partial, and distributional)
  
  4. **Active adversaries**: Can choose some, but not all, docs

- Simple attacks for query recovery, document recovery, with experiments
Datasets for Attack Experiments

Enron Emails
- 30109 Documents from employee sent_mail folders (to focus on intra company email)
- When considering 5000 keywords, average of 93 keywords/doc.

Apache Emails
- 50582 documents from lucene project’s java-user mailing list
- With 5000 keywords, average of 291 keywords/doc

Processed with standard IR keyword extraction techniques (stemming, stopword removal)
Outline

1. Simpler query recovery
2. Document recovery from partial knowledge
3. Document recovery via active attack
Outline

1. Simpler query recovery

2. Document recovery from partial knowledge

3. Document recovery via active attack
Query recovery using document knowledge

[Islam-Kuzu-Kantarcioglu]

Attack setting:

- Server knows all documents
- k random queries issued
- Minimal leakage: Only leak access pattern
- Target: Recover the queries

Inverted index (known):

<table>
<thead>
<tr>
<th>keyword</th>
<th>documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>4, 37, 62, 75</td>
</tr>
<tr>
<td>Admissions</td>
<td>9, 37, 93, 94, 95</td>
</tr>
<tr>
<td>Rutgers</td>
<td>4, 9, 37</td>
</tr>
<tr>
<td>Committee</td>
<td>8, 37, 89, 90</td>
</tr>
</tbody>
</table>

Leakage:

<table>
<thead>
<tr>
<th>Q1</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2</th>
<th>D1</th>
<th>D2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q4</th>
<th>D1</th>
<th>D2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q5</th>
<th>D1</th>
<th>D2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q6</th>
<th>D1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Observation: When a query returns a unique number of documents, then it can immediately be guessed.

- Then we can often “disambiguate” remaining queries by checking intersections.

Leakage:

<table>
<thead>
<tr>
<th></th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q4</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q3 matched 3 documents, so it must be “Rutgers”.
**Query Recovery Experiment**

Setup:
- Enron email subset
- k most frequent words
- 10% queried at random

- Nearly 100% recovery, scales to large number of keywords, runs in seconds
Possible Defense: Padding

Pad rows of index to multiples (e.g. of 100) with random false positives

- Reduces number with unique counts
- Confuses co-incidence information
- Increases space

We generalized counting attack to account for padding and tested it with varying levels of padding.
**Padding Performance**

---

**Diagram Description:**

- **Y-axis:** Reconstruction rate
- **X-axis:** Padding overhead – # index entries

**Data Points:**

- **Blue Circles:** Enron emails
- **Red Squares:** Apache "java-user"

**Graph Analysis:**

- The reconstruction rate decreases as the padding overhead increases.
- Enron emails show a steeper decrease compared to Apache "java-user".

**Legend:**

- Enron emails
- Apache "java-user"
Query Recovery with Partial Knowledge

- What if document set is only partially known?

- We tested count and IKK attacks when only $x\%$ of the document was revealed
Query Recovery with Partial Knowledge

- **IKK Attack**: The results are with the server knowing noisy to prevent any disambiguation. Note that unlike with drops, there is more co-occurrence information. The recovery rate is about 15% for the Enron data, and 30% for the Apache data.

- **Count Attack**: Figure 5 shows the effectiveness of the count attack from being carried out exactly: The number of co-occurrences of two keywords in the padded index may exceed the co-occurrence count of the corresponding keywords. However, we can modify the algorithm to adjust its comparison so that, instead of requiring an exact match of co-occurrence counts, it accepts co-occurrence with slight modifications in the case when the server does not know the full document set. As with the case of showing that if the server is trained on a random subset of the same corpus, (the unknown documents case), the IKK attack fails completely.

**Analysis of the IKK attack with partial knowledge.**

- To understand better how real-world limitations on a server's knowledge affect the accuracy of the attack, we devised a new experiment. Instead of adding noise to the co-occurrence probabilities, technically it does not require the server to know only 80% of the dataset for significant query recovery. Figure 6 shows that if the server has access to 99% of the true documents, and compute the co-occurrence probabilities from that. We duplicated the experiments of \[6\] with the true documents, and the dataset known to the server. In brief, these results indicate that unless the server has access to 99% of the true documents, the query recovery rate is quite poor.

