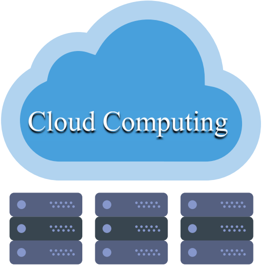
Secure Cloud Data Analytics with Trusted Processors

Fahad Shaon

The University of Texas at Dallas

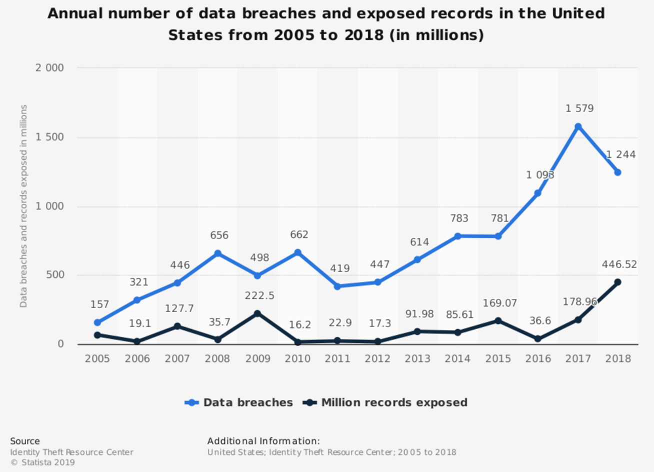
Introduction

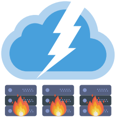
- Cloud computing is ubiquitous
  - No upfront infrastructure cost
  - Speed
  - Scale as needed
- Market is still growing fast
  - Market size: 2018 - $\$182.4$B, 2022 - $\$331.2$B



Cloud computing - Issue

- **Security breaches** are very frequent now-a-days





Security Breach in Cloud Context

- Third-party vendor system had **publicly accessible** AWS S3 bucket
- **Impact:** **6 million** records were compromised
- **Solution:** Enable encryption in the S3 bucket

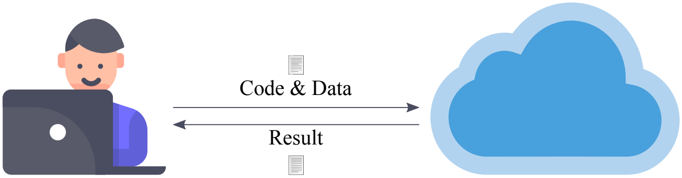
THE VERGE

Verizon partner data breach exposes millions of customer records

Accessed through an unprotected Amazon S3 storage server

By Dani Deahl | @danideahl | Jul 12, 2017, 7:53pm EDT

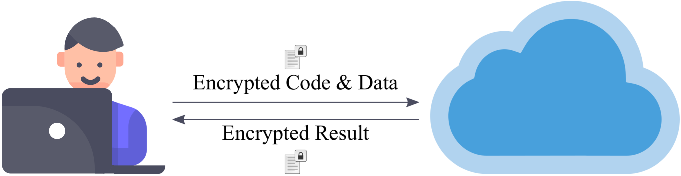
Data Analytics on Cloud - Issues



Some Issues

- **Sensitive** data (e.g., medical, financial data) exposure
- Highly vulnerable to **insider attack**
- Service provider can **observe** the data and patterns

One Solution - Secure Data Analytics on Cloud



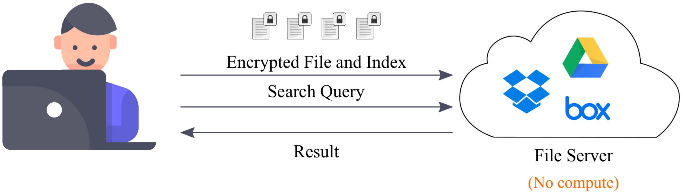
- We **do not trust** the cloud, unless it has trusted processor
- We only outsource **encrypted data**
- However, encrypted data is **difficult** to analyze

Chapters

- A Practical Framework for Executing Complex Queries over Encrypted Multimedia Data [DBSec 2016]
- SGX BigMatrix: A Practical Encrypted Data Analytic Framework with Trusted Processors [CCS 2017]
- SGX IR: Secure Information Retrieval with Trusted Processors

A Practical Framework for Executing Complex Queries over Encrypted Multimedia Data

Problem Definition



- User wants to **safely** store documents in cloud storage
- User also wants to **search** the uploaded file
- We are using servers **without** computation capability, such as, Google Drive, Dropbox, Box, Amazon S3, etc.

Searchable Encryption - Introduction

- Given a set of documents we encrypt the documents and create an **encrypted inverted index**.
- Then encrypted document and inverted index is uploaded to server
- To search we create special **trapdoor** from the input keyword and sent to server
- Server then **find** documents using the trapdoor.

Target



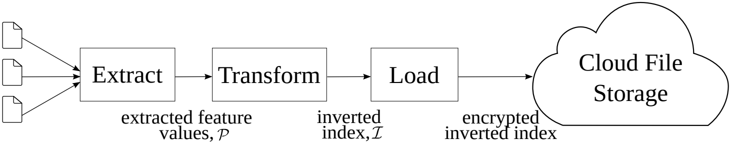




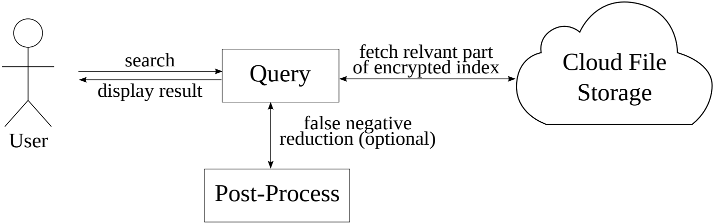

Find photos of Jhon taken in last summer in Hawaii during sunset?

