SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors

Fahad Shaon
The University of Texas at Dallas
Richardson, Texas
fahad.shaon@utdallas.edu

Murat Kantarcioglu
The University of Texas at Dallas
Richardson, Texas
muratk@utdallas.edu

Zhiqiang Lin
The University of Texas at Dallas
Richardson, Texas
zhiqiang.lin@utdallas.edu

Latifur Khan
The University of Texas at Dallas
Richardson, Texas
lkhan@utdallas.edu

ABSTRACT

Recently, using secure processors for trusted computing in cloud has attracted a lot of attention. Over the past few years, efficient and secure data analytic tools (e.g., map-reduce framework, machine learning models, and SQL querying) that can be executed over encrypted data using the trusted hardware have been developed. However, these prior efforts do not provide a simple, secure and high level language based framework that is suitable for enabling generic data analytics for non-security experts who do not have concepts such as "oblivious execution". In this paper, we thus provide such a framework that allows data scientists to perform the data analytic tasks with secure processors using a Python/Matlab-like high level language. Our framework automatically compiles programs written in our language to optimal execution code by managing issues such as optimal data block sizes for I/O, vectorized computations to simplify much of the data processing, and optimal ordering of operations for certain tasks. Furthermore, many language constructs such as if-statements are removed so that a non-expert user is less likely to create a piece of code that may reveal sensitive information while allowing oblivious data processing (i.e., hiding access patterns). Using these design choices, we provide guarantees for efficient and secure data analytics. We show that our framework can be used to run the existing big data benchmark queries over encrypted data using the Intel SGX efficiently. Our empirical results indicate that our proposed framework is orders of magnitude faster than the general oblivious execution alternatives.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**;

KEYWORDS

Secure data analytics; Intel SGX

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134095

1 INTRODUCTION

Cloud computing has become an important alternative for large scale data processing due to its high scalability and low cost. One of the main challenges in cloud computing is protecting the security and privacy of the outsourced data. Recently, efficient solutions that leverage secure processors emerged as an important alternative for protecting the data stored in the cloud (e.g., [46, 50, 56]).

Secure processors allow users to execute programs securely in a manner that operating systems cannot directly observe or tamper with program execution without being detected. Previously, one had to purchase specialized hardware to build such systems. Recently, Intel has included a special module in CPU, named *Software Guard eXtension (SGX)*, into its 6th generation Core i7, i5, and Xeon processors [33] that can execute software securely, even when an operating system or a virtual machine monitor (i.e., hypervisor) is compromised. In short, SGX reduces the *trusted computing base(TCB)* to a minimal set of *trusted code* (programmed by the programmer) and the *SGX processor*, where TCB of a system is the set of all components that are critical to its security.

Still, building a robust secure application with SGX is non-trivial due to several shortcomings of the SGX architecture. In particular, operating systems can still monitor memory access patterns by the secure trusted code. It has been shown in [34, 44] that access pattern leakage can reveal a significant amount of information about encrypted data. Furthermore, SGX is a memory constrained environment. Current version of SGX can only support up to 128MB of memory for secure code execution, which includes on demand memory allocation using `malloc` or `new`. In our experiments, we observe that we can allocate at most about 90MB effectively for storing data. Therefore, we still need efficient memory management mechanisms to process large datasets.[1] Finally, the SGX architecture does not have built-in support for secure multi-user interactive computation.

In this paper, we present a generic data analytics framework in a cloud computing environment using SGX. We consider two setups: (1) *Single user scenario*, where a single end user has a large amount of data and wants to perform data analytics tasks using cloud computing infrastructure. However, the user does not trust the cloud provider with data and wants to perform operations

[1] It is worth to mention that, Intel also proposed a general dynamic memory allocation mechanism for the next version of SGX in [42]. However, to efficiently analyze very large datasets, we still need some form of memory allocation mechanisms.

on the encrypted data. (2) *Multi-user scenario*, similar to secure multi-party computation, where multiple users possess data that they want to use together to perform complex data analytics tasks. However, these users do not trust other participants with their input data, but they trust a central SGX based system due to its security guarantees and willing to share the output of the analytics operations with everyone. For example, such a setup can be used among law enforcement organizations to build threat detection models without actually sharing information other than the final result.

In our framework, we built a programming language that allows data scientists to build data analytic programs with basic operations. Our Python inspired language is designed to vectorize computations to enable simple and efficient representation of many practical data analysis tasks. Furthermore, to enable such vectorized computation, we build an efficient matrix abstraction for handling large data. To that end, we propose *BigMatrix* abstraction, which handles encrypted matrix data storage transparently and allows data scientists to access data in a block-by-block manner with integrity and privacy protection. In addition, our programming language does not allow certain constructs such as "if-statement" that may make it hard to create efficient oblivious executions. For example, a data scientist who wants to compute the average income of individuals may typically write a for-loop with if statements to compute such average (see the listing below).

```
sum = 0, count = 0
for i = 0 to Person.length:
    if Person.age >= 50:
        count++
        sum += P.income
print sum / count
```

With our framework, such a computation needs to be done using Python NumPy [6] or pandas [7] like constructs with vectorization. In the listing below, binary vector S that returns 1 for i^{th} tuple when the selection condition is satisfied ('age' > 50), which is used for computing the average income using the element-wise product operation. As we discuss later, such a vectorized computation automatically hides important sensitive information such as data access patterns.

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

By designing, efficient and oblivious matrix sorting, selection, and join operations, combined with simple for-loops, we show that all most all of the practical data analytics tasks can be programmed and executed in our framework. Furthermore, during our experimental evaluation, we observed that block sizes and the order of certain operations (e.g., SQL like operations) has an impact on execution time. As such, we proposed an optimization mechanism with the programming abstraction, that will find the optimum execution policies for a given sequence of basic operations. In addition, to utilize our proposed data analytics framework, we have provided specific protocols to load code and data, provision and execute program, and distribute the result. Furthermore, we emphasize on building data oblivious system, where code execution does not depend on data. Instead of using generic complex Oblivious RAM (ORAM) algorithm (e.g., [53]) to hide data access patterns, we leverage our knowledge of the vectorized computation algorithms to *provide operation specific but very efficient oblivious algorithms*. We have made all of our individual operations to be data oblivious and provided a theoretical proof that combination of such operations remains oblivious. As a result, an adversary cannot learn extra information based on data access alone.

