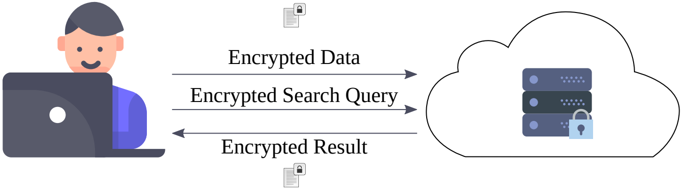
SGX IR

Secure Information Retrieval with Trusted Processors

Fahad Shaon, Murat Kantarcioglu

The University of Texas at Dallas

Problem - Secure Cloud based Information Retrieval



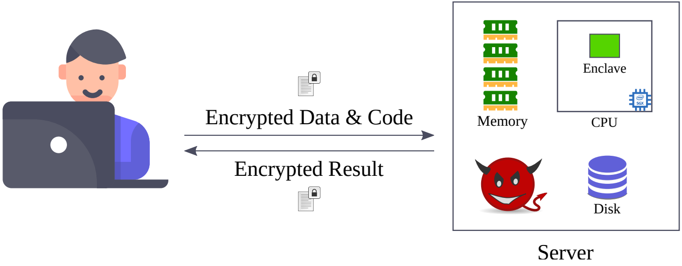
Build a secure information retrieval system

- User stores encrypted files in cloud server
- Perform selective retrieval

Build Block - Intel SGX

- We use **Intel SGX** - **S**oftware **G**uard E**x**tensions
- SGX is new Intel instruction set
- Allows us to create secure compartment inside *processor*, called **Enclave**
- Privileged softwares, such as, OS, Hypervisor, can not *directly* observe data and computation inside enclave

Threat Model - Intel SGX



Adversary can control hypervisor, OS, memory, disk of the server

State of The Art

- Relevant search or indexing systems that uses SGX - **HardIDX** (Fuhry et al., 2017), **Rearguard** (Sun et al., 2018), **Oblix** (Mishra et al., 2018), **Hardware-supported ORAM** (Hoang et al., 2019)
- These works mainly focus on building efficient data structures for searching using SGX
- Assume inverted index is built and/or build the index in client
- Did not look into ranked retrieval

Challenges - Access Pattern Leakage

Challenge: Access Pattern Leakage

- Adversary can observe memory accesses in SGX
- Memory access reveals about encrypted data (Islam, Kuzu, and Kantarcioglu, 2012; Naveed, Kamara, and Wright, 2015)

Challenges - Access Pattern Leakage

Challenge: Access Pattern Leakage

- Adversary can observe memory accesses in SGX
- Memory access reveals about encrypted data (Islam, Kuzu, and Kantarcioglu, 2012; Naveed, Kamara, and Wright, 2015)

Solution

- **Data Obliviousness** - we build custom data oblivious indexing algorithms

Data Obliviousness - Oblivious Select

- **Data Obliviousness:** Program executes **same path** for all input of same size

Data Obliviousness - Oblivious Select

- **Data Obliviousness:** Program executes **same path** for all input of same size
- **Example:** `x == y ?  a :  b`

Data Obliviousness - Oblivious Select

- **Data Obliviousness:** Program executes **same path** for all input of same size
- **Example:** `x == y ? a : b`

```
oblivousSelect(a, b, x, y):
...
mov %[x],%%eax
mov %[y],%%ebx
xor %%eax, %%ebx
...
mov %[a],%%ecx
mov %[b],%%edx
cmovz %%ecx,%%edx
...
mov %%edx, %[out]
```

Challenge - Memory Constraint

Challenge: Memory Constraint

- SGX (v1) only 90MB enclave

Challenge - Memory Constraint

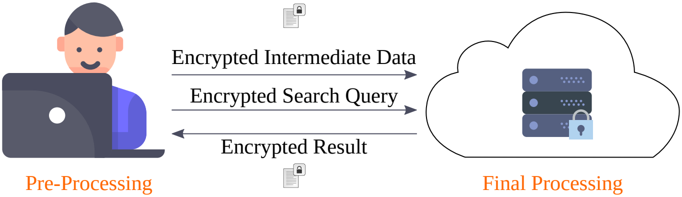
Challenge: Memory Constraint

- SGX (v1) only 90MB enclave

Solution

- **Blocking** - Break large data into small blocks
- We utilize SGXBigMatrix (Shaon et al., 2017) primitives
- BigMatrix handles the complexity of data blocking

