Practical Dynamic Searchable Encryption with Small Leakage

Charalampos (Babis) Papamanthou
University of Maryland
cpap@umd.edu

joint work with Emil Stefanov (UC Berkeley) and Elaine Shi (University of Maryland)
Storing private files in the cloud

- How can you search your encrypted files?
  - Not feasible with a widely-used encryption algorithm (e.g., AES)
  - Encrypt with fully-homomorphic encryption (FHE)?
    - Not very practical
  - Access with an ORAM scheme?
    - Not very practical
Searchable encryption (SE)

- Lots of work since [SPW00]
- Static schemes (setup, search)
  - e.g., [CGKO06], [KO12], [CJJKRS13]
- Dynamic schemes (setup, search, add, delete)
  - e.g., [SPW00], [G03], [vSDHJ10], [KPR12], [KP13], [CJJJKRS14], [NPG14]
Some leakage

- All existing (dynamic) SE schemes leak
  - search pattern
    - hashes of keywords I am searching for
  - access pattern
    - matching document identifiers
  - size pattern
    - the current size of the index
More leakage

- Some dynamic SE schemes also leak
- forward pattern
- documents can be searched with old
- update pattern
  - hashes of keywords in the updated documents

But, linear search or update time: $O(N)$ 😞
Our contribution

- The first dynamic SE scheme
  - Supports searches, insertions, deletions
  - No forward pattern leakage
  - No update pattern leakage
  - Sublinear search time: $O(m \log^3 N)$
    - $m$ is the number of documents matching the search
  - Sublinear update time: $O(k \log^2 N)$
    - $k$ is the number of unique keywords contained in the document
- Provably secure
- System implementation
  - 100,000 queries per second for 100 search results
Simple SE scheme: Token

- Client has secret key $K$
- Definition of token for word $w$

$w \xrightarrow{K} PRF \xrightarrow{t_w}$

Tokens are deterministic!
Simple SE scheme: Construction

encoded hash table $T$

- initial index $D$
  - $(w, d)$
  - $(w, d')$

- KEY = \( \text{HASH}(t_w \Vert \text{count}) \)

- VALUE = $d \oplus \text{HASH}(t_w \Vert \text{count})$
Searching for keyword \( w \)

- Client: Sends \( t_w \)
- Server: Looks up the entries mapping to \( t_w \)
  - Learns nothing about keyword \( W \)
Adding (w', d')

- Client: Sends new (KEY, VALUE) for (w', d')
Adding \((w', d')\)

- Client: Sends new \((\text{KEY, VALUE})\) for \((w', d')\)
- Server: Updates the hash table
- But...
  - Tokens are deterministic
  - No forward privacy 😞

How about re-encrypting with a different key?
Linear time: \(O(N)\) 😞
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- $l = \log_2 N + 1$ levels
Levelled data structure

- $l = \log N + 1$ levels
Levelled data structure

- \( l = \log N + 1 \) levels
Levelled data structure

- $l = \log N + 1$ levels

Time per operation:

$O(\log N)$
Our scheme: Search

- Maintain on key per level
- Client: Sends tokens $t_1, t_2, \ldots, t_l$ for $w$
- Server: For each level $i$, un_masks entries for $w$
Our scheme: Add

- Try level 1: It does not fit
Our scheme: Add

- Try level 1. It does not fit.
- Client downloads consecutive-filled levels (levels 1 and 2)
**Our scheme: Add**

- Try level 1. It does not fit.
- Client downloads consecutive-filled levels (levels 1 and 2)
- Client **reencrypts with new secret keys** and uploads to level 3
- Only $O(\log^2 N)$ per operation

**Forward privacy:**
Old tokens are no good
How about deletes?

- Treat them as special “add” entries
- Can create problems
  - 5 addition entries for word $w$ at level 4
  - 4 deletion entries for word $w$ at level 3

$O(N)$ time for retrieving one document 😞

We show in the paper how to do that in $O(\log^3 N)$
Implementation

- Implementation in C#
- Experiments were run on Amazon EC2
- 244 GB of memory
Query throughput

![Graph showing query throughput with DB size in millions of document-keyword pairs and number of results per query on the x-axis and throughput (queries/second) on the y-axis.]
Update throughput

![Graph showing update throughput against database size with different network latencies.](image)

**Update Throughput**

- **Network Latency**:
  - 25 ms
  - 50 ms
  - 100 ms

- **Y-axis**: Document-KeyWord Pairs / second
- **X-axis**: Database Size (millions of document-keyword pairs)
Bandwidth utilization
Thanks!
Updates: Data structure

![Update Throughput Graph](image)
Fig. 10: Update Bandwidth. The bandwidth used to add an
Updates: Encrypted data structure

- hash tables
Updates: Data structure

- \( l = \log N + 1 \) levels
Updates: Data structure

- $l = \log N + 1$ levels
Updates: Data structure

- \( l = \log N + 1 \) levels
Updates: Data structure

- $l = \log N + 1$ levels
Searchable encryption

- Lots of work since 2000
- Static constructions
  - \([\text{CGKO06}, \text{KO12}, \text{CJJKRS13}]\)
- Dynamic constructions

- **My work:** First dynamic efficient scheme, \([\text{CCS12}]\)
  - Privately indexes keywords, not only files
  - Efficient system implementation
Verifiable Computing

- \( \pi \) should be \( O(|F(u)|) \)
- Cloud should not be able to cheat
- Many works in the literature
Recent breakthroughs

- In theory
  - Give me any circuit C, I can create a VC protocol for you
    - E.g., Quadratic Span Programs (EUROCRYPT 13)

- In practice
  - Many systems have been developed to implement VC
    - E.g., Pinocchio (SSP 13), Pantry (SOSP 13)
  - Immense improvement in the practical landscape of VC since 2010…
    - …when the only way to do general VC was FHE and PCPs
  - Still not practical for real-life applications
    - E.g, a SELECT query over a database of one billion records?
My approach: Focus on popular applications

practicality

expressiveness

popular cloud applications

my approach

my

Dropbox

any circuit

any RAM program
Some numbers

- **Intersection** of two sets of 10,000 entries each where the output is 200 elements:
  - ~2 seconds (proof computation)
- **Shortest path** over a planar graph of 10,000 nodes
  - ~3 seconds (proof computation)
- **Pattern matching** of a 10-character pattern (match/mismatch) over a text of 100,000 characters
  - ~25 µs (proof computation)
- Verification is always fast
Grand challenges ahead

- Still we are not practical enough
- Normal conjunctive keyword search takes order of microseconds
  - The added verifiability guarantee takes order of seconds
  - Same with shortest paths
- Plenty of room for improvement
  - Expertise from crypto and systems and algorithms required
- Grand challenge: Build a verifiable DBMS with reduced overhead