- Restriction: server does **not** support custom computation

Our Solution - ETL QP Frmework



(a) Index creation, encryption and upload

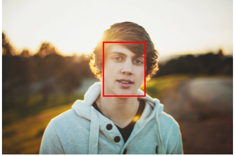


(b) Query and post-process phase to search content

Extract

- Necessary features of documents are extracted in this phase.
- **Features extractors** are defined based on application need.
- Features can be defined by the user
- Output of this phase is feature, value pairs per document.

Extract - example



D_i

Features

- $(Location,$ `(2116'42"N, 15750'02.8"W)`$)$
- $(CreatedAt,$ `6/7/2018 7:00pm`$)$
- $(Aperture,$ `2.4`$)$
- $(ShutterSpeed,$ `1/100`$)$
- $(Faces,$ [`(X:60, Y:34, H: 25, W: 32)`]$)$

We extract necessary features(i.e. meta-data) from images and output sequence of tuples in the form

$$\langle id(D_i), (f_a, v_\alpha), (f_b, b_\beta), (f_c, v_\gamma)\rangle$$

Transform

- Generate signature value based on **feature-value combination**
- **Example:** Location
  - Input: $\langle id(D_i), (Location, (\texttt{longitude, latitude})\rangle$
  - We look up the address of the geo location value and generate search signatures based on country, state, city, address, etc.
  - $S_1 = H(`Location' \mid\mid `Country' \mid\mid \texttt{Country\_Value})$
  - $S_2 = H(`Location' \mid\mid `State' \mid\mid \texttt{State\_Value})$
  - **Output:** $\langle S_1, id(D_i)\rangle, \langle S_2, id(D_i)\rangle$

Transform output example

Search Signature	Document ID List
s_1	$id(D_1), id(D_3)$
s_2	$id(D_1), id(D_2), id(D_4)$
s_3	$id(D_1), id(D_3), id(D_4)$
s_4	$id(D_2)$
s_5	$id(D_4)$

(d) Inverted index, $\mathcal{I}$

Load - Overview

- Here we encrypt and load the inverted index to cloud file server.
- We observe that distribution of the length of the document list of search signatures can be approximated with **Pareto distribution**.
- Based on that we further block the document list (details in full version)
- Then we generate search signatures of the **blocked document list**.
- And keep certain information in a cache.

Load - Algorithm

Algorithm 1 Load encrypted index

Require: K = Master key, $\mathcal{I}$ = Inverted index of search signatures, $\mathcal{C}$ = Synchronized cache, K_C = encryption key for cache, $\mathcal{Z}$ = File storage server.

$b \leftarrow \mathit{optimize}(\mathcal{I})$

for all signature s in $\mathcal{I}$ **do**

    $\mathit{blocks}_s \leftarrow \lceil \frac{|\mathcal{I}[s]|}{b} \rceil$

    for $j = 1 \rightarrow \mathit{blocks}_s$ **do**

        $T_j^s \leftarrow H(K, s \,||\, j \,||\, C_1)$, $K_j^s \leftarrow H(K, s \,||\, j \,||\, C_2)$

        $\mathit{sub} \leftarrow \mathcal{I}[s].\mathit{slice}((j-1) \times b, j \times b)$

        $\mathcal{E}[T_j^s] \leftarrow \varphi(K_j^s, \mathit{pad}(\mathit{sub}))$

    end for

    $\mathcal{C}.\mathit{freq}[s] \leftarrow |\mathcal{I}[s]|$

end for

for all trapdoor t in $\mathcal{E}$ **do**

    $\mathcal{Z}.\mathit{write}(t, \mathcal{E}[t])$

end for

$C_{sig} \leftarrow H(K_C \,||\, C_3, 1)$

$\mathcal{Z}.\mathit{write}(C_{sig}, \varphi(K_C, \mathcal{C}))$

Query and Post Process - Overview

- Given a query we first extract and transform it
- Next we generate search signatures
- Generate trapdoors
- Get those trapdoor related information
- Then decrypt the document ids
- Finally, remove false positives (if necessary)

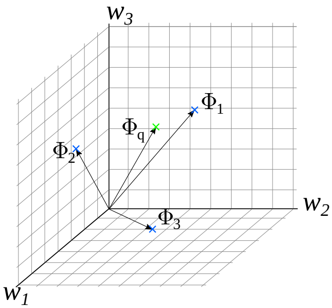
Query

Algorithm 2 Query

Require: K = Master key, q = Query, b = block size, $\mathcal{Z}$ = File storage server
$\mathcal{Q} \leftarrow$ Extract and Transform q
for all search signatures s in $\mathcal{Q}$ **do**
    $blocks_s \leftarrow \lceil \frac{\mathcal{C}.freq[s]}{b} \rceil$
    for $i = 1 \rightarrow blocks_s$ **do**
        $T_j^s \leftarrow H(K, s \,||\, j \,||\, C_1)$, $K_j^s \leftarrow H(K, s \,||\, j \,||\, C_2)$
        $L \leftarrow \mathcal{Z}.read(T_j^s)$
        add $\varphi^{-1}(K_j^s, L)$ in $\mathcal{R}[s]$
    end for
end for
return $\mathcal{R}$

EigenFace

- We normalize input face images $A = [\Phi_1 \ \Phi_2 \ldots \Phi_M]$
- Find eigen vectors (u_j) of $A^T A$
- Get top K eigen vectors
- Represent input $\Phi_i = \sum_{j=1}^{K} w_j u_j$, where weight $w_j = u_j^T \Phi_i$
- Calculate $\Omega_i = \left[w_1 \ w_2 \ \ldots \ w_k \right]^T$, which is the projection in **eigen space**.
- To match, we normalize (Φ_q), project (Ω_q), and compute distance