Objectives - Summary

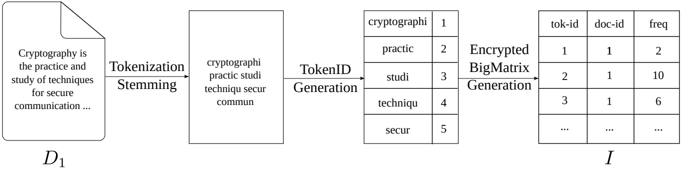


- Very **low** client side processing
- Build index securely **in the cloud** using SGX
- Build **data oblivious** algorithms
- Support **ranked retrieval**

SGX IR - Document and Query Types

- **Text Data**
  - Ranked document retrieval using TF-IDF (Token Frequency and Inverse Document Frequency)
- **Image Data**
  - Face recognition using Eigenface

Text Pre-Processing - Client



- We **tokenize** and **stem** the input text files
- We build a matrix I with $token\_id$, $document\_id$, and $frequency$ columns
- Finally, we encrypt I and upload
- **Single round** of read and write is required

Text Indexing - Server

tok-id	doc-id	freq
1	1	2
2	1	3
...	...	...
8	2	1
1	2	5
...	...	...
17	3	8
1	4	1
...	...	...

I

Indexing $\longrightarrow$

tok-id	count	sum
1	8	20
2	4	9
3	7	15
4	5	3
5	1	2
6	1	1
...	...	...
#	#	#
...	...	...

$\mathcal{U}'$

tok-id	doc-id	freq	
$\sigma(1,0)$	1	2	b
...	...	...	
$\sigma(1,1)$	7	2	b
...	...	...	
$\sigma(2,0)$	1	3	b
...	...	...	
$\sigma(2,1)$	9	10	b
...	...	...	
$\sigma(3,0)$	9	10	

$\mathcal{T}$

- Input I, we output two matrices
- U' containing total frequencies of the tokens, for **IDF** calculation
- $\mathcal{T}$ containing equal length blocks of token to document frequency mapping for **TF** calculation

Text Indexing - IDF - Server

tok-id	doc-id	freq
1	1	2
2	1	3
...	...	...
8	2	1
1	2	5
...	...	...
17	3	8
1	4	1
...	...	...

I

$\xrightarrow{\text{Sort}}$

tok-id	doc-id	freq
1	1	2
1	2	5
1	4	1
...	...	...
2	1	3
2	5	10
...	...	...
3	6	4
...	...	...

I'

$\xrightarrow{\text{Count \& Sum}}$

tok-id	count	sum
1	0	0
#	#	#
...	...	...
2	8	20
#	#	#
...	...	...
3	4	9
#	#	#
...	...	...

$\mathcal{U}$

Sort and Adjust $\longrightarrow$

tok-id	count	sum
1	8	20
2	4	9
3	7	15
4	5	3
5	1	2
6	1	1
...	...	...
#	#	#
...	...	...

$\mathcal{U}'$

- $I' \leftarrow$ **Obliviously sort** I on $token\_id$ column
- We generate $\mathcal{U}$, to keep $count$ and sum of frequencies
  - $c \leftarrow I'[i].token\_id \neq I'[i-1].token\_id$
  - $\mathcal{U}[i].sum \leftarrow obliviousSelect(sum, \#, 1, c)$
  - $sum \leftarrow obliviousSelect(sum, 0, 1, c) + I[i].frequency$
- Finally, we sort this matrix so that the dummy entries go to the bottom

Text Indexing - TF - Block Size Optimization

- We can read document frequency of tokens from matrix I'
- This will reveal number of documents having a specific token
- So, we split I' into equal length blocks
- We optimize block size b from $count$ column of $\mathcal{U}'$ using technique outline in (Shaon and Kantarcioglu, 2016)
  - We assume the frequency follow Pareto distribution
  - Mathematically find the value minimize the padding

Text Indexing - TF - Padding Generation

We regenerate token id with bucket number function σ

tok-id	doc-id	freq
1	1	2
1	2	5
1	4	1
...	...	...
2	1	3
2	5	10
...	...	...
3	6	4
...	...	...

I'

Regenerate TokenId $\longrightarrow$

tok-id	doc-id	freq	
$\sigma(1,0)$	1	2	b
...	...	...	
$\sigma(1,1)$	7	2	$\mathcal{U}'[1]_{count}\%b$
...	...	...	
$\sigma(2,0)$	1	3	b
...	...	...	
$\sigma(2,1)$	9	10	$\mathcal{U}'[2]_{count}\%b$
...	...	...	
$\sigma(3,0)$	9	10	

J

We generate padding

tok-id	count	sum
1	8	20
2	4	9
3	7	15
4	5	3
5	1	2
6	1	1
...	...	...
#	#	#
...	...	...

