Newcastle University

Forward Private Searchable Symmetric Encryption with Optimized I/O Efficiency

Changyu Dong
<changyu.dong@newcastle.ac.uk>

Joint work with Xiangfu Song, Dandan Yuan, Qiuliang Xu, Minghao Zhao

Motivation: Data Outsourcing

- Explosive growth in enterprise data
  - storage needs grow 52% per year [Forrester Research]
  - escalating storage management costs: $9,555/TB/year [Forrester Research]
- Increased importance of data availability and business continuity
  - remote backup to prevent data loss in disasters like 9.11
- Here they come to help you!
  - Amazon, Google, IBM, Microsoft, HP ...
  - ... by providing cheap-as-chips data storage outsourcing service.

You Don't Trust Them, Do You?

- You might save money, you might get better fault-tolerance, you might even get better performance.
- But how about data confidentiality and privacy? Do you really want someone else to see and control all your sensitive data?

A True Story

In Oct 2003, a woman in Pakistan obtained sensitive patient documents from the University of California, San Francisco, Medical Centre through a medical transcription subcontractor that she worked for, and she threatened to post the files on the Internet unless she was paid more money.

> *"Your patient records are out in the open... so you better track that person and make him pay my dues."*
>
> *– San Francisco Chronicle (October 22, 2003)*

Question

How do we store sensitive data on an untrusted server?

Answer

Encrypt the data before sending it to the server

- hides all information about data
- the server performs only basic I/O functions and has no knowledge of what is stored

But

- users must download all data, decrypt and perform operations locally

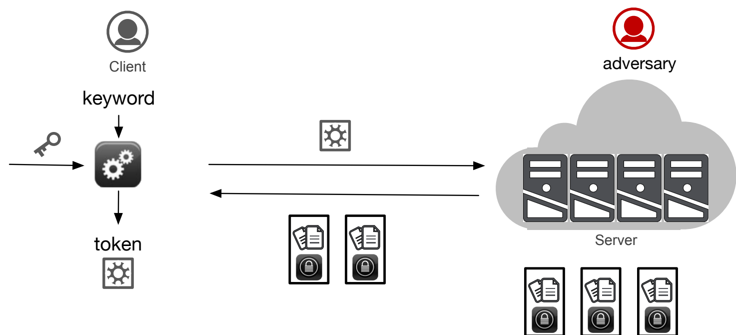
Can we let the server do more?

Searchable Encryption

- Typical scenario:
  - User has a collection of data items that each associates with a set of keywords, e.g. "new iPhone design", "list of CIA agents"
  - The data items and keywords are encrypted before sending to the server
- Functionality: the server should support the following type of queries:
  - "Find all data items that contain a given keyword"
- Confidentiality: Allow the server to help, but reveal as little information as possible
- First paper published in 2000, now 7,270 results in Google scholar (Feb, 2019)

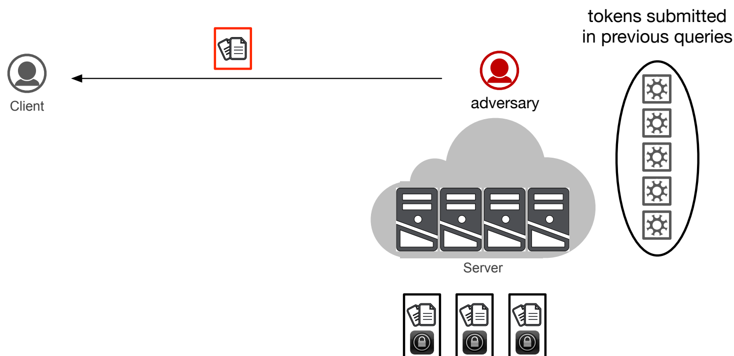
Query Privacy

- The server should not know the plaintext of keywords being queried.



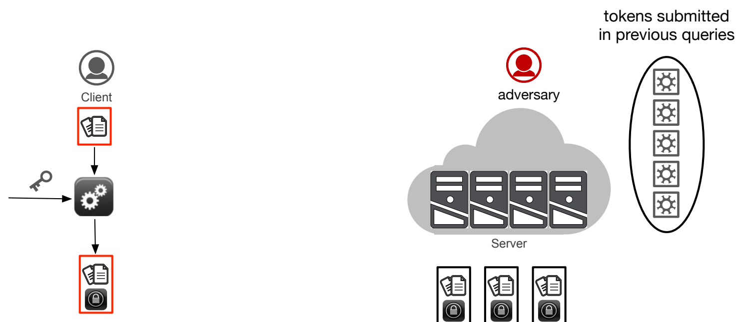
File Injection Attack

- In USENIX Security 2016, Zhang et.al. showed that query privacy can be totally broken by a file injection attack.