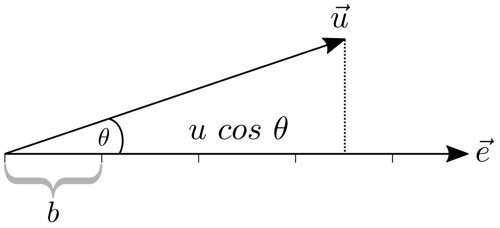


Faces in Eigen Space

Encrypted Eigenface Recognition - ETL

- **Extract:** Find face locations in image
  - $id(D_1) : \langle `Face', \text{(X:10px, Y:12px, H: 120px, W: 120px)} \rangle$
- **Transform:**
  - Convert face to point in EigenFace Plane ω
  - Define Euclidean LSH function
  - $bucket\_ids =$ Find LSH bucket ids of ω
  - $search\_signatures = generate\_signatures(bucket\_ids)$
- **Load:**
  - Upload $search\_signatures$ and document assignments

Euclidean LSH



- Random LSH vector, $\vec{e}$
- Input point/vector, $\vec{u}$
- LSH line bucket length, b
- $BucketId = Hash(\frac{u \times cos\ \theta}{b}, \hat{e})$

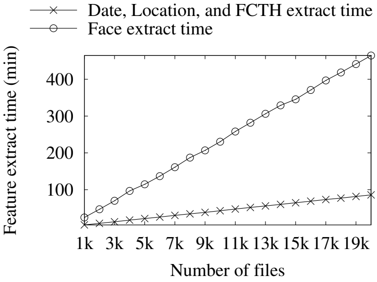
Encrypted Eigenface Recognition - QP

- **Query:**
  - Given a new Face
  - Convert to a point in eigen plane point
  - Create $bucket\_ids$ of previously defined LSH schema.
  - Create $search\_signatures$ of the $bucket\_ids$
  - Now search the search $search\_signatures$ in the encrypted index
- **Post Process:**
  - Remove the false positives due to LSH

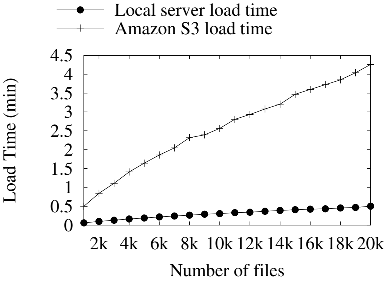
Experiment - Features and DataSet

- Our prototype image storage system can handle **4** types of features
  - **Location**
    - Find images based on location
  - **Time**
    - Find images that are taken on a specific time or in a time range
  - **Texture and Color**
    - Find images that are similar, e.g., images of sunset, sky, etc.
  - **Face**
    - Find images of a particular person.
- **Dataset:** Randomly selected *20,109* images from *YFCC100M* dataset.

Load time and Index Size

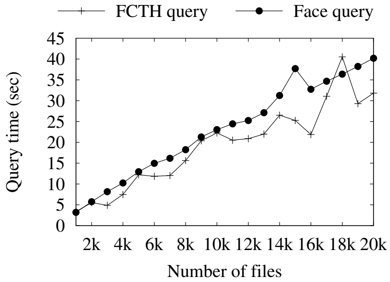


Extract time

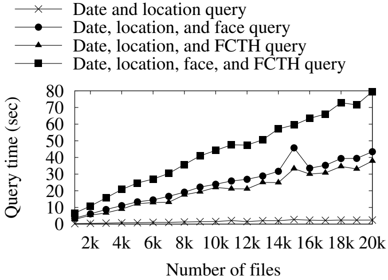


Load time

Experiment - Query Time



Similarity Query and
Face recognition Time



Combination Query Time

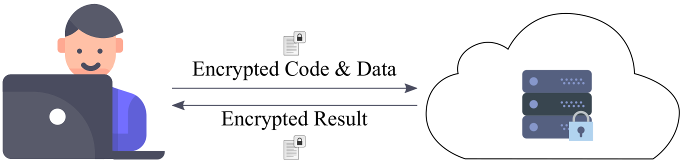
Conclusion

- We have proposed a practical framework for performing complex queries over encrypted multimedia data.

SGX BigMatrix

A Practical Encrypted Data Analytic Framework with Trusted Processors

Secure Data Analytics - with Outsourced Computation



- We outsource encrypted *sensitive* data
- We also want to perform secure **computation** in cloud

Secure Data Analytics - Approaches

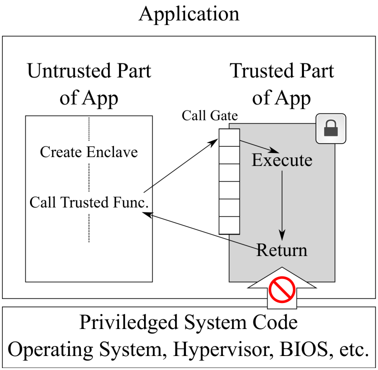
Pure Cryptographic Approach

- Secure Multi-party Computation
- Provides highest level of security
- High computational cost
- Impractical for large data processing

Trusted Hardware

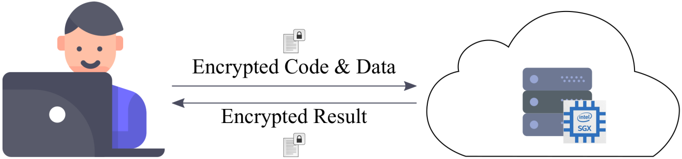
- Cost effective
- Provides reasonable security
- Intel SGX is available in all new processors
- Needs careful consideration of side channel attacks