Contributions. The main contributions of this paper can be summarized as follows:

- We propose a generic framework for secure data analytics in an untrusted cloud setup with both single user and multi-user settings. Compared to existing work that leverages trusted processors (e.g., relational database system [13, 15], mapreduce [50], sql execution on spark [56], etc.), to our knowledge, we are among the first to provide a high level python inspired language that allows efficient, generic, and oblivious execution of data analytics tasks.
- We present *BigMatrix*, an abstraction for handling large matrix operations in a data oblivious manner to support vectorization (i.e., represent various data processing operations as matrix operations).
- We also provide a programming abstraction that can be used to execute a sequence of commands obliviously with optimum cost. We also theoretically prove that combinations of oblivious methods remains oblivious.
- We have implemented a prototype showing the efficiency of our proposed framework compared to existing alternatives.

2 BACKGROUND

In this section we provide necessary background on Intel SGX and Data Obliviousness, in order to understand the motivation and design of our framework.

2.1 Intel SGX

Intel SGX is a new CPU extension for executing secure code in Intel processors [12]. In SGX computation model, programmers need to partition the code into trusted and untrusted components. The trusted code is encrypted and integrity protected, but the untrusted code is observable by the operating system. During the program execution, the untrusted component creates a secure component inside the processor called *enclave* and loads trusted code into it. After creating the enclave, users can verify that intended code is loaded and securely provision the code with secret keys, which is called *attestation*. Internally, the infrastructure uses Enhanced Privacy ID (EPID) [22] for hardware based attestation. In addition, trusted and untrusted components communicate between each other using programmer defined entry points. Entry points defined in trusted code is called *ECalls*, which can be called by untrusted part once enclave is loaded. Similarly, entry points defined in untrusted code is called *OCalls*, which can be called by the trusted part. More details about the SGX execution model are described in [24, 48].

2.2 Data Oblivious Execution

A program is called data oblivious if for all data inputs the program executes exactly the same code path. The main benefit of data obliviousness is that any powerful adversary that is capable of observing code execution, does not learn anything extra about the data based on the code execution path. To explain data obliviousness, we also have to clearly define the capabilities of an adversary in our design. We assume that an adversary in an SGX environment can observe memory accesses, time to execute, OCalls, and any resource usages from OCalls. However, an adversary in SGX cannot observe internal CPU registers.

We define a program is data obvious in the SGX environment if the same memory regions are accessed for all possible input datasets. For example, data arithmetic operations, such as `add`, `mult`, etc., are by definition data oblivious because the instruction performs the same task irrespective of any input data. However, conditional instructions, such as, `jne`, `jeq`[2] are *not* data oblivious because these instruction force different part of the code to be executed based on input data.

To implement programs that require such conditional operations, we first assign values from different possible code paths, to different registers, then set a flag based on the condition that we want to test, swap according to the flag, and finally return the contents of a fixed register. Such techniques are used in previous works (e.g., [46, 49]). Data oblivious approach of programming protects against attacks from access pattern leakage as described in [34, 44]. Because these attacks are based on the frequency of data access for different input and data obliviousness guarantees that data access frequency should be the same irrespective of same sized input data.

3 SECURE DATA ANALYTICS FRAMEWORK

Processing a large amount of data with Intel SGX is particularly difficult because of the limited memory of a given enclave. In current SGX processor we can allocate at most about 90MB of dynamic memory inside an enclave. In addition, as discussed in subsection 2.2, data access patterns during encrypted data processing could also leak significant information.

Furthermore, from our own experience, we observe that Intel SGX development life cycle is somewhat time consuming. We first need to divide the whole program into two components - trusted and untrusted parts with defined entry points. Next, we have to carefully implement the required algorithms in trusted part in C/C++. Finally we have to deploy into a SGX server, verify the loaded code, provision secret, and finally run the code. However, in modern data analytics, we observe that data scientists tend to prefer interactive tools. In fact, popular analytical platforms (such as R [8], Octave [4], Matlab [5], Apache Spark [1], etc.) offer REPL (Read-Eval-Print Loop) environments where users can perform operations on data, get instant feedback, and repeat the whole process. In a recent survey [35] on data science practitioners, top 3 preferred programming languages for data scientists are, R, Python, and SQL. Furthermore, only 9% of the data scientists in the survey use C/C++ for data analysis. One major reason behind this might be, easy data exploration and visualization is often more important than writing the most optimized solution.

[2]jne, jeq are assembly instructions for jumping based on zero flag.

We also observe that complex data analytics tasks can be expressed as basic matrix operations if the data is represented as a matrix. In fact, entire language and analytical stacks, such as, Matlab [5], Octave [4], NumPy [6], and Pandas [7], has been proposed around matrix operations. Moreover, basic matrix operations, such as, multiplication and transpose are by definition data oblivious.

In light of these observations, we propose an efficient and interactive framework to handle large encrypted datasets for generic data analytics tasks by leveraging the Intel SGX instruction set. Our main objective is to bring matrix based computation into secure processing environment in a way that would allows us to perform any matrix operation on large encrypted matrices. So we propose *BigMatrix Runtime*, and at the core we have *BigMatrix* abstraction that split a large matrix into a sequence of smaller ones and performs individual matrix operations using smaller block. BigMatrix handles the blocking and encryption of the small block automatically and transparently. In addition, we add other key operations, such as, sorting and selection, on top of BigMatrix abstraction to support most data analytics computations.

3.1 Setup, Protocols, and Threat Model

In our framework, we consider a setup where a single participant or multiple participants are connected to an Intel SGX enabled server. The server is assumed to be controlled by an adversary, who can observe the main memory (RAM) content and main memory processor communications. Furthermore, the adversary can delete/-modify the stored data, provide wrong data, and stop the execution of the enclave. At the same time, due the capabilities of the Intel SGX, we assume that the attacker cannot modify the code that is running in the enclave.

Participants do not trust each other with their data but they want to execute a program that will perform some data analytics task over all the participants' data. Also, we assume that the participants are not sending invalid datasets or aborting the process abruptly. In addition, each user has the capability to verify that the server has loaded the proper code. If the server does not load the proper code, participants will be able to detect the deviation. All the communications between the server and the participants are done over a secure communication channel, such as, Https. We also assume that the owner of the server is not colluding with the participants.

In our framework, given the attacker capabilities, our goal is to detect any tampering by the attacker and limit information leakage during the data analytic task execution process. Furthermore, we want to make the framework suitable for multi-user setting where different parties can combine their data and build collaborative model. To achieve these goals, our proposed secure data analytics framework had three distinct phases: 1) Code agreement and loading phase that allows multiple parties to agree on the common task that will run on their joint data, 2) Input data and encryption key provisioning phase that allows data encryption and key sharing, 3) Result distribution phase that provides the computation result to multiple users. We discuss these phases in detail in Appendix C.