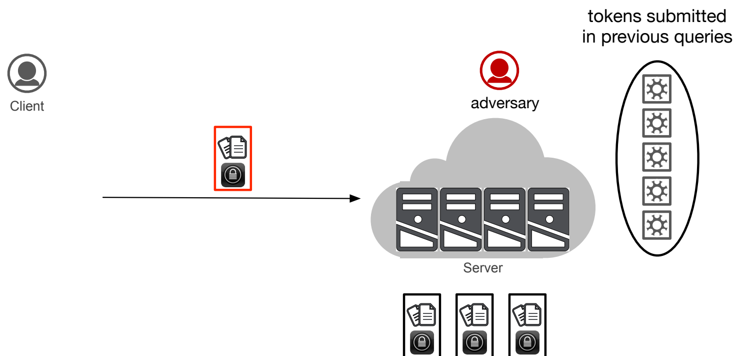
File Injection Attack

- In USENIX Security 2016, Zhang et.al. showed that query privacy can be totally broken by a file injection attack.



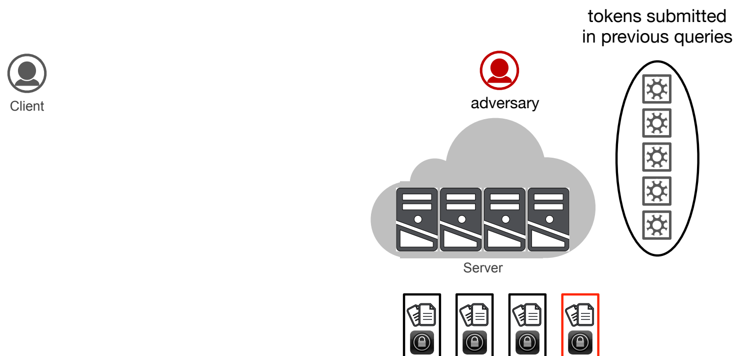
File Injection Attack

- In USENIX Security 2016, Zhang et.al. showed that query privacy can be totally broken by a file injection attack.



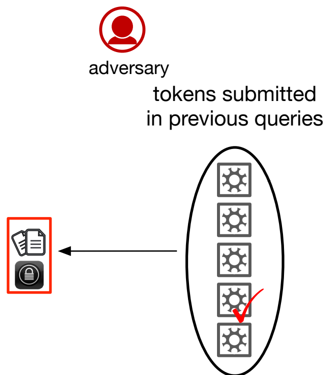
File Injection Attack

- In USENIX Security 2016, Zhang et.al. showed that query privacy can be totally broken by a file injection attack.



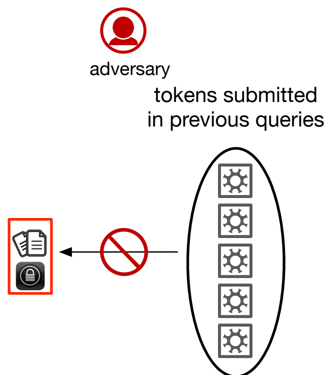
File Injection Attack

- In USENIX Security 2016, Zhang et.al. showed that query privacy can be totally broken by a file injection attack.



Forward Privacy

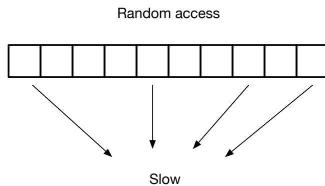
- Informally, the adversary should not be able to link newly inserted file in anyway to previous search queries
  - until the link being revealed in a future search query

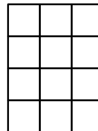


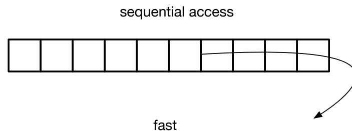
Prior Work on Forward Private Searchable Encryption