Background - Intel SGX Application



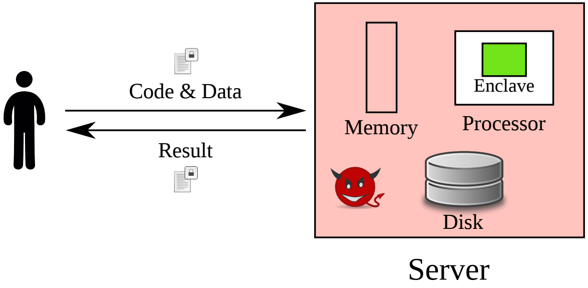
- We only trust the processor and the code inside the enclave (Intel, 2015)

Background - Intel SGX Impact



- We can outsource computation securely
- No need to trust the cloud provider (i.e. Hypervisor, OS, Cloud administrators)

Threat Model



- Adversary can control OS (i.e. memory, disk, networking)
- Adversary can *not* temper with enclave code
- Adversary can *not* observe CPU register content

Challenges - Obliviousness

Challenge: Access Pattern Leakage

- SGX uses system memory, which is controlled by the adversary
- Adversary can observe memory accesses
- Memory access reveals a lot about the data (Islam, Kuzu, and Kantarcioglu, 2012; Naveed, Kamara, and Wright, 2015)

Challenges - Obliviousness

Challenge: Access Pattern Leakage

- SGX uses system memory, which is controlled by the adversary
- Adversary can observe memory accesses
- Memory access reveals a lot about the data (Islam, Kuzu, and Kantarcioglu, 2012; Naveed, Kamara, and Wright, 2015)

Solution

- To reduce information leakage we ensure **Data Obliviousness**

Data Obliviousness - Example

- Program executes **same path** for all input of same size

Data Obliviousness - Example

- Program executes **same path** for all input of same size

Example: Non-Oblivious swap method of Bitonic sort

```
if (dir == (arr[i] > arr[j])) {
    int h = arr[i];
    arr[i] = arr[j];
    arr[j] = h;
}
```

Data Obliviousness - Example (Cont.)

Example: Oblivious swap method of Bitonic sort

```
int x = arr[i];
int y = arr[j];

_asm{
...
mov eax, x
mov ebx, y
mov ecx, dir

cmp ebx, eax
setg dl

xor edx, ecx
```

```
mov eax, x
mov ecx, y

mov ebx, y
mov edx, x

cmovz eax, ecx
cmovz ebx, edx

mov [x], eax
mov [y], ebx
}
```

Data Obliviousness - Challenges

Challenge

- Building data obliviousness solution is non-trivial
- Requires a lot of time and effort

Data Obliviousness - Challenges

Challenge

- Building data obliviousness solution is non-trivial
- Requires a lot of time and effort

Solution

- We provide our own python (NumPy, Pandas) inspired **language** that ensures data obliviousness

Data Oblivious - Vectorization

- We removed **if** and emphasis on vectorization

Example: Compute average income of people with $age >= 50$

```
sum = 0, count = 0
for i = 0 to Persons.length:
    if Persons[i].age >= 50:
        count++
        sum += Persons[i].income

print sum / count
```

Data Oblivious - Example

Example: Compute average income of people with $age >= 50$

```
S = where(Person, "Persons['age'] >= 50")
print (S .* Persons['income'] ) / sum(S)
```

Challenge - Memory constraint

Challenge

- Current version of SGX (v1) allows only **90MB** of memory allocation

Challenge - Memory constraint

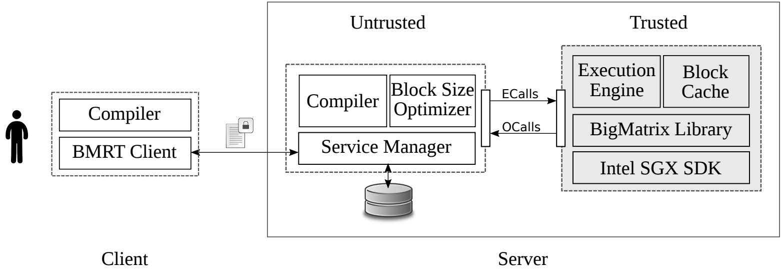
Challenge

- Current version of SGX (v1) allows only **90MB** of memory allocation

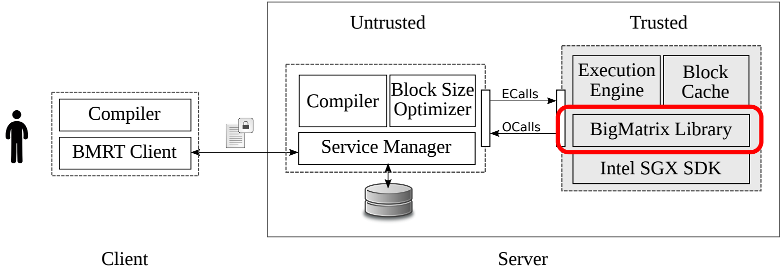
Solution

- We build flexible data **blocking mechanism** with efficient and secure caching
- We build matrix manipulation library that supports blocking and we call the abstraction **BigMatrix**

System Overview - SGX BigMatrix



SGX BigMatrix



SGX BigMatrix - BigMatrix Library

BigMatrix Library

Operations in BigMatrix Library

- Data access operations - `load`, `publish`, `get_row`, etc.
- Matrix Operations - `inverse`, `multiply`, `element_wise`, `transpose`, etc.
- Relational Algebra Operations - `where`, `sort`, `join`, etc.
- Data generation operations - `rand`, `zeros`, etc.
- Statistical Operations - `norm`, `var`

