Making Searchable Encryption Scale to the Cloud

Ian Miers and Payman Mohassel
End to end Encryption

No encryption

Transport encryption

End2End Encryption
E2E Encrypted Messaging

Need Crypto

Want Crypto
E2E Encrypted Messaging

Ban Crypto

Need Crypto

Want Crypto
Deploying E2E encrypted messaging

- WhatsApp
  - No feature loss
  - Many users probably don’t know they are using it
- iMessage
  - Same features as SMS
- WebRTC video chat
Search

- For some communication mechanisms, people expect search
- Email is the canonical example, but not the only one.
  - Slack
  - Any “email replacement “
“I’m not particularly thrilled with building an apartment building which has the biggest bars on every window” – Jeff Bonforte (Yahoo VP mail and messaging)
E2E Encrypted Messaging

Need Crypto

Want Crypto
Searchable Encryption
An index for search

The

f_1  f_5  f_9  f_{52}  f_{67}  f_{72}  f_{99}

Man

f_7  f_{11}  f_{21}  f_{42}  f_{67}

Bites

f_2

Dog

f_{21}  f_{27}  f_{31}
Index on client, store on server
Index on client, store on server

Dog

Man

Bites
Index on client, store on server
An index for search

The

Man

Bites

Dog
A Naïvely Encrypted Index

\[ H(k|\text{keyWord}) \quad E(k,\text{list of files}) \]

- 8afa2
- 1c35f
- dc4cf
- 9f126
Index on client, store on server

$H(k|"Dog")$
A Naïvely Encrypted Index

H(k|keyWord)  E(k,list of files)

- 8afa2
- 1c35f
- dc4cf
- 9f126

Leaks term frequency
- 8afa2 is the most frequent keyword
- “The” is the most frequent English word
- ......
An Inefficient Encrypted Index

For a given keyword, each file containing it is stored in a separate random location.

This hides keyword frequency in a space efficient way.

Very inefficient to search:

- Requires one random read per result
- Results in a ~25-50x increase in I/O usage
- Yahoo! Mail search is already IO bound !!!
- Not viable for a server supporting multiple users who are not paying for it
Search at Cloud Scale

- Many small indexes
  - < 1GB each
  - > 1 Billion accounts
- Cannot store in memory
- Must use disk storage
- IO Bound
- Fragmented index causes massive increase in iO for search
- A search for one keyword returning N documents takes N times as many reads.
Good news

- Email search queries are fairly simple
  - Typically single keyword
  - Conjunctive search nice, but not necessary
- Most searches are on meta data
- Searches on mail content are rare
  - ~250 searches a second across all users
  - ~300 million monthly active users
- But we must solve the IO issue.
An Inefficient Encrypted Index

\[ H(k|\text{keyWord}|\text{KeyWord}_\text{ctr}) \]

\[ E(k, f_i) \]
IO Efficient search for static indexes
Chunked Encrypted Index

$H(k|keyWord|chunk\_ctr)$

$E(k, chunk\ of\ files)$

- Assume we have all documents initially
- We break up the list into chunks
- Way more efficient to search
- Can scale to terabytes
- Cash et al (Crypto ’13, NDSS ’14)
Problem: updates

H(k|keyWord|chunk_ctr)  E(k, chunk of files)

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1

- “lost DOG”
- Dog is “9f126”
- Need to add to “Dog” entry.
- But … that leaks what we updated
Problem: updates

H(k|keyWord|chunk_ctr)  E(k, chunk of files)

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1

- :“lost DOG”
- Dog is “9f126”
- Need to add to “Dog” entry.
- But … that leaks what we updated
Problem: updates

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1
- fe52
Problem: updates
IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality
Obliviously Updateable Index

Standard search index

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1

fe52
Obliviously Updateable Index

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1

fe52
Obliviously Updateable Index
Obliviously Updateable Index

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1

fe52
Obliviously Updateable
Index
Obliviously Updateable

Index
Chunked Encrypted Index

- 8afa2
- 1c35f
- 41bb
- 9f126
- dc4cf
- d4c1
- fe52
Buffer locally, put full chunks on server

- Keywords have a power law distribution: common ones are really frequent, others are sparse
- We will end up with too many partial buckets on the client
- We can’t upload partial buckets
We need an obliviously updatable index.
Oblivious RAM

- ORAM hides locations of access to memory (both reads and writes)
- How to build ORAM
  1. Encrypt memory
  2. “Shuffle” memory locations on reads or writes to hide locations
- In Path ORAM, shuffling has logarithmic overhead.
OUI from Path ORAM RAM

