Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions

Javad Ghareh Chamani
HKUST

Mohammadamin Karbasforushan
UCSC

Dimitrios Papadopoulos
HKUST

Ioannis Demertzis
UCSC

USENIX Security
August 10th – 12th, 2022
**Motivation:** Private Cloud Computing

- Sensitive Data, GDPR, US HIPAA, ...
- Encryption + Encryption key at client-side
  + Efficient search and update

**Searchable Encryption** [SWP00] [DPPS20] [KM18] [KMO18] [PPYY19] [CJJKRM13] [CNR21] [CK10] [DPPDG18] [DPPDG16] [FJKNRS15] [KM17] [MKNK15] [ANSS16] [DPP18] [DP17] [DTP18] [MM17] [BBFMR21] [HSWW18] [RMO15] [RMO18] [WP21] [YLLJC14]

- Especially, **dynamic with efficient search** [B16] [KP13] [KKPR12] [GPPR18] [CXWZSJ21] [DGPP20] [EKPE18] [KKLPK17]

**Applications:**

- MongoDB’s Queryable Encryption
- Pixek: Annotated Image Search
- Gun Registry (US Senate & Brown University)
- Encrypted key-value stores and databases
- Private end-to-end email communications with search capabilities
  - Via encrypted inverted index
Dynamic Searchable Encryption (DSE) [LSDHJ10] [SPS14] [BMO17]

Client

Untrusted Cloud

**What does the Cloud learn?**
Controlled information called leakage.

Well-Defined and Cryptographically Proven!

**Setup Leakage**

**Search Leakage**

**Update Leakage**

**Total Size**

Search Pattern
Query Repetition

Access Pattern
Records Fetched

Next slide...

Client Untrusted Cloud

Setup

Search

Update (Insert/Delete)

Encrypted Multi-Map

Encrypted Search Token

Encrypted Multi-Map

Encrypted Search Result

Encrypted Search Token

Encrypted Search Result

What does the Cloud learn?
Controlled information called leakage.

Well-Defined and Cryptographically Proven!
DSE Update Leakage

Forward Privacy [SPS14] [ZKP16]: The untrusted cloud should not link updates to previous searches. ... during updates, about searches

Backward Privacy [SPS14] [BMO17]: Controlled information about inserted-then-deleted values should be leaked. (at least not the value) ... during searches, about updates

Formally, the update should leak nothing about the key

Accessed memory locations during search

Forward and Backward Privacy have become the de-facto requirements in the literature.
Prior Work: Sub-Optimal Search

\[
\begin{align*}
\text{Insert}(\text{Key1}, \text{Val1}) \\
\text{Insert}(\text{Key1}, \text{Val2}) \\
\vdots \\
\text{Insert}(\text{Key1}, \text{Val}_i) \\
\text{Delete}(\text{Key1}, \text{Val1}) \\
\text{Delete}(\text{Key1}, \text{Val2}) \\
\vdots \\
\text{Delete}(\text{Key1}, \text{Val}_{i-1}) \\
\text{Search}(\text{Key1})
\end{align*}
\]

\(i: \text{insertions}\)

\(a = i + d: \text{all updates}\)

\(d: \text{deletions}\)

\(r = i - d: \text{result size}\)

- **Cancellation Records** [GPPJ18] [DGPP20] [BMO17]: \(O(a)\)
  - Easy but bad scaling with deletions

- **Quasi-Optimal** [SPS14] [GPPJ18] [GPPJ18] [DGPP20]: \(O(r \cdot \text{polylog}(N))\)

- **Optimal**: \(O(r)\)
  - Only in plaintext schemes!

\[N: \text{capacity}\]
Achieving Optimal Search with Deletions (OSSE)

- Updates propagate up the tree
- $O(D \cdot \log N)$ space for deletions ($D$: total deletions)
  - Always write $\log N$ nodes to stay forward private
- Number of valid nodes remains $O(r)$...

Search Time $= O(r + \log i)$

- Binary tree on top of entries to avoid deleted regions
- $v_i$: node version, increased when pointers change
**Optimal Search Details**

**Encrypted Search Token**: points to the root
- Hash (PRF) of encryption key and search query + root id + root version
  \[ H(\text{Key1}, \text{id}=1-8, \text{ver}=2) \]

**Client needs root id and version of each key**
- Stores per-key insertion and deletion counters
- Can use Oblivious Encrypted Map (OMap) [WNLCSSH14] for O(1) client storage
  - Hides op and args; adds \( O(\log^2 N) \) cost over \( O(\log N) \) rounds

**Updates and Forward Privacy**
- OMaps to store tree state across updates (value-leaf mappings, node contents)

More costly updates to get optimal search
- \( O(N + D \cdot \log N) \) Space
Second Scheme (LLSE)

- OSSE uses $O(N + D \cdot \log N)$ space
- How to prevent delete propagation?

- Idea: Don’t store children’s versions
- Binary search for latest node version
- Asymptotically and empirically faster than previous State-of-the-Art!

$O(\log N)$ Deletion Speedup

$O(N)$ Space

$O(r \cdot \log \log N)$ Search

Key1: 1 2 3 4 5 ...
Experiments

- Implemented in C++, using OpenSSL for crypto, AES-NI enabled
  - Open-source: github.com/jgharehchamani/OS-SSE

- compared with the best scheme with cancellation records (SDD) and previous SotA quasi-optimal scheme (QOS) [DGPP20]

- Hardware setup: 8-core Intel Xeon E-2174G 3.8Ghz, 128GB RAM, Ubuntu 16.04 LTS

- Experiments ran on a single machine

- Several optimizations compatible with privacy and leakage profile
Experiment: Search Time

Setup
- Search Time vs. Delete Percentage
  - DB Size = 1M, inserted values = 20K
  - Deleted uniformly chosen values
- SDD: Best cancellation records scheme
- QOS: Best quasi-optimal search scheme
  - Previous SotA
- OSSE: Optimal Search Scheme (ours)
  - OSSE*: OSSE + optimizations

Results
- OSSE beats SDD faster than QOS
- OSSE beats QOS right off the bat!
- Optimizations bring a very significant boost
Conclusion

- OSSE: First forward and backward private dynamic searchable encryption scheme with optimal search
- LLSE: Asymptotically and empirically faster than previous SotA with $O(N)$ space vs. $O(N + D \log N)$ in OSSE
- Open-source!

Thank You!