BigMatrix Library - Security Properties

- All the operations are **data oblivious**
- All the operations supports **blocking**
- We proved that combination of data oblivious operations is also data oblivious (in *Section 4*)
- Data oblivious and blocking aware implementation details in the paper

BigMatrix Library - Trace

- Each operation has fixed **trace**
- **Trace** is the information disclosed to adversary during execution
- For example: operation type, input and output data size

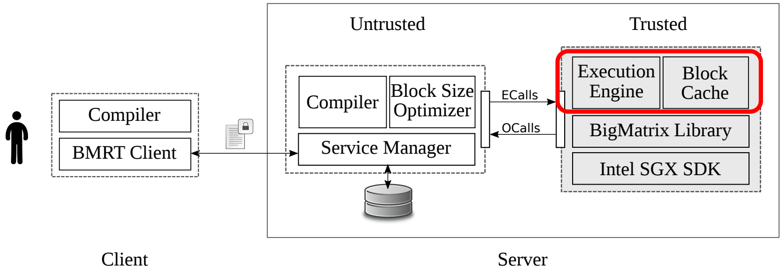
BigMatrix Library - Trace

- Each operation has fixed **trace**
- **Trace** is the information disclosed to adversary during execution
- For example: operation type, input and output data size

Example: Trace of Matrix Multiplication $C = A * B$

- Instruction type (i.e. `multiplication`)
- Input Matrices size (i.e., $A.rows, A.cols, B.rows, B.cols$)
- Output Matrix size (i.e., $C.rows, C.cols$)
- Block size
- *Oblivious* memory read and write sequences, which does not depend on data content

Exec. Engine & Block Cache



SGX BigMatrix - Execution Engine and Block Cache

Exec. Engine & Block Cache

Execution Engine

- Execute BigMatrix library operations
- Parse instruction in the form of

```
Var ASSIGN Operation (Var, Var, ...)
```

- Process sequence of instructions
- Maintain intermediate states required to execute complex program, such as, variable to BigMatrix assignments

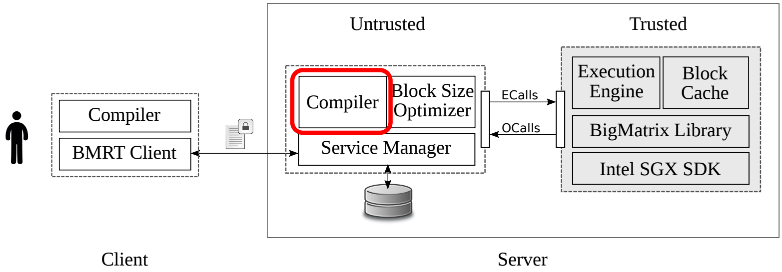
Block Cache

- Help with the decision when to remove a block from memory based on next sequence of instructions

Exec. Engine & Block Cache - Security Properties

- *Execution Engine* and *Block Cache* is also data oblivious given the input program is data oblivious
- Compiler warns about potential data leakage
- Adversary can not infer anything more about data, apart from the trace of all the operations

Compiler



SGX BigMatrix - Compiler

Compiler

- Compiles our python inspired language into basic command
- It ensures *data obliviousness* by removing support for *if*
- We emphasis on operation vectorization

Input: Linear Regression

```
x = load('path/to/X_Matrix')
y = load('path/to/Y_Matrix')
xt = transpose(x)
theta = inverse(xt * x) * xt * y
publish(theta)
```

Output: Linear Regression

```
x = load(X_Matrix_ID)
y = load(Y_Matrix_ID)
xt = transpose(x)
t1 = multiply(xt, x)
unset(x)
t2 = inverse(t1)
unset(t1)
t3 = multiply(t2, xt)
unset(xt)
unset(t2)
theta = multiply(t3, y)
unset(y)
unset(t3)
```

Compiler - Track data leakage

- We report against accidental data leakage through **trace**
- We check if any *sensitive data* is used in trace of any operation
- In our system, sensitive data - content of any BigMatrix, content of intermediate variables

Example

```
X = load('path/to/X_Matrix')
s = count(where(X[1] >= 0))
Y = zeros(s, 1)
publish(Y)
```

We report that `zeros` operation revealing sensitive data `s`

SQL Support

- We also support basic SQL

Input

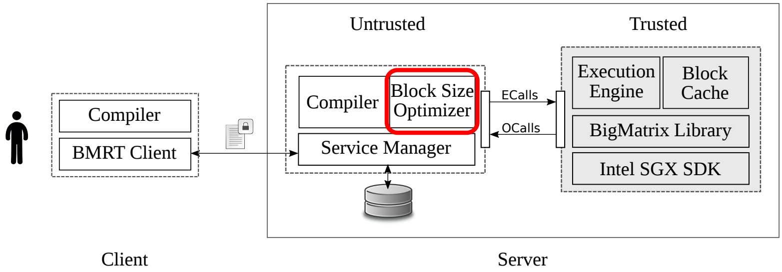
```
I = sql('SELECT *
FROM person p
JOIN person_income pi (1)
ON p.id = pi.id
WHERE p.age > 50
AND pi.income > 100000')
```

SQL Support (Cont.)