3.2 Overview

BigMatrix Runtime has two major components: 1) *BigMatrix Runtime Client*, where a user provides input data and tasks to perform

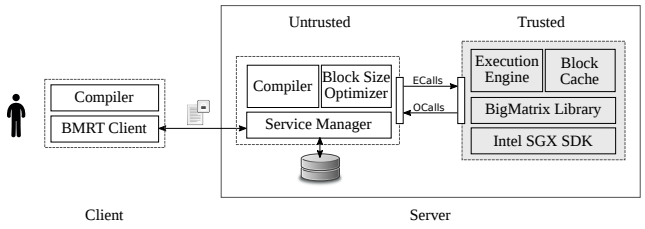


Figure 1: Framework Overview.

on the input data, 2) *BigMatrix Runtime Server*, which interprets user's commands and performs the requested tasks. Before going into the details of each component, we first provide a top-level overview of code execution in BigMatrix Runtime.

A user first makes sure that the server is started properly with secure enclave code and provision the enclave with proper secret keys using proposed protocols. Next, the user provides input program and data to the BigMatrix Runtime client, which uses a *compiler* to compile the program into *execution engine* compatible code and perform error checking. The client also encrypts the data using the proper key. Next, the client sends the code and encrypted data to *Service Manager*. Service Manager next performs block size optimization and loads encrypted data in the enclave with optimum block size information. Then service manager starts the execution engine that performs the user specified operations. Once the operation execution has been finished Service Manager sends enclave generated data back to the client, which later displays the result back to the user.

BigMatrix Runtime client consists of two components: a) Client and b) Compiler. BigMatrix Runtime server consists of six logical components: a) BigMatrix Library, b) Execution Engine, c) Compiler, d) Block Cache, e) Block Size Optimizer, and f) Service Manager. In the rest of this section, we explain each of these components.

3.3 Key Operations of BigMatrix Library

At the very bottom layer, we have BigMatrix library, which contains sets of operations on our proposed BigMatrix abstraction. A BigMatrix is essentially a matrix of smaller matrices. Basically, we compute a specific block size that we can fit into SGX enclave and split a large input matrix into smaller blocks and perform operations using these blocks. This abstraction is needed since SGX is a memory constrained environment.

We have defined few basic functions in BigMatrix library, which we later use to build more complex operations. Our defined functions falls into the following five categories:

1. Data access operations. a) `load(participant_id, mat_id)`: load matrix with `mat_id` from the storage, which is encrypted with session key of participant, `participant_id`. b) `publish(A)`: publish the matrix `A` for all the participants. c) Partial access operations: `get_row(A, i)`, `set_row(A, i, r)`, `get_column(A, j)`, `set_column(A, j, c)`, `get_element(A, i, j)`, `set_element(A, i, j, v)`.

We defer the discussion of how we serialize, encrypt, store, and load the BigMatrix in subsection 3.8, once we define other relevant components of the system.

2. Matrix Operations. a) `scalar(op, A, value)`: perform scalar operation `op` on each element of input matrix `A` and return the output, where `op` is a binary operation such as addition, multiplication, and, or, etc., `A` is a BigMatrix and `value` is numeric value. b) `element_wise(op, A, B)`: perform element wise operation `op` on two big matrices and return the result. c) `multiply(A, B)`: perform matrix multiplication of big matrices `A` and `B`. d) `inverse(A)`: perform inverse of big matrix `A`. e) `transpose(A)`: create the transpose of big matrix `A`.

3. Relational Algebra Operations. a) `where(A, condition)`: perform basic selection operation on `A` for a given `condition` and return a `0-1` column matrix. b) `sort(A, columns, direction)`: sort the rows of matrix `A` using bitonic sort. c) `join(A, B, condition, k)`: perform SQL like join of `A` and `B` based on `condition`. `k` is a parameter to ensure obliviousness, which we discuss in Appendix A.9. d) `aggregation(A, commands, columns)`: perform basic aggregation on `A` on `columns`. Allowed aggregation commands are `sum`, `average`, `count`, `min` and `max`. We also implemented `argmax(A, columns)` and `argmin`, which provide the index of highest and lowest value row in the matrix.

4. Data generation operations. a) `rand(m, n)`, `zeros(m, n)`, `ones(m, n)`: generate a BigMatrix of size $m \times n$ containing uniform random numbers, zeros and ones respectively. b) `eye(n)`: generate an identity matrix of $n \times n$.

5. Statistical Operations. a) `norm(A, p)`: compute p-norm of the vector ($n \times 1$ matrix) `A`. b) `var(A)`: compute variance of the vector `A`.

All the operations in our BigMatrix library also have pre-defined *trace*, which is the amount of information leakage due to performing the operation. For example, the `multiply` operation leaks the information about the size of the matrix `A` and `B`. We refer readers to Appendix A for more internal details including *trace* and cost of important BigMatrix operations.

3.4 Compiler and Execution Engine — Programming Abstraction

As stated earlier, quick and secure data analytical development cycle is a major target of the proposed framework. To that end, we define a compiler and execution engine that can process and execute code, which is written in a python-like language. The execution engine is part of our trusted environment and can interpret assembly-like instructions, such as, `C = multiply(A, B)`. On the other hand, the compiler resides outside the enclave and creates execution engine compatible code from our custom language, which is inspired by languages such as python and octave. The main reason behind such a split architecture is to reduce the size of TCB (trusted computing base). There is no regular expression or context free grammar functionality in SGX library. So if we want to support interactive computation in any language we would need to bring in the complete grammar processing library into the TCB, which increases the risk of introducing potential vulnerability through bugs of these libraries. On the other hand, we could build a parser that outputs code into `X86` assembly architecture and put more simplified execution engine. However, we avoided this option because traditional assembly instruction set has complex branching instructions that

are very hard to convert to the equivalent data oblivious version. Furthermore, the instruction set is highly restricted to a fixed set of registers, which is not the case for our execution engine.

