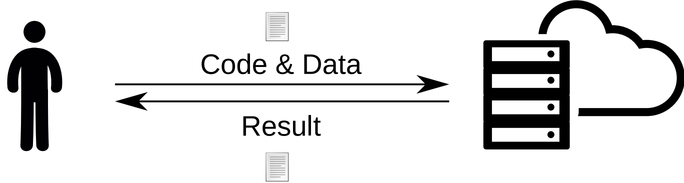
SGX BigMatrix

A Practical Encrypted Data Analytic Framework with Trusted Processors

Fahad Shaon Murat Kantarcioglu Zhiqiang Lin
Latifur Khan

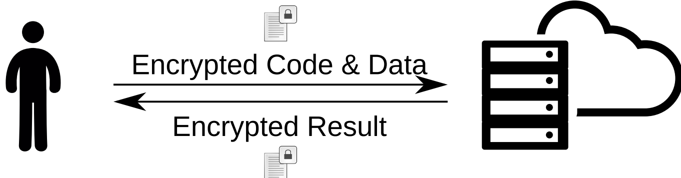
The University of Texas at Dallas

Problem - Secure Data Analytics on Cloud



- 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)

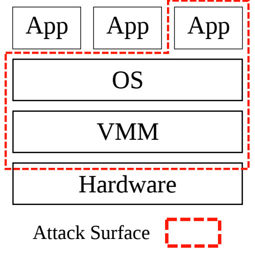
- Oblivious and encrypted distributed analytics platform using Apache Spark and Intel SGX (mainly focused on supporting SQL)

Background - Intel SGX

- SGX stands for **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't *directly* observe data and computation inside enclave

Background - Intel SGX - Attack Surface

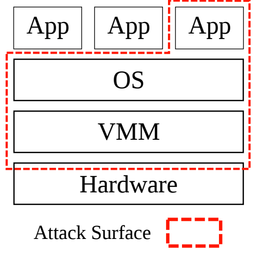
- SGX essentially reduce the attack surface to processor and enclave code



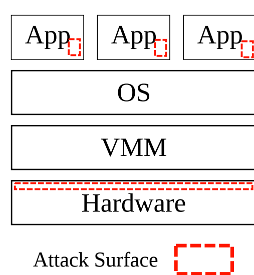
Attack surface of traditional computation system

Background - Intel SGX - Attack Surface

- SGX essentially reduce the attack surface to processor and enclave code

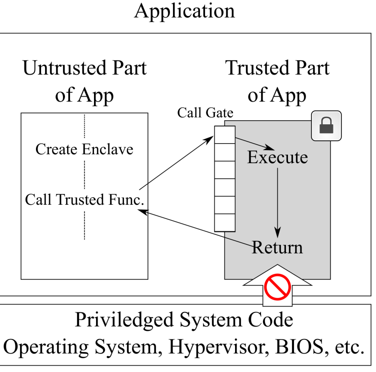


Attack surface of traditional computation system



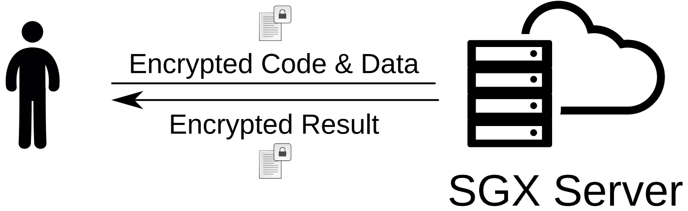
Attack surface with SGX

Background - Intel SGX Application



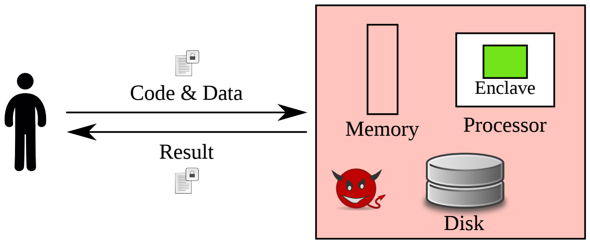
- We only trust the processor and the code inside the enclave (Intel, 2015)