Output

```
t1 = where(person, 'C:3;V:50;O:=')
    # person.age is in column 3
t2 = zeros(person.rows, 2)
t3 = get_column(person, 0)
    # person.id is in column 0
set_column(t2, 0, t3)
set_column(t2, 1, t1)
t4 = where(person_income, 'C:1;V:100000;O:=')
t5 = zeros(person_income.rows, 2)
t6 = get_column(person_income, 0)
    # person_income.id is in column 0
set_column(t5, 1, t4)
set_column(t5, 0, t6)
A = join(t2, t5, 'c:t2.0;c:t2.0;O:=', 1)
```

Block Size Optimizer



SGX BigMatrix - Block Size Optimizer

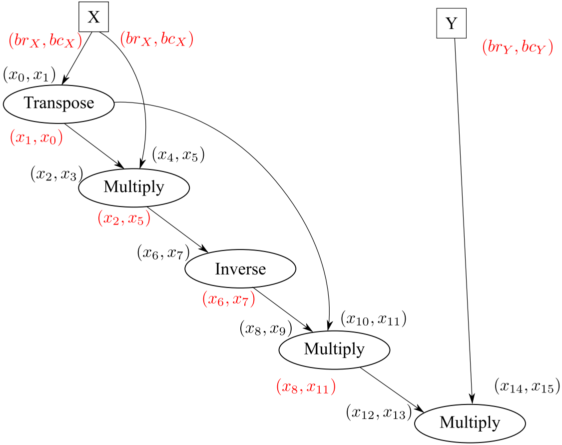
Block Size Optimizer - Intro & Design Decisions

- We observed that input block size has impact on performances of the system
- Adversary doesn't gain any knowledge about data based on block size
- So, we find optimum block size for each instruction before executing a program
- We explicitly do not want to perform optimization inside enclave because
  - Optimization libraries are large and complex, which can introduce unintended security flaws
  - Any efficient optimization algorithm will reveal information about data
  - So we only perform optimization on *trace* data, nothing else

Block Size Optimizer - Overview

- We generate DAG of execution graph
  - Internal nodes represent operations
  - Edges represent block conversions
- We know cost for each operation for different matrix and block size
- Given input matrix sizes we can find optimized block size
- We can convert one block configuration to another and know the cost of conversion

Block Size Optimizer - Example - Linear Regression



- Execution graph (DAG) of $\Theta = (X^T X)^{-1} X^T Y$ in liner regression training phase

Block Size Optimizer - Example - LR Cost Function

$$
\begin{aligned}
Cost = {} & Convert(X, (br_X, bc_X), (x_0, x_1)) \\
& + OP\_Cost('Transpose', X, (x_0, x_1)) \\
& + Convert(X^T, (x_1, x_0), (x_2, x_3)) \\
& + Convert(X, (br_X, bc_X), (x_4, x_5)) \\
& + OP\_Cost('Multiply', [X^T, X], [(x_2, x_3), (x_4, x_5)]) \\
& + ...
\end{aligned}
$$

We convert this into integer programming and solve it for all the x_n variables.

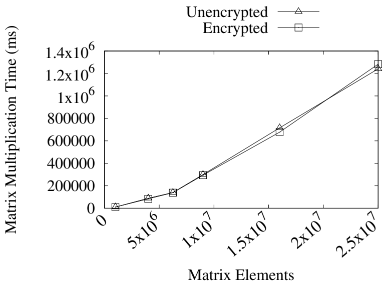
Experimental Evaluations

We implemented a prototype using Intel SGX SDK and observe performance of different operations

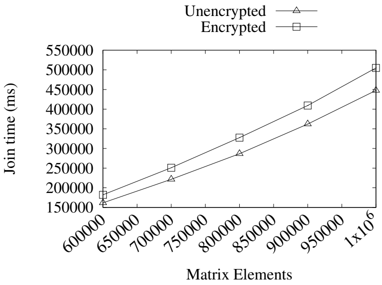
Setup

- **Processor** Intel Core i7 6700
- **Memory** 64GB
- **OS** Windows 7
- **SGX SDK Version** 1.0
- **Number of Machine** 1

Performance Impact - Matrix Size



Matrix Multiplication
(e.g. $C = A * B$)

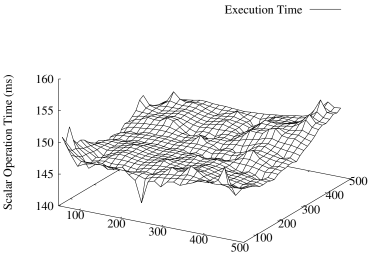


Oblivious Join

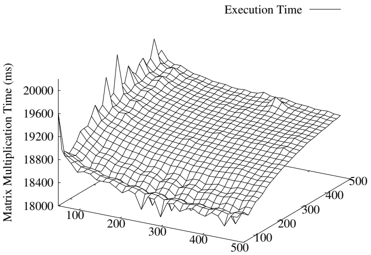
Performance Impact - Matrix Size - Summary

- We observe similar trends for all matrix operations
- We observe minimal overhead for encrypted computation
- However, the overhead depends on operation type
- More experimental evaluations in *Section 5*

Performance Impact - Block Size



Scalar Multiplication



Matrix Multiplication

Performance Impact - Block Size - Summary

- We observe execution time increases with block size
- Also, very small block size increases execution time, due to blocking overhead
- As a result, we performed optimization

Comparison with ObliVM

- We compare performance of SGX-BigMatrix with ObliVM for two-party matrix multiplication
- We observe that SGX-BigMatrix is magnitude faster because we are utilizing hardware and do not require expensive over the network communication

Matrix Dimension	ObliVM	BigMatrix SGX Enc.	BigMatrix SGX Unenc.
100	28s 660ms	10ms	10ms
250	7m 0s 90ms	93ms	88ms
500	53m 48s 910ms	706.66ms	675.66ms
750	2h 59m 40s 990ms	2s 310ms	2s 260ms
1,000	6h 34m 17s 900ms	10s 450ms	10s 330ms