Our compiler is divided into five components: Lexical analyzer, syntax analyzer, semantic analyzer, optimizer, and code generator. Lexical analyzer takes the input file and outputs a stream of tokens. Syntax analyzer takes the token streams and creates a syntax tree representing the input source code. During syntax tree creation syntax analyzer also lists any syntax errors. Semantic analyzer analyzes the syntax tree and checks for semantic mistakes. One of the semantic tests that we perform is matrix conformability [32], where we test operand matrices whether they have proper dimensions for intended operations. In this stage, we also *perform a sensitive data leakage analysis*, where we check if any sensitive information is leaking as a side effect of some operations. For a given program, we define the non-sensitive information as: (a) input size, and (b) constants in the input program. On the other hand, we also know the *trace* (set of values per operation that is disclosed) of all the operations in the input program. Semantic analyzer checks for items in the *trace* that is not non-sensitive and warns users of possible information leakage. For example, our semantic analyzer will raise error for the following input code.

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

Because, here the value of `s` is in the *trace* of function `zeros` but the value is not in the non-sensitive data list. Next, optimizer performs few compile time optimizations, such as basic query optimization (detail in subsection 3.7), and matrix multiplication order optimization. Finally, code generator takes the syntax tree and generates execution engine compatible code. In addition, our compiler also outputs complete *trace* of a input program so that programmers can easily understand information leakage.

The execution engine can run in two modes: interactive and non-interactive. In the interactive mode, a user loads the enclave, verifies, starts a session, provides sequence of instructions, and closes the session at will. So the system does not know all the instructions to be executed. In this mode, the values of variables are retained until user explicitly `unset` it. In non-interactive mode, the user provides completed tasks to be executed and the compiler generates necessary `unset` commands depending on the last used instructions.

Our framework supports variables of types `int32`, `int64`, `double`, BigMatrix of different types, and fixed length strings. The language is not strictly typed, i.e., during initialization a user does not have to specify the type of a variable. Our system can handle *fixed length loops* and we are assuming that the number of loop iterations can be leaked to the adversary (e.g., constant or some known value, such as rows, columns, block_size). In addition, we also protect intermediate data tampering. We keep an internal table of matrix id and header MAC (message authentication code) of matrices in a computation. So, if an operating system sends invalid data (i.e., an active attack, or unintentional data corruption), our execution engine will be able to detect it. We discuss our MAC generation in subsection 3.8.

An Example. Now we provide an example on how our framework could be used to execute fundamental data analytics tasks. **Linear Regression** is an approach for modeling the relationship between a scalar dependent variable y and one or more independent variables [37]. Let, m be the number of inputs, X be the training dataset, Y be the output of training dataset, $X^{(j)}$ and $Y^{(j)}$ be the j^{th} training set and class respectively, Θ be the regression parameters, and $\hat{y}$ be the predicted class of test input x, then

$$\hat{y} = \Theta^T x$$

where, $\Theta = (X^T X)^{-1} X^T y$. In our programming language, we can compute the Θ using the following code snippet.

```
x = load(0,  path/to/X_Matrix )
y = load(0,  path/to/Y_Matrix )
xt = transpose(x)
theta = inverse(xt * x) * xt * y
publish(theta)
```

Our compiler will convert the above code snippet into the following sequence of instructions that can be executed by our execution engine.

```
x = load(0, X_Matrix_ID)
y = load(0, 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)
```

Again if the code ran in the interactive mode, our compiler would not generate the `unset` instructions. In this case, the leaked information to adversary is the size of `x` and `y` matrices and sequence of operations. [3]

We also defined PageRank, Naive Bayes, and K-Means clustering algorithm in our programming language. We refer readers to Appendix B for these examples.

3.5 Block Cache

Next we briefly describe a cache layer which caches the loaded blocks and dynamically replaces existing big matrix blocks from cache. In addition, we can also minimize the total cache misses. In the non-interactive mode, i.e., where a user provides the entire work load, we replace the cache using furthest in future policy [20]. It is particularly possible in our case since the work load is known and most importantly the code is data oblivious meaning data access

[3] We discuss the security guarantees of our framework in more detail in section 4.

does not depend on input dataset content rather only on the size. The furthest in future is known as an optimal policy, where we replace the cache element that will be required furthest in the future. On the other hand, in the interactive mode, we replace in least frequently used model.

3.6 Block Size Optimization

In our experiments, we observed that the cost of each operation varies depending on the block size. So, we propose an optimization mechanism that reduces the total cost of a sequence of operations. We formalize this optimization by assuming that the input program can be represented as a directed acyclic graph (DAG) of operations.

Let, $O = \{o_1, o_2, ..., o_n\}$ be the set of operations, $\mathcal{M} = \{M_1, M_2, ...\}$ be all big matrices in the computation that are divided into blocks, $B \in \mathbb{R}^d$ be the block dimensions, where d is the number of dimensions (for simplicity we are considering $d = 2$), $\mathcal{B} = \{B_1, B_2, ...\}$ be the sets of the block dimensions of BigMatrix set $\mathcal{M}$, $\Pi(o_i, \mathcal{M}_i, \mathcal{B}_i)$ is the processing cost of o_i on $\mathcal{M}_i$ that is blocked as $\mathcal{B}_i$ size blocks, $\Delta(M, B_i, B_j)$ is the cost of converting the block size of BigMatrix M from B_i to B_j, $\lambda(o_i, \mathcal{B}, B)$ is the peak memory required to perform operation o_i with input BigMatrix blocked in $\mathcal{B}$ and output BigMatrix blocked in B.

Next, we define functions that will help us define the cost function. Let, $\mathcal{P} = \{\rho_1, \rho_2, ..., \rho_m\}$ be a program defined as DAG of operation, $\texttt{Op}(\rho_i) \in O$ operation of node ρ_i, $\texttt{InNodes}(\rho_i) \subseteq \mathcal{P}$ is the node that is input of ρ_i, $\texttt{InBlks}(\rho_i)$ is the sets of input blocks dimension of node ρ_i, $\texttt{InBlks}(\rho_i)[j]$ is the j^{th} input block dimension of node ρ_i, $\texttt{OutBlk}(\rho_i)$ is the output block dimension of node ρ_i, $\texttt{InBigM}(\rho_i)$ is the set of input BigMatrix of node ρ_i, $\texttt{OutBigM}(\rho_i)$ is the output BigMatrix of node ρ_i.

Therefore, the cost of operation of node ρ_i can be defined in the following:

$$cost(\rho_i) = \Pi(\texttt{Op}(\rho_i), \texttt{InBigM}(\rho_i), \texttt{InBlks}(\rho_i)) + \sum_{\rho_j \in \texttt{InNodes}(\rho_i)} \left[\Delta(\texttt{OutBigM}(\rho_j), \texttt{OutBlk}(\rho_j), \texttt{InBlks}(\rho_i)[j])\right]$$

Finally, we can define the minimization function as

$$\sum_{\rho_i \in \mathcal{P}} \texttt{cost}(\rho_i)$$

subject to: $\lambda(\texttt{Op}(\rho_i), \texttt{InBigM}(\rho_i), \texttt{OutBigM}(\rho_i)) < \texttt{MaxMem}$ (memory limit) and $\texttt{InBigM}(\rho_i)$ is conformable (i.e., the dimensions are suitable for the operation.) The above formalized optimization can easily be converted into an integer programming.