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 Person.length:
    if Person.age >= 50:
        count++
        sum += P.income
print sum / count
```

Data Oblivious - Example

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

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

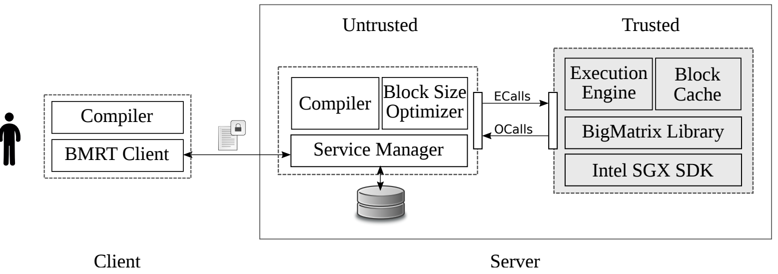
Solution

- 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**

Security Properties - Summary

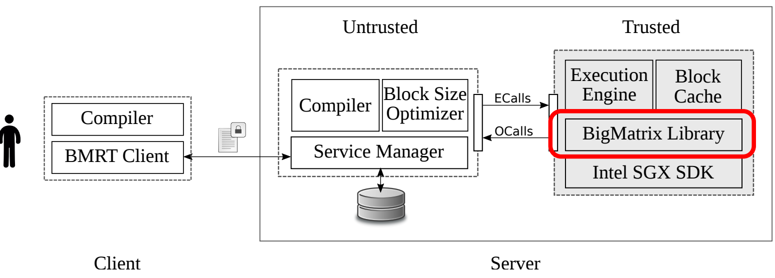
- Individual operations in our system is **data oblivious**
- **Combination** of oblivious operations is also oblivious
- Compiler warns user about **potential leakage**
- We perform optimization based on publicly known information, e.g. data size

System Overview - SGX BigMatrix



SGX BigMatrix

BigMatrix Library



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 *Appendix A*

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

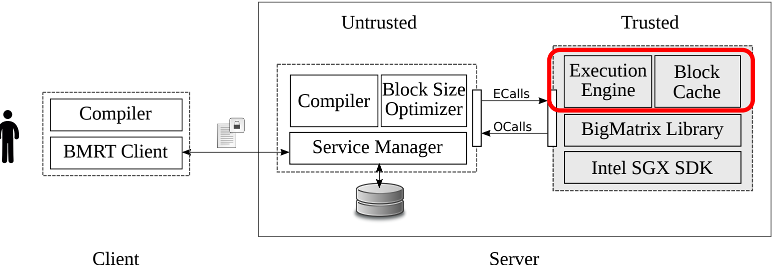
BigMatrix Library - Trace

- 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, ...)
```

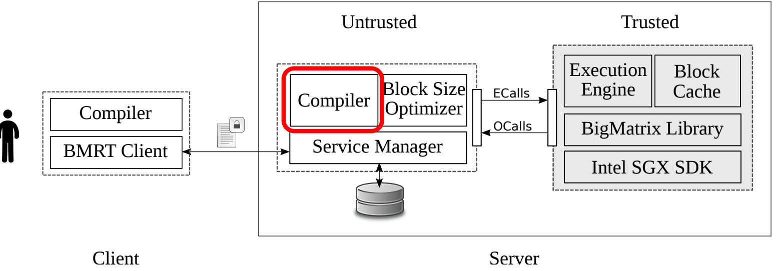
- Process sequence of instructions
- Maintain intermediate states required to execute complex program, such as, variable to BigMatrix assignments

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



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)
```

Compiler - Output

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)
publish(theta)
```