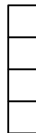
- Chang and Mitzenmacher 2005
  - search query size grows linearly in the number of updates,
  - communication cost for the search will eventually become unacceptable.
- Stefanov et al. 2014, Garg et al. 2016, Hoang et al. 2016
  - use ORAM like structures
  - communication cost is too high
  - not practical
- Sophos (Bost, CCS 2016)
  - first practical scheme
  - communication complexity is optimal ✓
  - search operation is public key based (slow) ✗
  - slow I/O due to access (read & write) to storage media is random ✗

I/O Efficiency









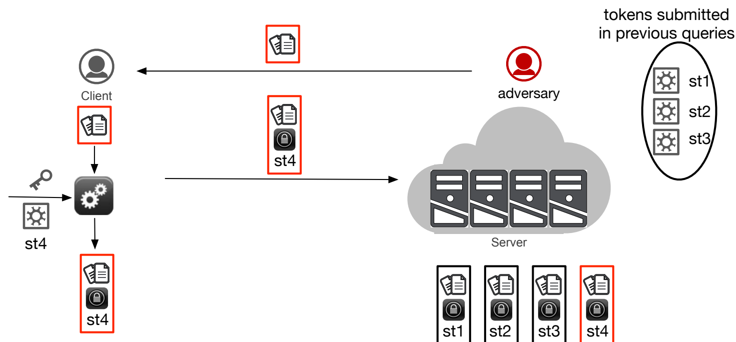
Our Contributions

- FAST (**F**orward priv**A**te searchable **S**ymmetric encryp**T**ion): Uses only symmetric key crypto
- FASTIO (FAST + **I**/O **O**ptimized): as the name suggests.

How Forward Privacy was Achieved in Sophos?

- The client stores a state, and update it every time inserting a new file.
- When inserting a new file, the client also inserts an index entry (to enable search)
  - The state is used as an input to encrypt the index entry
- The search token is essentially the latest state
- The server can compute all previous states from the token
- Each state matches the corresponding index entry.
- The function to update the state is public key based:
  - The server who has the public key can only go backward to the previous states of the given one – but not to later states
  - Only the client can evolve the state forward using the private key.

How Forward Privacy was Achieved in Sophos?



How About Symmetric Key Crypto?

- There is only one key, not two
- So Bost's strategy cannot be migrated to symmetric key crypto.

How did we solve it?

Setup$(\lambda, \perp; \perp)$

Client:
1: $k_s \xleftarrow{\$} \{0,1\}^\lambda$
2: $\mathbf{\Sigma} \leftarrow$ empty map
Server:
3: $\mathbf{T} \leftarrow$ empty map

Update$(k_s, \mathbf{\Sigma}, ind, w, op; \mathbf{T})$

Client:
4: $t_w \leftarrow F(k_s, h(w))$
5: $(st_c, c) \leftarrow \mathbf{\Sigma}[w]$
6: **if** $(st_c, c) = \perp$ **then**
7: $\quad st_0 \xleftarrow{\$} \{0,1\}^\lambda, c \leftarrow 0$
8: **end if**
9: $k_{c+1} \xleftarrow{\$} \{0,1\}^\lambda$
10: $st_{c+1} \leftarrow P(k_{c+1}, st_c)$
11: $\mathbf{\Sigma}[w] \leftarrow (st_{c+1}, c+1)$
12: $e \leftarrow (ind||op||k_{c+1}) \oplus H_2(t_w||st_{c+1})$
13: $u \leftarrow H_1(t_w||st_{c+1})$
14: send (u, e) to server
Server:
15: $\mathbf{T}[u] = e$