A block optimization example for linear regression. Next, we show how to apply our optimization technique to minimize the cost for executing linear regression training phase, i.e. θ computation, as shown earlier. The corresponding execution tree is illustrated in Figure 2. Here, X and Y are two input matrices formatted into BigMatrix format with block size of (br_X, bc_X) and (br_Y, bc_Y). Again, for simplicity we are considering 2-dimensional matrices. The first operation in our framework is `Transpose` that takes input of BigMatrix X and outputs BigMatrix X^T. Let us assume that for this operation the input matrix was blocked into (x_0, x_1), so the output is blocked into (x_1, x_0) block. (In reality $x_0 = br_X$ and $x_1 = bc_X$.)

Next, the operation in this program is `Multiply` that performs matrix multiplication over BigMatrix X^T and X, which are blocked into (x_2, x_3) and (x_4, x_5). The output will be blocked into x_2, x_5. So on and so forth. Now we can compute the over all cost in term of variables x as follows

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

Our target here is to assign values to these x variables in such a way that it satisfies the computation requirements and also reduce the over all cost. From our experiments, we know the values of Π and Δ for different combinations of block size. As observed in our experimental evaluation, the cost is quite easy to approximate with a very low error rate. Finally, it is worth mentioning that, we perform the optimization outside the enclave using the information already leaked in the trace of the operations (e.g., the size of data matrix). Therefore, the optimization will not leak any further information.

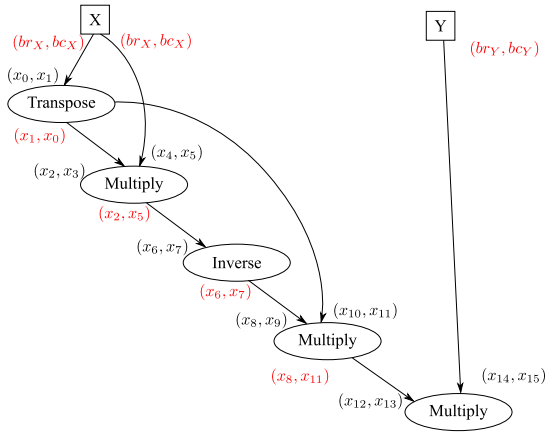


Figure 2: Linear regression execution tree for block size optimization.

3.7 SQL Parsing and Optimization

Our basic instruction set contains a subset of the SQL operations. To make the programming easier, we provide a SQL parser that takes a SQL `SELECT` query as input and create an optimized sequence of instruction to execute the query using our basic commands. For instance, a SQL query `A = sql("SELECT * FROM person WHERE age > 50")` would be compiled into `A = where(person, 'C:3;V:50;O:=')`, (assuming that, 'age' is in the third column). Here, the condition is encoded in postfix notation [31]. More specifically, the `C:3` part of the expression means the third column, `v:50` means value 50 and `O:=` means operation equal. We choose postfix notation because it is easy to evaluate. The compiler can also parse join queries such as:

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

which will be converted as follows

```
...
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)
...
```

Our compiler also takes into consideration of SQL optimizations. In our implementation, we applied a few standard heuristics such as pushing selection operations [27]. In our future work, we are considering utilizing optimization engine from popular open-source databases. However, we also observed that most of these optimizations heavily depend on existing index and data stored in the database (e.g., predicate sensitivity). In contrast, our datasets are encrypted and we protect against data access patterns so index utilization is not an option for us. Furthermore, optimizations that depend on data distribution are not applicable due to the sensitive information disclosure issues. It is worth mentioning that, we only support subset of standard SQL in our current implementation and our join query requires an additional parameter k that is the maximum number of row matches from the first table to the second.

3.8 BigMatrix Storage

Next, we briefly discuss how we serialize, encrypt, load and store the big matrices.

Serialization and Encryption. One important aspect of our framework is that it provides transparent security for large datasets. First, we compute the number of blocks we need to keep in memory to perform the intended operations. Next, we compute the total number of elements that we can keep in memory. Based on these two values, we partition our matrix into smaller blocks. Also, it is possible to have edge blocks in a BigMatrix, which does not have the same number of elements compared to the rest of the blocks. We serialize each individual block matrix and encrypt the block with authenticated encryption AES-GCM [26], and store MAC of all blocks into their header. We also store the total number rows and the total number of columns into the header. Essentially with information from header we can find out the necessary details of a given block and ensure the authenticity and integrity of the individual blocks. Finally, we serialize and encrypt the header. Figure 3 illustrates our serialization process.

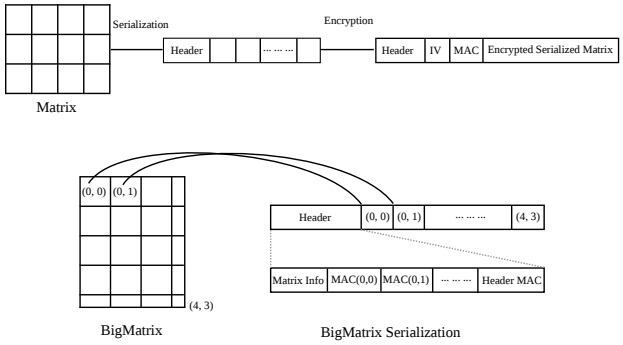


Figure 3: Serialization of a Sample BigMatrix.

Storing and Loading mechanism. As we explained in section 2, we create secure enclaves using Intel SGX API. To write a BigMatrix from enclave to disk we designed `init_big_matrix_store`, `store_block`, and `store_header` *OCall* functions. The first function initialized an empty file for a BigMatrix and assign a randomly generated matrix id to the BigMatrix. The second function stores a block of a particular BigMatrix. The third one stores the header of the BigMatrix. We need to call the store header function after writing of all the blocks because our header contains MAC of all the individual blocks to provide the integrity protection. Similarly, we defined `load_header` and `load_block` *OCall* functions to load header and blocks of an existing BigMatrix, respectively.

During code execution in the execution engine, we also keep an internal table of id and header MAC. Every time we store a BigMatrix using `store_header` function, we store the header MAC and matrix id. Every time we load a BigMatrix using `load_header` function, we check the header MAC and stop execution in case of MAC mismatch.