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

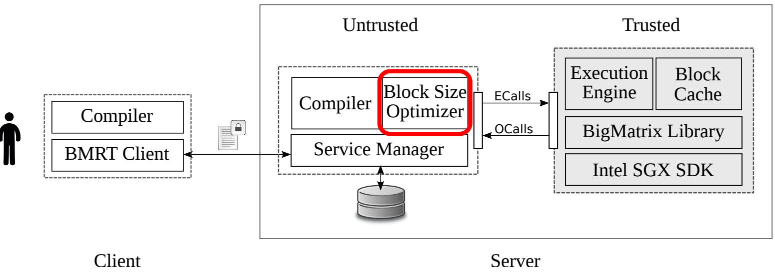
```
I = sql('SELECT *
FROM person p
JOIN person_income pi (1)
ON p.id = pi.id
WHERE p.age > 50
AND pi.income > 100000')
```

SQL Support (Cont.)

Output

```
t1 = where(person, 'C:3;V:50;O:=')
    # person.age is in column 3
t2 = zeros(person.rows, 2)
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;O:=')
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;O:=', 1)
```

Block Size Optimizer



SGX BigMatrix - Block Size Optimizer

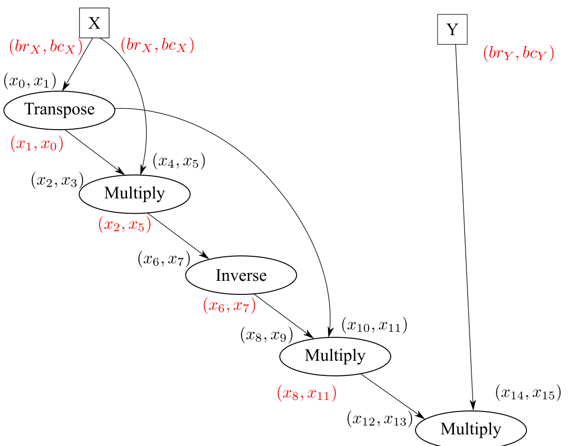
Block Size Optimizer - Intro & Design Decisions

- 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.

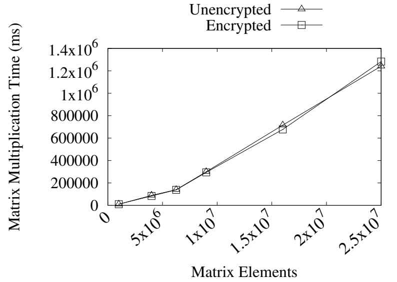
Experimental Evaluations

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

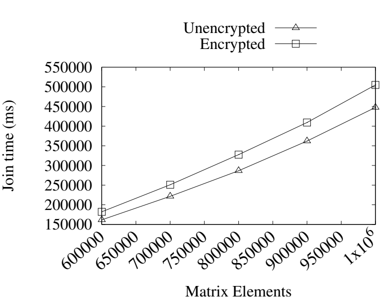
Setup

- **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. $C = A * B$)

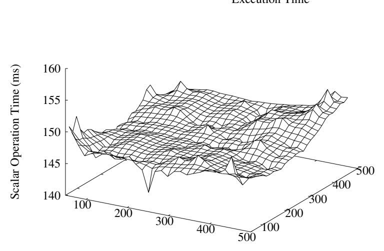


Oblivious Join

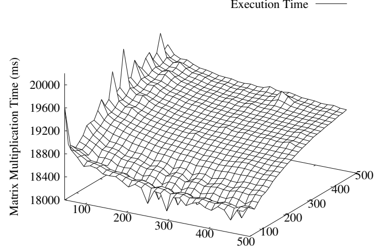
Performance Impact - Matrix Size - Summary

- 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

Thank You

Questions / Comments

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

References I

- Intel (2015). *Presentation for Intel SGX: ISCA 2015*. URL: https://software.intel.com/sites/default/files/332680-002.pdf.
- 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.
- Liu, Chang et al. (2015). "Oblivm: A programming framework for secure computation". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, pp. 359–376.
- 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.

References II

Ohrimenko, Olga et al. (2016). "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, pp. 619–636. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko.

Zheng, Wenting et al. (2017). "Opaque: A Data Analytics Platform with Strong Security". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.