1. Read (for search)
2. Shuffle
3. Read (for search)
4. Shuffle

5. Read/write for update
6. Shuffle
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
From ORAM to an OUI

- ORAM allows you to write to a location in memory without revealing the location.
- Can add to a partial chunk without revealing we did so.
- Bandwidth costs get worse as ORAM gets larger.
  - Requires you to read and write $\log(N) \times B$ bytes for a read of $B$ bytes from an ORAM of size $N$.
  - For 16GB of ORAM, server needs 32.06 GB of space and reading 4KB takes 350KB read + 350KB write.
- Storing full index in ORAM requires too much bandwidth.
From ORAM to an OUI

- ORAM hides both reads and writes
- Search explicitly leaks repeated reads
  - Same files are returned each time.
  - Same search token/hash used.
  - No need to hide reads using ORAM
- Updates may happen in batches
OUI from Oblivious RAM

1. Read (for search)
2. Shuffle
3. Read (for search)
4. Shuffle

5. Read/write for update
6. Shuffle
Partial ORAM?

1. Read (for search)
2. Read (for search)
3. Read/write for update
4. Shuffle
1. Read (for search)
2. Shuffle
3. Read (for search)
4. Shuffle
5. Read/write for update
6. Shuffle
1. Read (for search)
2. Read (for search)
3. Shuffle + Shuffle
4. Read/write for update
5. Shuffle
1. Read (for search)
2. Read (for search)
3. Shuffle + Shuffle
4. Read/write for update
5. Shuffle
1. Read (for search)
2. Read (for search)
3. Shuffle + Shuffle
4. Read/write for update
5. Shuffle
Reads
Why not just read directly?
Leaks updates
OUI from ORAM

- Searching triggers a read and write of $\log(n) \times B$ data
- To avoid $\log(N) \times B$ read +write for each search
  - Just read address for chunk for given keyword
  - Defer read and write until later (i.e. when the phone is plugged in and on Wi-Fi)
  - Search is constant bandwidth and has nice locality
- All updates **must** happen after deferred IO is done
- We get some savings from batching the IO together
- Multiple searches on the same keyword are free
OUI from ORAM

- Read directly from tree for search BUT
- Must complete full path read and write prior to any updates
- Call these “deferred” reads
Batched reads and writes

- Deferred (full reads) reads and updates are not random events
- They will happen in groups either
  - When an email comes in we get many updates
  - We might update the non local index only once a day (if system is not multi client)
- Batched reads and writes reduce the amount of data read and written
- For n full reads/writes,
  - The root is only updated once instead of n times
  - Its children once instead n/2 times, etc
Deferred Reads
Deferred Reads
Deferred Reads
Deferred Reads + Batching
Batched update
Performance

IO Savings (percentage) vs
- Simple encrypted index (including all previous works under purely dynamic insertion)
- Savings just for search (ignoring updates)
- Oblivious index from path ORAM
Conclusion

- Searchable encryption might be feasible for cloud based messaging with effort
- It pays to examine problems in context
- You can always get better performance by relaxing security assumptions
- Sometimes the relaxation is inherent to the setting and free
Updates

- **Query** local, ORAM, and index with efficient access

- **Update**: Buffer locally, overflow to ORAM, then commit full chunks to index

- **Defer ORAM I/O** from queries until update period

**Requirements**
- 40 to 250mb of client storage to store a list of keywords
- Client has fast internet sometimes
- Ideally, client has large local buffer

![Diagram showing local buffer, key data, Obliviously updateable Index (OUI), and query on keyword.]
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
Path ORAM

Client side stash
From ORAM to an OUI

- ORAM allows you to write to a location in memory without revealing the location.
- Can add to a partial chunk without revealing we did so.
- Bandwidth costs get worse as ORAM gets larger.
  - Requires you to read and write \(\log(N)\)*\(B\) bytes for a read of \(B\) bytes from an ORAM of size \(N\).
  - For 16GB of ORAM, server needs 32.06 GB of space and reading 4KB takes 350KB read + 350KB write.
- Storing full index in ORAM requires too much bandwidth.
From ORAM to an OUI

- ORAM hides both reads and writes
- Search explicitly leaks repeated reads
  - Same files are returned each time.
  - Same search token/hash used.
  - No need to hide reads using ORAM
- Updates may happen in batches
Why not just read directly?
Leaks updates
OUI from ORAM

- Read directly from tree for search BUT
- Must complete full path read and write prior to any updates
- Call these “deferred” reads
Batched reads and writes

- Deferred (full reads) reads and updates are not random events
- They will happen in groups either
  - When an email comes in we get many updates
  - We might update the non local index only once a day (if system is not multi client)
- Batched reads and writes reduce the amount of data read and written
- For n full reads/writes,
  - The root is only updated once instead of n times
  - Its children once instead n/2 times, etc
Deferred Reads
Deferred Reads
Deferred Reads
Deferred Reads + Batching
Batched update