$\mathcal{U}'$

Generate Padding Rows $\longrightarrow$

tok-id	doc-id	freq	
$\sigma(1,1)$	#	#	$b - \mathcal{U}'[1].count\%b$
...	...	...	
#	#	#	$\mathcal{U}'[1].count\%b$
...	...	...	
$\sigma(2,1)$	#	#	$b - \mathcal{U}'[2].count\%b$
...	...	...	
#	#	#	$\mathcal{U}'[2].count\%b$
...	...	...	
$\sigma(3,1)$	#	#	

X

Finally we merge and sort X and J to get the output $\mathcal{T}$ matrix.

TF - IDF Calculation

- On $\mathcal{T}$ we run **term frequency** functions - (log normalization)

$$1 + log(tf_{t,d})$$

- On $\mathcal{U}'$ we run **document** frequency functions, such as, IDF

$$log\frac{N}{df_t}$$

- Query result we use $\mathcal{T}$ for TF and $\mathcal{U}'$ for IDF

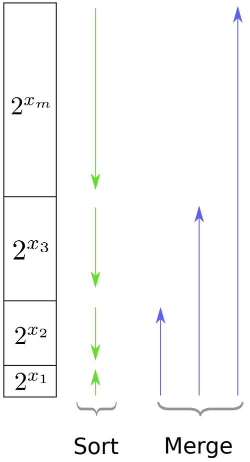
Bitonic Sorting of Arbitrary Input Size

- Sorting is one of the most frequently used operations
- We use **arbitrary length** Bitonic sort version (Lang, 1998)
- However, existing definition is recursive
- Not suitable for memory constrained environments like SGX
- So, we propose a **non-recursive** algorithm **without** using stack

Bitonic Sort Non Recursive Algorithm - Concept

Concept

- We can express a number as $N = 2^{x_m} + ... + 2^{x_3} + 2^{x_2} + 2^{x_1}$
- Merge network can sort a descending and an ascending block into ascending order block
- We sort then merge from smallest to biggest block



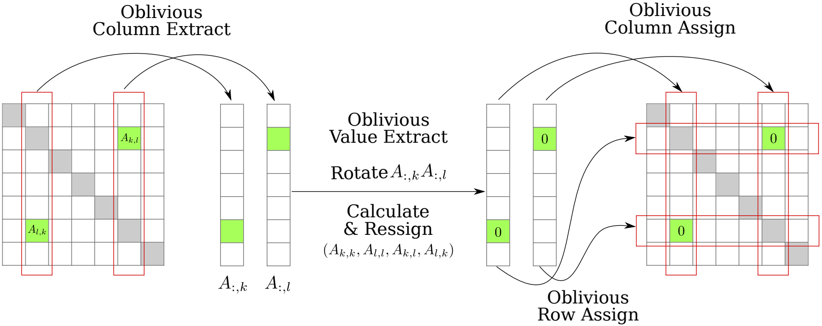
Bitonic Sort Non Recursive Algorithm

for d = 0 **to** $\lceil log_2(N) \rceil$ **do**
  if $((N >> d)\ \&\ 1) \neq 0$ **then**
    $start \leftarrow (-1 << (d+1))\ \&\ N$
    $size \leftarrow 1 << d$
    $dir \leftarrow (size\ \&\ N\ \& -N) \neq 0$
    $bitonicSort2K(start, size, dir)$
    if $!dir$ **then**
      $bitonicMerge(start, N - start, 1)$
    end if
  end if
end for

Face recognition indexing

- We adopt **EigenFace**
- Pre-processing and matching face are simple matrix operations
- Core problem to solve **obliviously** is eigenvector calculation
- We adopt **Jacobi** method of eigenvector calculation

Eigenvector calculation - Jacobi method



We find the max off-diagonal element at $A_{k,l}$, then rotate column k and l. Repeat until A becomes diagonal. The diagonal values are eigenvalues.