3.9 Writing Customized Operations

In addition to our own basic operations, an expert programmer can provide customized code to be executed as operations. We designed our code in such a way that the user just needs to provide us an implementation of a predefined abstract class and add the class name in a configuration file. During the build process our build script will look into the configuration file, generate call table for execution engine. Our internal operations are also implemented using the same mechanism. However, building customized method requires code building and can easily introduce unintentional vulnerabilities. Furthermore, the programmers need to guarantee data obliviousness of the implementation. In our current implementation, compiler considers the input sizes as trace of the implementation. In addition, the current version of our language does not support functions yet. We are planning to add the function support in future version.

4 SECURITY ANALYSIS

In this section, we give an overview of the oblivious execution guarantees provided by our system. As we discuss in subsection 3.4, our framework is designed to detect any modification to the underlying data and program execution. Furthermore, we assume that due to SGX capabilities, a malicious attacker cannot observe the register contents. So an attacker can only observe the memory and disk access patterns. Below, we formally define what is leaked during the program execution for an adversary that can observe only memory and disk access patterns. Protection against other type of side channel attacks such as timing, energy consumption is outside the scope of this work.

4.1 Composition Security

Let, $D = \{D_1, .., D_\alpha\}$ be the input data, $\mathcal{I} = \{\mathcal{I}_1, ...\mathcal{I}_\alpha\}$ be the encrypted input data, $R = \{R_1, ..., R_\beta\}$ be the intermediate output set, $\mathcal{R} = \{\mathcal{R}_1, ...\mathcal{R}_\beta\}$ be the encrypted intermediate output set, O be the output, $\mathcal{O}$ be the encrypted output, $\mathcal{F} = \{\mathcal{F}_1, ..., \mathcal{F}_f\}$ be the set of available oblivious functions, where each function $\mathcal{F}_i$ takes the predefined number of inputs from $\mathcal{I} \cup \mathcal{R}$ and outputs the predefined number of outputs from $\mathcal{R} \cup \{\mathcal{O}\}$ set. $C_\eta = \{\mathcal{F}_1, ..., \mathcal{F}_\eta\}$ be what the code participants agreed on. Here, C_η is a combination of η functions from $\mathcal{F}$.

- **Input Access Pattern ($\mathcal{A}_p$):** Suppose $\mathcal{F}_i$ is the i^{th} function executed in C_η and during the execution $\mathcal{F}_i$ accessed $\{\mathcal{I}_1, ..., \mathcal{I}_z\}$, i.e., $\mathcal{F}_i$ depends on $\{\mathcal{I}_1, ..., \mathcal{I}_z\}$, then, $\mathcal{A}_{p_i} = \{1, ..., z\}$. Finally, $\mathcal{A}_p(\mathcal{H}_\eta)$ is defined as the sequence of all the $\mathcal{A}_{p_i}$. The input access pattern captures the access sequence of input data during the secure code execution.
- **Intermediate Access Pattern ($\mathcal{B}_p$):** Suppose $\mathcal{F}_i$ is the i^{th} function executed in C_η and during the execution $\mathcal{F}_i$ accessed $\{\mathcal{R}_1, ..., \mathcal{R}_z\}$, i.e., $\mathcal{F}_i$ depends on $\{\mathcal{R}_1, ..., \mathcal{R}_z\}$, then, $\mathcal{B}_{p_i} = \{1, ..., z\}$. Finally, $\mathcal{B}_p(\mathcal{H}_\eta)$ is defined as the sequence of all the $\mathcal{B}_{p_i}$. The intermediate access pattern captures the access sequence of intermediate data during the secure code execution.
- **Intermediate Update Pattern($\mathcal{U}_p$):** Suppose $\mathcal{F}_i$ is the i^{th} function executed in C_η and during the execution $\mathcal{F}_i$ modifies $\{\mathcal{R}_1, ..., \mathcal{R}_z\}$, e.g., $\mathcal{F}_i$ outputs on $\{\mathcal{R}_1, ..., \mathcal{R}_z\}$, then, $\mathcal{U}_{p_i} = \{1, ..., z\}$. Finally, $\mathcal{U}_p(\mathcal{H}_\eta)$ is defined as the sequence of all the $\mathcal{U}_{p_i}$. The intermediate update pattern captures the update of intermediate data during the secure code execution.
- **History($\mathcal{H}_\eta$):** The history of the system is $\mathcal{H}_\eta = (D, R, O, C_\eta)$.
- **Trace (λ):** Let $|\mathcal{I}_i|$ be the size of encrypted input $\mathcal{I}_i$, $|\mathcal{R}_i|$ be the size of intermediate output $\mathcal{R}_i$, and $|\mathcal{O}|$ be the size of the output. Then, trace $\lambda(\mathcal{H}_\eta) = \{(|\mathcal{I}_1|.., |\mathcal{I}_\alpha|), (|\mathcal{R}_1|.., |\mathcal{R}_\beta|), |\mathcal{O}|, \mathcal{A}_p(\mathcal{H}_\eta), \mathcal{B}_p(\mathcal{H}_\eta), \mathcal{U}_p(\mathcal{H}_\eta)\}$. Trace can be considered as the maximum amount of information that a data owner allows its leakage to an adversary.
- **View (v):** The view of an adversary observing the system is $v(\mathcal{H}_\eta) = \{\mathcal{I}, \mathcal{R}, \mathcal{O}\}$. View is the information that is accessible to an adversary.

Now, there exists a probabilistic polynomial time simulator $\mathcal{S}$ that can simulate the adversary's view of the history from the trace.

Theorem 4.1. *The proposed function composition does not reveal anything other than the view v.*

Proof. We show there exists a polynomial size simulator $\mathcal{S}$ such that the simulated view $v_S(\mathcal{H}_\eta)$ and the real view $v_R(\mathcal{H}_\eta)$ of history $\mathcal{H}_\eta$ are computationally indistinguishable. Let $v_R(\mathcal{H}_\eta) = \{\mathcal{I}, \mathcal{R}, \mathcal{O}\}$ be the real view. Then $\mathcal{S}$ adaptively generates the simulated view $v_S = \{\mathcal{I}^*, \mathcal{R}^*, \mathcal{O}^*\}$

$\mathcal{S}$ first generates α number of random data of size $\{|\mathcal{I}_1|, ..., |\mathcal{I}_\alpha|\}$ and saves it as $\mathcal{I}^*$. Then $\mathcal{S}$ generates random data for $\mathcal{R}^* = \{|\mathcal{R}_1|, ..., |\mathcal{R}_\beta|\}$ similarly.

