Leakage-Abuse Attacks against Searchable Encryption

David Cash
Rutgers University

Paul Grubbs
Cornell University

Jason Perry
Lewis University

Tom Ristenpart
Cornell Tech
Outsourced storage and searching

• “records” could be emails, text documents, Salesforce records, …
• searching is performed efficiently in the cloud via standard techniques
End-to-end encryption breaks searching

- Searching incompatible with privacy goals of traditional encryption
Companies actively trying to navigate this tension, providing **Searchable Encryption**

[Logos of CipherCloud, bitglass, skyhigh, CRYPTERON]
Searchable Symmetric Encryption \([\text{SWP}'00, \text{CGKO}'06, \ldots]\)

- Also includes *Update Protocol* for adding new documents/records
Example SSE Construction

Client uploads an *encrypted inverted index*

All SE **leaks** some information to the server
What does SSE leakage look like?

Keywords/data not in the clear, but some info can be derived by server

<table>
<thead>
<tr>
<th>keyword</th>
<th>records</th>
</tr>
</thead>
<tbody>
<tr>
<td>45e8a</td>
<td>4, 9, 37</td>
</tr>
<tr>
<td>992ff</td>
<td>9, 37, 93, 94, 95</td>
</tr>
<tr>
<td>f61b5</td>
<td>9, 37, 89, 90</td>
</tr>
<tr>
<td>cc562</td>
<td>4, 37, 62, 75</td>
</tr>
</tbody>
</table>

“this keyword is the most common”

“document #37 contains every keyword, and appears together with #9 often”

• Highly unclear if/when leakage is dangerous
How SE is analyzed in the literature

Crypto security definitions usually formalize e.g.:

“nothing is leaked about the input, except size”

**SE uses a weakened type of definition:**

- Identify a formal “leakage function” $\mathcal{L}$
- Allows server to learn info corresponding to $\mathcal{L}$, but no more

**Example $\mathcal{L}$ outputs:**

- Result lengths (number of records matching a query term)
- Query repetition
- Access pattern: Repeated record IDs across searches
- Update information
Main question:
How serious is “leakage” in practice?

Currently almost no guidance in the literature.

A messy question that depends on many variables.

- Type of data, size of dataset, how data is processed, what queries look like, update frequency, adversary knowledge, attacker goal, etc., etc.
One prior work: Learning queries

Under certain circumstances, queries can be learned at a high rate (80%) by a curious server who knows the contents of all of the documents that were encrypted.

[Islam-Kuzu-Kantarcioglu ‘12]

(sketched later)
This work: Practical Exploitability of SE Leakage

Broaden and systematize the approach of [IKK ‘12]:

1. **Different adversary goals**: Document (record) recovery in addition to query recovery

2. **Different levels of adversary knowledge**: (full, partial, and distributional)

3. **Active adversaries**: planted documents

Simple **leakage-abuse attacks** for query recovery and document recovery, with experiments

- Attacks apply to all constructions with the same or greater leakage profile
Datasets for Attack Experiments

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

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

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

1. Simpler query recovery
2. Document recovery from partial knowledge
3. Document recovery via active attack
Query recovery using document knowledge [IKK ‘12]

Attack setting:

- **Minimal leakage**: Only which records match each query (as SSE)
- **Server knows all document content**: e.g., public financial data
- **$k$ random queries issued**
- **Adversary Goal**: Learn the queries

Server Observes:

<table>
<thead>
<tr>
<th>keyword</th>
<th>records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4, 37, 62, 75</td>
</tr>
<tr>
<td>Q2</td>
<td>9, 37, 93, 94, 95</td>
</tr>
<tr>
<td>Q3</td>
<td>4, 9, 37</td>
</tr>
<tr>
<td>Q4</td>
<td>8, 37, 89, 90</td>
</tr>
</tbody>
</table>

Unknown term-doc matrix:

<table>
<thead>
<tr>
<th></th>
<th>rec1</th>
<th>rec2</th>
<th>rec3</th>
<th>rec4</th>
<th>rec5</th>
<th>rec6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
The IKK attack (sketch)

Term Co-occurrence matrix:

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2</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>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>3</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>

- Server constructs term co-occurrence matrix
- Attempts to solve optimization problem of matching to known term co-occurrence matrix
- Over 80% correct for small index sizes (1500 keywords)
Hypothesis

The IKK attack works only if the server has highly accurate knowledge of the document set.