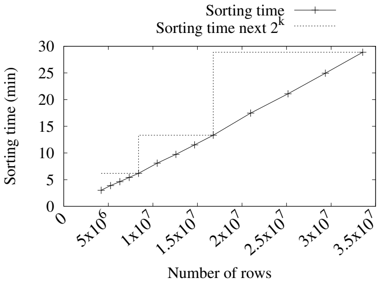
Experimental Evaluations

We implemented a prototype using Intel SGX SDK 2.6 for Linux

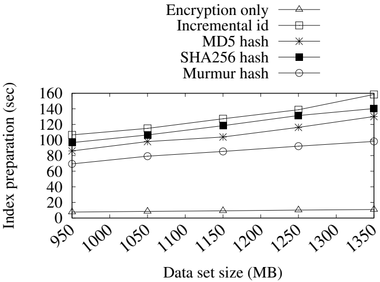
Setup

- **Processor** Intel Xeon E3-1270
- **Memory** 64GB
- **OS** Ubuntu 18.04
- **SGX SDK Version** 2.6 for Linux

Experimental Results - Bitonic Sort and Text Indexing

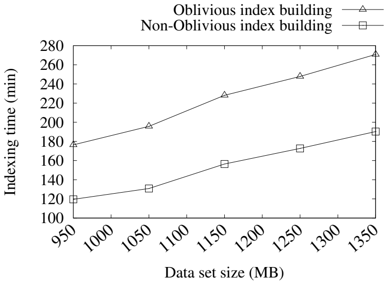


Bitonic sort

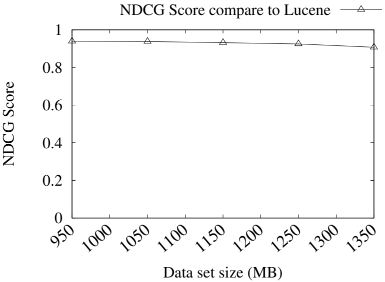


Client end processing cost on Enron Dataset

Experimental Results - Text Indexing

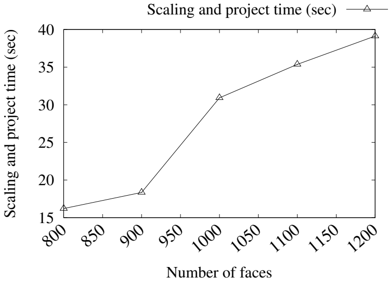


SGX index processing on Enron Dataset

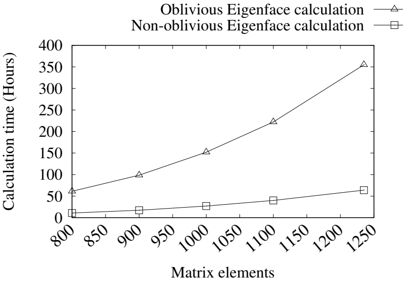


NDCG results compare to Apace Lucene on Enron Dataset

Experimental Result - Eigenvector Calculation



Pre-processing overhead



Eigenvector calculation time

Thank you

Questions / Comments

- Fahad Shaon - fahad.shaon@utdallas.edu
- Murat Kantarcioglu - muratk@utdallas.edu

Acknowledgments: This research is supported in part by NIH award 1R01HG006844, NSF awards CICI-1547324, IIS-1633331, CNS-1837627, OAC-1828467 and ARO award W911NF-17-1-0356.

References I

- Fuhry, Benny et al. (2017). “HardIDX: Practical and secure index with SGX”. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, pp. 386–408.
- Hoang, Thang et al. (2019). “Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset”. In: *Proceedings on Privacy Enhancing Technologies* 2019.1, pp. 172–191.
- Islam, Mohammad Saiful, Mehmet Kuzu, and Murat Kantarcioglu (2012). “Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.”. In: *NDSS*. Vol. 20, p. 12.
- Lang, H.W. (1998). *Bitonic sorting network for n not a power of 2*. Accessed 05/31/2020. URL: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm.

References II

Mishra, Pratyush et al. (2018). "Oblix: An efficient oblivious search index". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 279–296.

Naveed, Muhammad, Seny Kamara, and Charles V Wright (2015). "Inference attacks on property-preserving encrypted databases". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 644–655.

Shaon, Fahad and Murat Kantarcioglu (2016). "A practical framework for executing complex queries over encrypted multimedia data". In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, pp. 179–195.

References III

Shaon, Fahad et al. (2017). “SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS 17. Dallas, Texas, USA: Association for Computing Machinery, 12111228. ISBN: 9781450349468. DOI: 10.1145/3133956.3134095.

Sun, Wenhai et al. (2018). “REARGUARD: Secure keyword search using trusted hardware”. In: *IEEE INFORM*.