Search$(k_s, \mathbf{\Sigma}, w; \mathbf{T})$

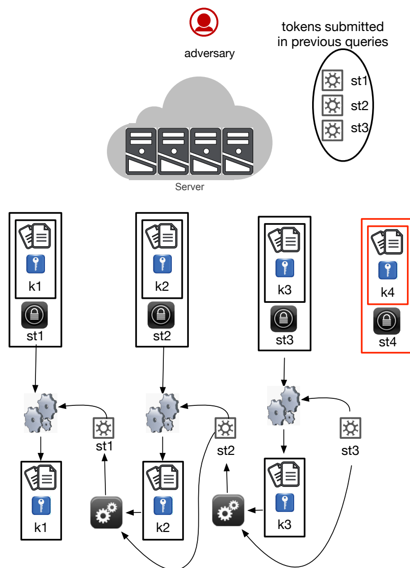
Client:
16: $t_w \leftarrow F(k_s, h(w))$
17: $(st_c, c) \leftarrow \mathbf{\Sigma}[w]$
18: **if** $(st_c, c) = \perp$ **then**
19: $\quad$ return $\emptyset$
20: **end if**
21: send (t_w, st_c, c) to Server
Server:
22: $\mathbf{ID}, \Delta \leftarrow \emptyset$
23: **for** $i = c$ **to** 1 **do**
24: $\quad u \leftarrow H_1(t_w||st_i)$
25: $\quad e \leftarrow \mathbf{T}[u]$
26: $\quad (ind, op, k_i) \leftarrow e \oplus H_2(t_w||st_i)$
27: $\quad$ **if** op = "del" **then**
28: $\quad\quad \Delta \leftarrow \Delta \cup \{ind\}$
29: $\quad$ **else if** op = "add" **then**
30: $\quad\quad$ **if** $ind \in \Delta$ **then**
31: $\quad\quad\quad \Delta \leftarrow \Delta - ind$
32: $\quad\quad$ **else**
33: $\quad\quad\quad \mathbf{ID} \leftarrow \mathbf{ID} \cup \{ind\}$
34: $\quad\quad$ **end if**
35: $\quad$ **end if**
36: $\quad st_{i-1} \leftarrow P^{-1}(k_i, st_i)$
37: **end for**
38: send **ID** to client

Fig. 1: Pseudocode of Protocols in FAST

Simplified Version

- Warning: this is not an accurate description
- When inserting the i-th file, we generate a fresh key k_i.
- The new state is the encryption of the previous state $st_i = E_{k_i}(st_{i-1})$.
- The index entry contains the pointer along with the new key, encrypted under the new state
  - $(pointer_i || k_i) \oplus H(st_i)$ (slightly simplified version)
- Given st_i, the server can compute $H(st_i)$, then recover $pointer_i || k_i$
- With k_i, the server can obtain the previous state $st_{i-1} = D_{k_i}(st_i)$
- With st_{i-1} the server can recover the state st_{i-2} and $pointer_{i-1}$
- Thus finds all files up to st_i
- But no way to get st_{i+1}

How did we solve it?



I/O Efficiency

- During search, the server needs to read the index from the disk
- The index entries are placed at random locations in an index file
- The ciphertext $(pointer_i||k_i) \oplus H(st_i)$ is 100% larger than the plaintext.
- Both are bad for I/O

FASTIO

```
Setup(λ, ⊥; ⊥)
    Client:
 1: k_s ←$ {0,1}^λ
 2: Σ ← empty map
    Server:
 3: T_e, T_c ← empty map

Update(k_s, Σ, ind, w, op; T_e)
    Client:
 4: (st, c) ← Σ[w]
 5: if (st, c) = ⊥ then
 6:     st ←$ {0,1}^λ
 7:     c ← 0
 8: end if
 9: u ← H_1(st||(c + 1))
10: e ← (ind||op) ⊕ H_2(st||(c + 1))
11: Σ[w] ← (st, c + 1)
12: send (u, e) to server
    Server:
13: T_e[u] = e

Search(k_s, Σ, w; T_e, T_c)
    Client:
14: (st, c) ← Σ[w]
15: if (st, c) = ⊥ then
16:     return ∅
17: end if
18: t_w ← F(k_s, h(w))
19: if c ≠ 0 then
20:     k_w ← st, st ←$ {0,1}^λ
21:     Σ[w] ← (st, 0)
22: else
23:     k_w ← ⊥
24: end if
25: send (t_w, k_w, c) to Server
    Server:
26: ID ← ∅
27: ID.add(T_c[t_w])
28: if k_w = ⊥ then
29:     return ID
30: end if
31: for i = 1 to c do
32:     u_i ← H_1(k_w||i)
33:     (ind, op) ← T_e[u_i] ⊕ H_2(k_w||i)
34:     if op = "del" then
35:         ID ← ID − {ind}
36:     else if op = "add" then
37:         ID ← ID ∪ {ind}
38:     end if
39:     delete T_e[u_i]
40: end for
41: T_c[t_w] ← ID
42: send ID to client
```

Fig. 3: Pseudocode of Protocols in FASTIO

High Level Ideas

- The server caches the search results after each search
  - This does not leak more information to the server
  - The server already knows the results, and has the token that can be used to re-generate it again
- The client only updates the state when a search query is performed
  - Instead of every file update
  - The states are truly random and independent
  - In between two search queries, sub-states are derived using a counter
  - No need to store keys because there are no keys

Consequences

- Entries from previous search now can be accessed sequentially, and only new entries after the last search still need random access;
- Small ciphertext expansion rate, less than 1%, ciphertext size is pointer size + 1 bit.