Table: Two-party matrix multiplication time in ObliVM vs BigMatrix

Case Studies - Page Rank

- Performed Page Rank on three popular datasets
- Each dataset contains directed graph

Data Set	Nodes	BigMatrix Encrypted
Wiki-Vote	7,115	97s 560ms
Astro-Physics	18,772	6m 41s 200ms
Enron Email	36,692	23m 19s 700ms

Table: Page Rank on real datasets

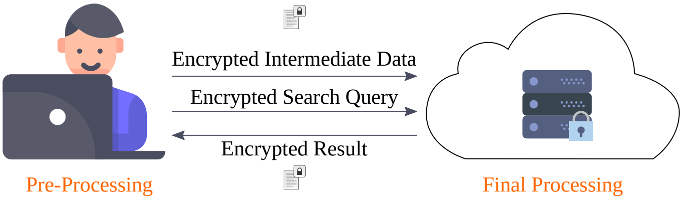
Conclusion

- We propose a practical data analytics framework with SGX
- We present BigMatrix abstraction to handle large matrices in constrained environment
- We proposed a programming abstraction for secure data analytics
- We applied our system to solve real world problems

SGX IR

Secure Information Retrieval with Trusted Processors

Problem - Secure Cloud based Information Retrieval System

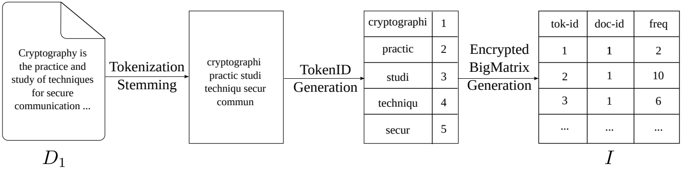


- We want to build a secure information retrieval system
- **Build** index securely **in the cloud**
- Allow secure information retrieval

Supported document and query types

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

Text pre-processing in 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

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

I

Sort $\longrightarrow$

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

I'

Count & Sum $\longrightarrow$

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
  - $c \leftarrow I'[i].token\_id \neq I'[i-1].tok\_id$
  - $\mathcal{U}[i].sum \leftarrow obliviousSelect(sum, \#, 1, c)$
  - $sum \leftarrow obliviousSelect(sum, 0, 1, c) + I[i].frequency$
- Finally, we adjust one space up to put

Oblivious Select

```
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]
```

Bucket size optimization

- We split token into smaller buckets to reduce dummy entries
- We optimize bucket size b from $count$ column of $\mathcal{U}'$
- Total buckets for i^{th} token $\lceil \frac{\mathcal{U}'[i].count}{b} \rceil$
- Elements in last bucket $\mathcal{U}'[i].count\%b$
- So, padding for i^{th} token $b - \mathcal{U}'[i].count\%b$

Padding Generation

We regenerate token id with bucket number function σ (J)

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
σ(1,0)	1	2
...	...	...
σ(1,1)	7	2
...	...	...
σ(2,0)	1	3
...	...	...
σ(2,1)	9	10
...	...	...
σ(3,0)	9	10

b

$\mathcal{U}'[1]_{count}\%b$

b

$\mathcal{U}'[2]_{count}\%b$

J

We generate padding (X)

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
σ(1,1)	#	#
...	...	...
#	#	#
...	...	...
σ(2,1)	#	#
...	...	...
#	#	#
...	...	...
σ(3,1)	#	#

$b - \mathcal{U}'[1].count\%b$

$\mathcal{U}'[1].count\%b$

$b - \mathcal{U}'[2].count\%b$

$\mathcal{U}'[2].count\%b$

X

Padding Generation - Algorithm

for i = 0 **to** $numToken$ **do**
    for j = $b-1$ **to** 0 **do**
        $c \leftarrow \mathcal{U}'[i].count\%b < j$
        $t \leftarrow \sigma(\mathcal{U}'[i].token\_id, \lfloor \frac{\mathcal{U}'[i].count}{b} \rfloor)$
        $X[i * b + j].token\_id \leftarrow obliviousSelect(t, \#, 1, c)$
    end for
end for

For each token we generate b rows, among that $b - \mathcal{U}'[i].count\%b$ rows have proper $token\_id$, remaining are totally dummy

Final token frequency table generation

- Finally we merge and sort X and J to get the $\mathcal{T}$ matrix.
- On $\mathcal{T}$ we run **term frequency** functions

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

- 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

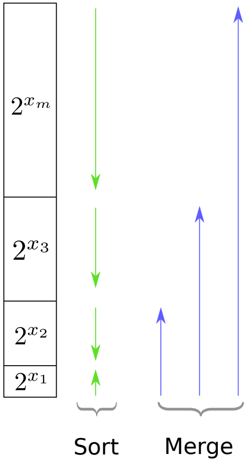
Bitonic Sorting of Arbitrary N

- Bitonic sort [Batcher, 1968] needs input to be size of 2^k
- Introduces huge overhead, when k is large
- We use **arbitrary length** version [Lang, 1998]
- However, this is recursive and SGX is memory constrained environment
- So we propose a non-recursive algorithm

Bitonic Sorting of Arbitrary N - Concept

Concept

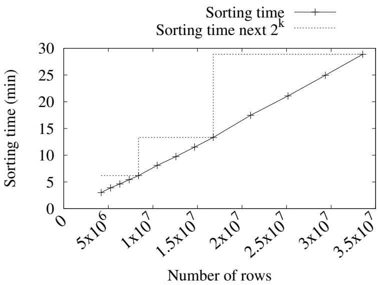
- We can express a number as $N = 2^{x_m} + ... + 2^{x_3} + 2^{x_2} + 2^{x_1}$
- Merge can sort a descending and an ascending block into ascending order
- We sort then merge from smallest to biggest block