Now, for the i^{th} function $\mathcal{F}_i$ in C_η, $\mathcal{S}$ accesses $\mathcal{I}^*[j]$ where $j \in \mathcal{A}_p(i)$, $\mathcal{S}$ accesses $\mathcal{R}^*[j]$ where $j \in \mathcal{B}_p(i)$, $\mathcal{S}$ replaces value in $\mathcal{R}^*[j]$ where $j \in \mathcal{U}_p(\mathcal{H}_\eta)(i)$ with new random and finally during the last operation $\mathcal{S}$ generates random data of size $|\mathcal{O}|$ and sets it to $\mathcal{O}^*$.

Since each component of $v_R(\mathcal{H}_\eta)$ and $v_S(\mathcal{H}_\eta)$ are computationally indistinguishable, we conclude that the proposed schema satisfies the security definition. □

4.2 Information Leakage Discussion

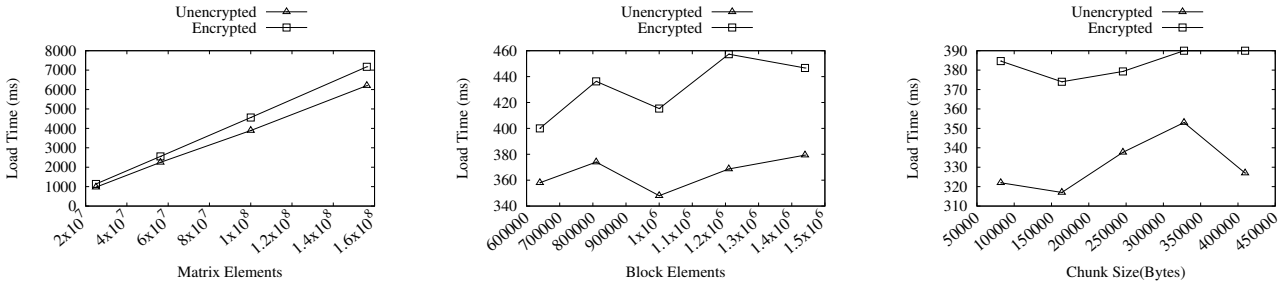
As we discussed all the data that is kept outside of the enclave is encrypted using AES-GCM mode, the storage does not leak any information and any modification to the stored data can be detected easily.

Although, our proposed framework is data oblivious, as stated in the above proof, we allow certain information leakage for efficiency. Intuitively, we allow the adversary to know the input and output size of a function. In addition, since trying to hide intermediate operation types would be too costly, we allow the adversary to know/infer intermediate input output operations required for the execution of a function. If we were to hide the operation type, we would have to perform equal number of operations for all functions (e.g., trying to hide whether we are doing secure matrix multiplication versus secure matrix addition on two encrypted matrices). Otherwise, the adversary will learn some information about the performed function. In our experimental evaluation, we observed that the overhead varies widely based on the intermediate functions. So, forcing all the functions to perform the exact same number of operations would make the framework very inefficient especially for large data sets.

Another issue is whether the size of the intermediate results can disclose any sensitive information. All of the matrix operations in our framework have fixed size outputs given the input data set size. Therefore, the size information is already inferable by knowing the matrix operation type and the input data set size. Therefore, intermediate result size does not disclose any further information.

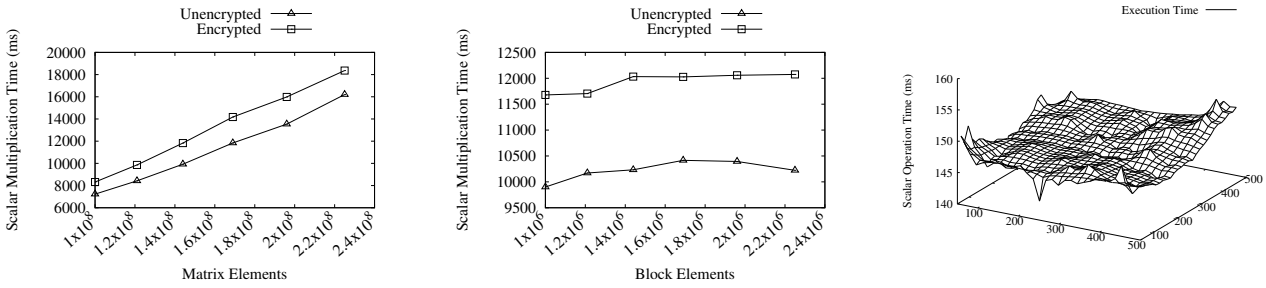
In some cases, to prevent leakage due to revealing intermediate result size, we may skip certain optimization heuristics. For example, as observed in [56], the heuristic of pushing selection predicates down the relational algebra operation tree may be skipped to prevent intermediate result size leakage. So our optimization heuristics discussed in subsection 3.7 could be turned off to prevent this type of leakage.

In other cases, intermediate results may reveal some sensitive information. For example, consider the statement `s = count( where(X[1]>=0))` discussed in subsection 3.4 where we learn the



(a) Matrix Size (b) Block Size (c) Chunk Size

Figure 4: Load time encrypted vs. unencrypted



(a) Matrix Size (b) Block Size (c) Block Variation

Figure 5: Scalar Multiplication time encrypted vs. unencrypted for different matrix (a) and block size (b). Surface plot of encrypted execution time for different block size (c).

number of tuples in `X` that has column 1 value bigger or equal than `0`. If `s` value is used in an operation that results in an object creation (e.g., `y=zeros(s)`), then the sensitive `s` value could be leaked by observing the output size. To protect against such a leakage, *our compiler automatically raises a warning* as discussed in subsection 3.4. This way users may consider changing their programs to prevent such leakage. Still, we believe that this will not be an issue in many scenarios. For example, in the case studies we have conducted such a leakage never occurred.