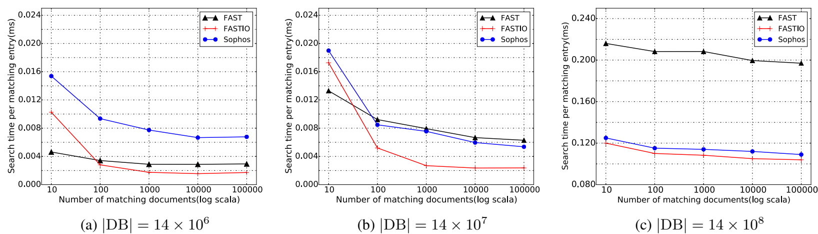
Update Efficiency

		FAST	FASTIO	Sophos
Local	Throughput (ops/s)	54060	76100	4890
	Single update time (ms)	0.018	0.013	0.20
WAN	Throughput (ops/s)	21650	31080	2990
	Single update time (ms)	0.046	0.032	0.334

TABLE 1: Update efficiency for FAST, FASTIO and Sophos

- WAN: server @ US west, client@ China

Search time per matched file



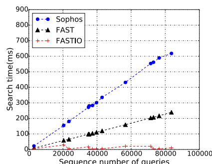
(a) $|\mathrm{DB}| = 14 \times 10^6$ (b) $|\mathrm{DB}| = 14 \times 10^7$ (c) $|\mathrm{DB}| = 14 \times 10^8$

Fig. 4: Search time per matched document for FAST, FASTIO and Sophos.

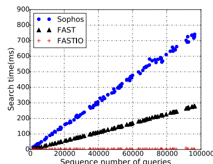
- without caching results

Search time based on random traces

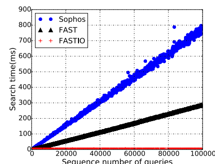
$$\alpha = \frac{\#search}{\#(search+update)}$$



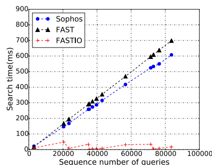
(a) $\alpha = 0.0001, |\mathrm{DB}| = 14 \times 10^6$



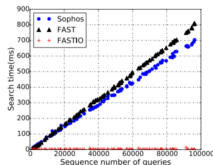
(b) $\alpha = 0.001, |\mathrm{DB}| = 14 \times 10^6$



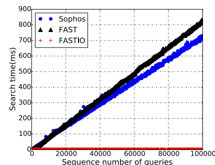
(c) $\alpha = 0.01, |\mathrm{DB}| = 14 \times 10^6$



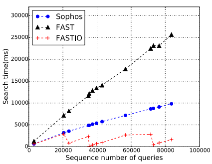
(d) $\alpha = 0.0001, |\mathrm{DB}| = 14 \times 10^7$



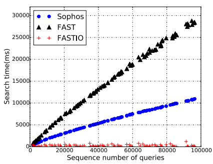
(e) $\alpha = 0.001, |\mathrm{DB}| = 14 \times 10^7$



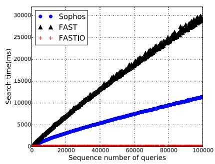
(f) $\alpha = 0.01, |\mathrm{DB}| = 14 \times 10^7$



(g) $\alpha = 0.0001, |\mathrm{DB}| = 14 \times 10^8$



(h) $\alpha = 0.001, |\mathrm{DB}| = 14 \times 10^8$



(i) $\alpha = 0.01, |\mathrm{DB}| = 14 \times 10^8$

Conclusion

- Designing a searchable encryption scheme with forward privacy and efficiency is not trivial.
- Questions:
    - What are the theoretical bounds for I/O efficiency?
    - Is forward privacy enough?
    - What does an "optimal" scheme in terms of security and efficiency look like?