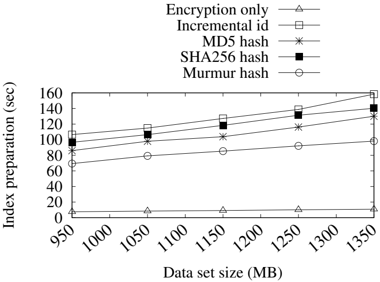
Bitonic Sorting Arbitrary N - Non-recursive Algorithm

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

Experimental Result

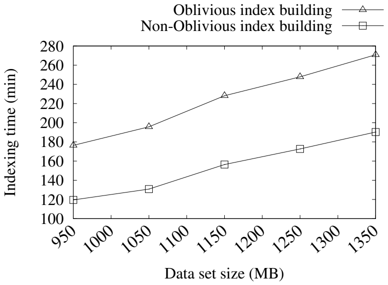


Bitonic sort

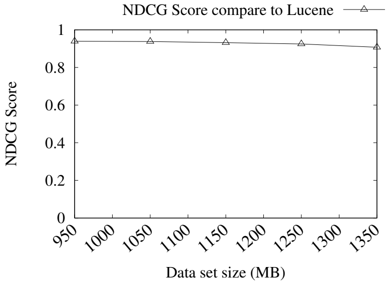


Client end processing cost

Experimental Result



SGX index processing

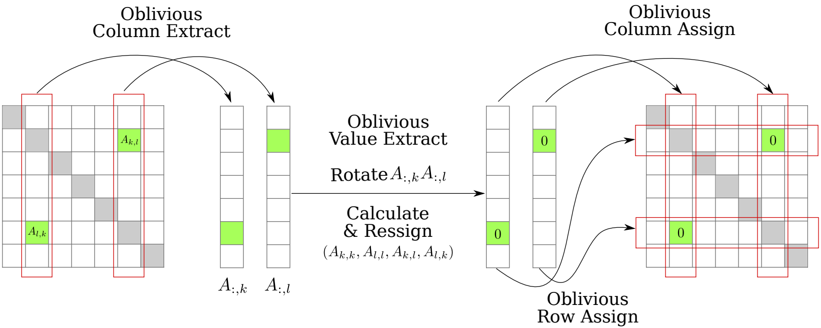


NDCG results compare to Apace Lucene

Face Recognition Indexing

- We adopt EigenFace
- Pre-processing and finding images 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 eigen values.

Oblivious Jacobi eigenvector calculation - Algorithm

$E \leftarrow identity(n)$
$\epsilon_1 \leftarrow 10^{-12}$, $\epsilon_2 \leftarrow 10^{-36}$
for it $=$ 0 **to** n^2
$max, k, l \leftarrow oMaxIndex(A)$
$\mathcal{C} = max < \epsilon_1$
$U \leftarrow oColExtract(A, k)$
$V \leftarrow oColExtract(A, l)$
$kk \leftarrow oValueExtract(U, k)$
$ll \leftarrow oValueExtract(V, l)$
$d = ll - kk$
$m = |max| < \epsilon_2 |d|$

$p \leftarrow \frac{d}{2 \times max}$
$t_1 \leftarrow \frac{max}{d}$
$t_2 \leftarrow |\frac{1}{|p| + \sqrt{p^2+1}}|$
$t \leftarrow oSelect(t_1, t_2, m, 1)$
$c = \frac{1}{\sqrt{t^2+1}}$
$s = t \times c$
$\tau = \frac{s}{1+c}$

$$\mathcal{R} = s. \begin{bmatrix} -\tau & -1 \\ 1 & -\tau \end{bmatrix}$$

$$\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$$

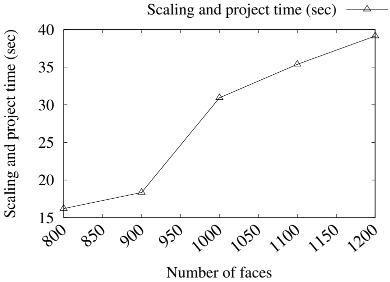
Oblivious Jacobi eigenvector calculation - Algorithm(Cont.)

$kk \leftarrow kk - t \times max$
$ll \leftarrow ll + t \times max$
$oValueAssign(U, k, kk)$
$oValueAssign(V, l, ll)$
$oValueAssign(U, l, 0)$
$oValueAssign(V, k, 0)$
$oCondColAssign(A, U, k, !\mathcal{C})$
$oCondColAssign(A, V, l, !\mathcal{C})$
$oCondRowAssign(A, U, k, !\mathcal{C})$
$oCondRowAssign(A, V, l, !\mathcal{C})$

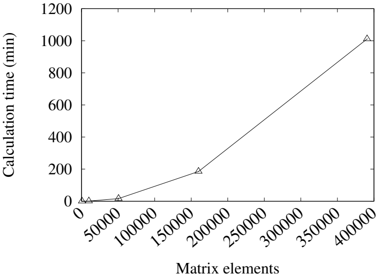
$U \leftarrow oColExtract(E, k)$
$V \leftarrow oColExtract(E, l)$
$\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$
$oCondColAssign(E, U, k, !\mathcal{C})$
$oCondColAssign(E, V, l, !\mathcal{C})$
end for

$V_i \leftarrow A_{i,i}, \forall i \in 0$ to n
$normalize(E)$
$sort(E)$ based on V

Experimental Result - Eigenvector calculation



Pre-processing overhead



Eigen calculation time

Thank You

Questions / Comments