If so, then why not just check the number of documents returned by a query?

**Observation:** When a query returns a unique number of documents, then it can immediately be guessed.
Query Recovery via Counts

• After finding unique-match queries, we then “disambiguate” remaining queries by checking intersections

<table>
<thead>
<tr>
<th>Leakage:</th>
<th>rec1</th>
<th>rec2</th>
<th>rec3</th>
<th>rec4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td>1</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 records, so it must be “rutgers”

Q2 overlapped w/ one record containing “rutgers” so it must be “denver”
Query Recovery Experiment

Setup:
- Enron email dataset
- 10% queried at random

Runs in seconds, not hours
Query Recovery with Partial Knowledge

• What if document set is only partially known?

• We generalized the counting attack to account for partial information, and tested the count and IKK attacks when only x% of the documents are known.
Query Recovery with Partial Knowledge

Enron emails, 500 most frequent keywords indexed (stemmed, non-stopwords), 150 queried at random, 5% of queries initially given to server as hint
1. Simpler query recovery

2. Document recovery from partial knowledge

3. Document recovery via active attack
This blob indexes some docs I happen to know and others I don’t… What does that tell me?
Passive Document Recovery Attack Setting

• No queries issued at all
• Some documents become “known” to the server
• **Attacker Goal**: Recover other document contents
New Leakage Profile

- Attack on weaker “appended keyword hash” constructions:

  Record 1:
  The quick brown fox [...]

  Record 2:
  The fast red fox [...]

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

  Record 1:
  zAFDr7ZS99TztuSBIf[...]
  H(K,quick), H(K,brown), H(K,fox), ...

  Record 2:
  Hs9gh4vz0GmH32cXK5[...]
  H(K,fast), H(K,red), H(K,fox), ...

Legacy-compliant
Simple Observation

Known:

Doc 1:

zAFDr7ZS99TztuSBIf[...]

H(K, quick), H(K, brown),

H(K, fox), ...

Unknown:

Doc 2:

zAFDr7ZS99TztuSBIf[...]

H(K, fast), H(K, red),

H(K, fox), ...

• If server knows Doc 1, then learns when any word in Doc 1 appears in other docs

• Implementation detail: We assume hash values stored in order.

• Harder but still possible if hash in random order. (see paper)
Document Recovery with Partial Knowledge

For each dataset, server knowing either 2 or 20 random emails

<table>
<thead>
<tr>
<th>Dataset, # Known Docs</th>
<th>Average Keywords Recovered / Doc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron, 2</td>
<td>16.3%</td>
</tr>
<tr>
<td>Enron, 20</td>
<td>56.0%</td>
</tr>
<tr>
<td>Apache, 2</td>
<td>50.7%</td>
</tr>
<tr>
<td>Apache, 20</td>
<td>68.4%</td>
</tr>
</tbody>
</table>
Anecdotal Example

Original email:
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.

Reconstructed in-order stems:
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

Local Proxy

Emails

SE index

Leakage from my crafted email!

update protocol

SE index

Local Proxy
Chosen-Document Attack ⇒ Learn chosen hashes

• Again we attack the appended-keyword-hash constructions

Doc 1:
The quick brown fox [...]

Doc 1:
zAFDr7ZS99TztuSBIf[...]

H(K,quick), H(K,brown), H(K,fox), ...

Doc 1:
zAFDr7ZS99TztuSBIf[...]

H(K,fox), H(K,quick), H(K,brown), ...

• Hashes in order ⇒ very easy attack
• Hashes not in order ⇒ more difficult (we attack now)
**Chosen Document Attack Experiment**

**Goal:** Maximize number of keywords learned from a minimum number of chosen documents (emails)

Procedure for generating chosen emails:
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 hash values of those k keywords
5. Guess hash → keyword mapping via frequency counts

Two different training setups:
1. Training and test sets from same corpus (both Enron or Apache)
2. Training and test from different corpora (i.e. train on Apache, test on Enron)
Chosen Document Attack Experiment Results

![Graph showing the fraction of keywords against keywords per chosen document. The graph includes lines for recovery rate, error rate, recovery unrelated, and error unrelated.]
Conclusion

Systematic study of exploitability of multiple SE leakage types

- Temptation to deploy ad-hoc solutions must be avoided
- Need framework + experiments for understanding what one can do with leakage
- We’ve only scratched the surface…
  - Info retrieval and natural language methods!
Thanks!