SGX BigMatrix

A Practical Encrypted Data Analytic Framework with Trusted Processors

Fahad Shaon  Murat Kantarcioglu  Zhiqiang Lin
Latifur Khan

The University of Texas at Dallas
▶ We want to utilize cloud environment for data analytics
▶ Service provider can observe the data
▶ Problematic for sensitive data (e.g., medical, financial data)
Problem - Secure Data Analytics on Cloud

- We outsource encrypted *sensitive* data
- However, encrypted data is **difficult** to analyze
Problem - Secure Data Analytics - Approaches

Homomorphic Encryption
- Theoretically robust and 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
Objective of the work

Create a data analytics platform utilizing trusted processor, which is - secure, practical, general purpose, and scalable.
State of the Art

**ObliVM** (Liu et al., 2015)
- Provides a language and covert the logic into circuit
- Difficult to perform analysis on large data set

**Oblivious Multi-party ML** (Ohrimenko et al., 2016)
- Performs important machine learning algorithms using SGX
- Specific for set of algorithms

**Opaque** (Zheng et al., 2017)
- Oblivious and encrypted distributed analytics platform using Apache Spark and Intel SGX (mainly focused on supporting SQL)
Background - Intel SGX

- SGX stands for **Software Guard Extensions**
- SGX is new Intel instruction set
- Allows us to create secure compartment inside *processor*, called **Enclave**
- Privileged softwares, such as, OS, Hypervisor, can’t *directly* observe data and computation inside enclave
SGX essentially reduce the attack surface to processor and enclave code.

Attack surface of traditional computation system
SGX essentially reduce the attack surface to processor and enclave code.

- Attack surface of traditional computation system
- Attack surface with SGX
We only trust the processor and the code inside the enclave (Intel, 2015)
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
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**

```c
if (dir == (arr[i] > arr[j])) {
    int h = arr[i];
    arr[i] = arr[j];
    arr[j] = h;
}
```
Example: Oblivious swap method of Bitonic sort

```c
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 [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
We removed `if` and emphasis on vectorization

**Example:** Compute average income of people with \( age \geq 50 \)

\[
\text{sum} = 0, \quad \text{count} = 0 \\
\text{for } i = 0 \text{ to Person.length:} \\
\quad \text{if Person.age} \geq 50: \\
\quad \quad \text{count}++ \\
\quad \quad \text{sum} += \text{P.income} \\
\text{print } \text{sum} / \text{count}
\]
Example: Compute average income of people with $age \geq 50$

$$S = \text{where}(\text{Person}, \ "Person[\text{"age\"}] \geq 50")$$

print ($S * \text{Person[\text{"income\"}] } ) / \text{sum}(S)$
Challenge - Memory constraint

Challenge

- Current version of SGX (v1) allows only 90MB of memory allocation
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
Security Properties - Summary

- Individual operations in our system is **data oblivious**
- **Combination** of oblivious operations is also oblivious
- Compiler warns user about **potential leakage**
- We perform optimization based on publicly known information, e.g. data size
System Overview - SGX BigMatrix

SGX BigMatrix

Client

Untrusted

Compiler

Block Size Optimizer

Service Manager

Server

Trusted

Execution Engine

Block Cache

BigMatrix Library

Intel SGX SDK

OCalls

ECalls
BigMatrix Library

SGX BigMatrix - 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 *Appendix A*
Each operation has fixed **trace**

**Trace** is the information disclosed to adversary during execution

For example: operation type, input and output data size
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** \( C = A \times B \)

- 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
SGX BigMatrix - Execution Engine and Block Cache
Execution Engine

- Execute BigMatrix library operations
- Parse instruction in the form of
  \[ \text{Var ASSIGN Operation (Var, Var, ...)} \]
- 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
Execution Engine and Block Cache is also data oblivious given the input program is data oblivious

- Compiler warns about potential data leakage
- Adversary cannot infer anything more about data, apart from the trace of all the operations
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**

```python
x = load('path/to/X_Matrix')
y = load('path/to/Y_Matrix')
x_t = transpose(x)
theta = inverse(x_t * x) * x_t * y
publish(theta)
```
Output: Linear Regression

\[ x = \text{load}(X\_\text{Matrix\_ID}) \]
\[ y = \text{load}(Y\_\text{Matrix\_ID}) \]
\[ xt = \text{transpose}(x) \]
\[ t1 = \text{multiply}(xt, x) \]
\[ \text{unset}(x) \]
\[ t2 = \text{inverse}(t1) \]
\[ \text{unset}(t1) \]
\[ t3 = \text{multiply}(t2, xt) \]
\[ \text{unset}(xt) \]
\[ \text{unset}(t2) \]
\[ \theta = \text{multiply}(t3, y) \]
\[ \text{unset}(y) \]
\[ \text{unset}(t3) \]
\[ \text{publish}(\theta) \]
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**

```python
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`
We also support basic SQL

Input

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

t1 = where(person, 'C:3;V:50;0:=')
    # person.age is in column 3

t2 = zeros(person.rows, 2)
set_column(t2, 0, t3)
t3 = get_column(person, 0)
    # person.id is in column 0
set_column(t2, 1, t1)

t4 = where(person_income, 'C:1;V:100000;0:=')

t5 = zeros(person_income.rows, 2)
set_column(t5, 0, t6)
t6 = get_column(person_income, 0)
    # person_income.id is in column 0
set_column(t5, 1, t4)
A = join(t3, t5, 'c:t1.0;c:t2.0;0:=', 1)
SGX BigMatrix - Block Size Optimizer
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.
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
Execution graph (DAG) of \( \Theta = (X^T X)^{-1} X^T Y \) in linear regression training phase
\[
\text{Cost} = \text{Convert}(X, (br_X, bc_X), (x_0, x_1)) \\
+ \text{OP\_Cost}(\text{Transpose}', X, (x_0, x_1)) \\
+ \text{Convert}(X^T, (x_1, x_0), (x_2, x_3)) \\
+ \text{Convert}(X, (br_X, bc_X), (x_4, x_5)) \\
+ \text{OP\_Cost}(\text{Multiply}', [X^T, X], [(x_2, x_3), (x_4, x_5)]) \\
+ \ldots
\]

We convert this into integer programming and solve it for all the \(x_n\) variables.
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. $C = A \times B$)

Oblivious Join

FEARLESS engineering
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 Operation Time (ms)

Scalar Multiplication

Matrix Multiplication

Matrix Multiplication Time (ms)
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.

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

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

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Nodes</th>
<th>BigMatrix Encrypted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiki-Vote</td>
<td>7,115</td>
<td>97s 560ms</td>
</tr>
<tr>
<td>Astro-Physics</td>
<td>18,772</td>
<td>6m 41s 200ms</td>
</tr>
<tr>
<td>Enron Email</td>
<td>36,692</td>
<td>23m 19s 700ms</td>
</tr>
</tbody>
</table>

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
Questions / Comments

- Fahad Shaon - fahad.shaon@utdallas.edu
- Murat Kantarcioglu - muratk@utdallas.edu
- Zhiqiang Lin - zhiqiang.lin@utdallas.edu
- Latifur Khan - lkhan@utdallas.edu