**Query recovery rate**

- **IKK Attack**: The query recovery rate is quite poor.

- **Count Attack**: The recovery rate is quite poor.

- **% of dataset known to server**
  - Enron subset, 500 most frequent keywords, 150 queried uniformly. Server knows 5% of queries at start.
Outline

1. Simpler query recovery

2. Document recovery from partial knowledge

3. Document recovery via active attack
Document Recovery using Partial Knowledge

This blob indexes some docs I happen to know… What does that tell me?
Passive Document Recovery Attack Setting

- Server knows type of documents (i.e. has training set)
- No queries issued at all
- Some documents become “known”
- **Target:** Recover other document contents
Leakage that We Attack

- Stronger SSE schemes are immune to document recovery until queries are issued
- So we attack weaker constructions of the form:

  Doc 1:
  The quick brown fox [...]
  zAFDr7ZS99TztuSBIf[...]
  PRF(quick), PRF(brown), PRF(fox), ...

  Doc 2:
  The fast red fox [...]
  zAFDr7ZS99TztuSBIf[...]
  PRF(fast), PRF(red), PRF(fox), ...

Example systems:
- Mimesis [Lau et al’14]
- Shadowcrypt [He et al’14]

(On-going work: generalize to CGKO)
Trivial Observation

- If I know Doc 1, then I learn when every word appears in other docs
- Implementation detail: We assume PRF values stored in order.
- Harder but still possible if PRFs in random order. (ongoing work)
For each dataset, we ran attack knowing either 2 or 20 random emails.
Anecdotal Example

The attached contract is ready for signature. Please print 2 documents and have Atmos execute both and return same to my attention. I will return an original for their records after ENA has signed. Or if you prefer, please provide me with the name / phone # / address of your customer and I will Fed X the Agreement.

Attach contract signatur pleas print 2 document have execut both same will origin ena sign prefer provid name agreement

- From Enron with 20 random known documents
- Note effect of stemming, stopword removal, and revealing each word once
The effect of one public document

Case study: A single email from the Enron corpus, sent to 500 employees

- 832 Unique Keywords
- Topic: an upcoming survey of the division by an outside consulting group.

The vocabulary of this single document gives us on average 35% of the words in every document (not counting stopwords).
Outline

1. Simpler query recovery

2. Document recovery from partial knowledge

3. Document recovery via active attack
Chosen-Document-Addition Attacks

Emails → Local Proxy → Add protocol → SE blob → Dropbox
Chosen-Document-Addition Attacks

Leakage from my crafted email!

Local Proxy

Emails

Add protocol

SE blob

Dropbox
Again we attack weaker constructions of the form:

- **Doc 1:**
  - The quick brown fox [...]
  - PRF(quick), PRF(brown), PRF(fox), ...

- **New Doc:**
  - contract sell buy
  - PRF(contract), PRF(buy), PRF(sell), ...

- **PRF values in order ⇒ very easy attack**

- **PRF values not in order ⇒ more difficult**
Chosen Document Attack Experiment

- **Goal:** Maximize fraction keywords learned from a minimum number of chosen documents emails

- **Procedure:**
  1. Divided dataset into half training / half test
  2. Based on training set, rank keywords by frequency
  3. Generate chosen emails with k keywords each
  4. Learn unordered PRF values of those k keywords
  5. Guess PRF → keyword mapping via frequency counts

- Ran with two different training setups:
  1. Used training with same test (i.e. both Enron or both Apache)
  2. Mixed training and test (i.e. train on Apache, test on Enron)
Chosen Document Attack Experiment Results

![Graph showing keyword recovery rate and error rate](imageURL)

**Keywords per chosen document**

**Fraction of keywords**

- **Recovery rate**
- **Error rate**
- **Recovery, unrelated**
- **Error, unrelated**

**Figure 9:** Keyword recovery rate and error rate.
Future Work and Open Problems

- Several directions left to explore
  1. Passive attacks when PRFs not in order
  2. Active attacks targeting specific information
  3. Document recovery against stronger SSE (need queries)
  4. Other attacks? (identify language, test if document present, …)

- On-going work: Attacks using automatic human language translation

- Should we talk about leakage in a scientific way, using information theory?
Thanks!