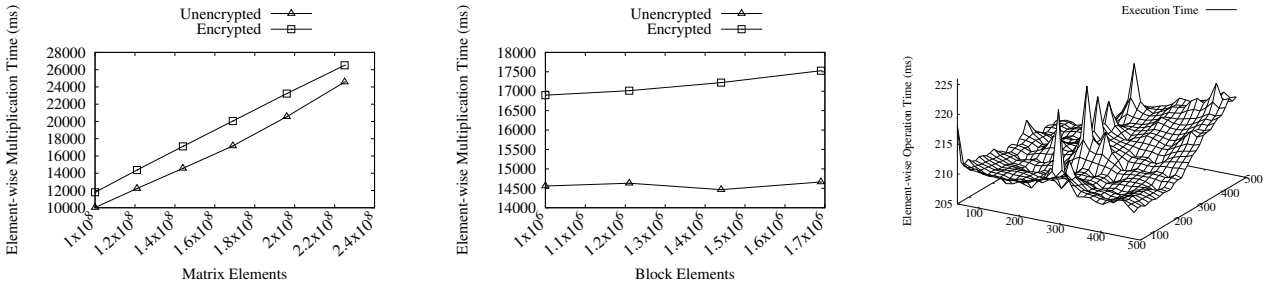
5 EXPERIMENTAL EVALUATIONS

In this section, we perform experimental evaluations to show the effectiveness of our proposed system. We developed a prototype application using *Visual Studio 2012* and *Intel Software Guard Extensions Evaluation SDK 1.0* for Windows. We perform the experiments on a *Dell Precision 3620* computer with *Intel Core i7 6700* CPU, *64GB* RAM, running *Windows 7 Professional.*

5.1 Individual Operation Performance

Experiment Setup. To understand the performance of the individual operations, we generated *random data sets* with varying sizes and observe the time it takes to perform important operations. However, we acknowledge that the time is sensitive to other events occurring on the operating system. So we rerun the same experiment (minimum 5 times) and report the average time. In addition, for all the individual operation experiments, we reported the results from encrypted and unencrypted version of our operations. For the unencrypted version, we use the SGX memory constrained environment to perform the same operations without encryption. In this way we can observe the encryption overhead of the system. We did not consider an implementation outside the enclave as a base line, because we observe that the same operations inside enclave takes significantly longer time compared to the outside enclave version. This might be due to the fact that SGX by itself does encryption of the pages and cannot really utilize existing caching mechanism. Finally, to ensure the correctness of our framework we collected data access trace of all the operations for different inputs of the same size and checked whether they match.

Load Operation. We start with load operation, which consists of loading data encrypted with user key, decrypt it, and store again with session key for further use (e.g., the key stored for writing intermediate results to the disk during the operation). As explained in section 3, we break a BigMatrix into smaller blocks and then load-store each block, as SGX enclave can allocate a certain amount of memory. In addition, we also observe that we cannot pass large amount of data through ECalls and OCalls. So, we had to further



(a) Matrix Size (b) Block Size (c) Block Variation

Figure 6: Element-wise multiplication execution time encrypted vs. unencrypted for different matrix size (a) and block sizes (b). Surface plot of encrypted element-wise matrix multiplication (c).

break the block to smaller chunks. Figure 4 illustrates the performance of load operation for randomly generated data. We report three different experiments. In Figure 4(a), we report load time vs matrix size for block size of 1000×1000. We observe that loading time increases with size of the matrix. In Figure 4(b), we report load time vs block size for the matrix 3000×3000. Here, we observe that certain block size causes load time to increase significantly. Finally in Figure 4(c), we report the effect of the chunk size in load time. We observe that the impact of the chunk size over the loading time is insignificant, so we do not report the chunk size experiments here. Furthermore, in each of the cases, we observe that encryption has very little overhead.

Scalar Operations. Next, we report the performance of scalar operations. We perform the scalar multiplication on varying matrix and block sizes as illustrated in Figure 5. In particular, we perform the scalar multiplication of a random value to all the elements of input matrix in a block-by-block manner and store the result as a different matrix. Here, we again observe that the operation time increases with matrix size in Figure 5(a). However, the block size change does not affect the operation time in most cases as illustrated in Figure 5(b). In Figure 5(c), we also report a surface plot of encrypted execution time of the scalar multiplication. Here x, and y axis represents block row and block column numbers, respectively, i.e., a point in x, y plain represents a block dimension, and z axis represents the execution time. We observe that the execution time remains steady and shows steady growth.

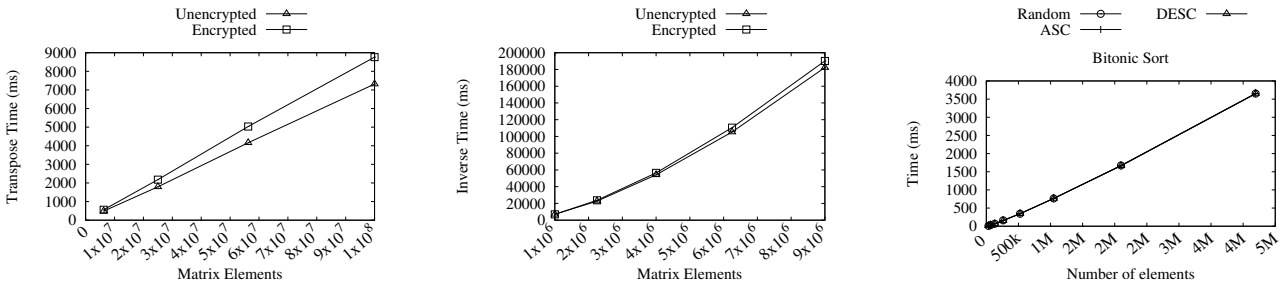
Element-wise Operation. Next, we report the performance of element-wise operations. For an element-wise operation, we take two randomly generated matrix and perform an element-wise multiplication and store the result. Similar to the scalar operation, we observe that the operation time is almost linearly proportional to the matrix size (in Figure 6(a)). Also we observe that the block size does not have huge effect on the operation time (in Figure 6(b)).

Matrix Multiplication Operation. In Figure 8, we report the time required to perform the matrix multiplication of two randomly generated matrices of varying matrix size and block size. Similar to the previous cases, we observe that matrix multiplication time linearly depends on matrix size (in Figure 8(a)). However, here we also observe that the overhead of encryption is very low due to the intensive computation required for matrix multiplications. In addition, we observe a big difference in various block sizes as illustrated in Figure 8(b). Here we observe a steady growth in the operation time with the block size increment. This can be attributed to the large number of memory access for multiplication. For a larger block size, our framework has to perform a large number of memory accesses. And in this case, load-store and encryption-decryption overhead is relatively smaller compared to the memory accesses and computation. So we observe a significant increase in the operation time.

From these sets of experiments, we observe that the operation time is almost always linearly proportional to the size of the matrix. However, block size has an important and varying impact on the execution time. Each operation behaves differently based on these two parameters. We argue that this is due to the nature of the operation that we perform on blocks in memory during various operations.

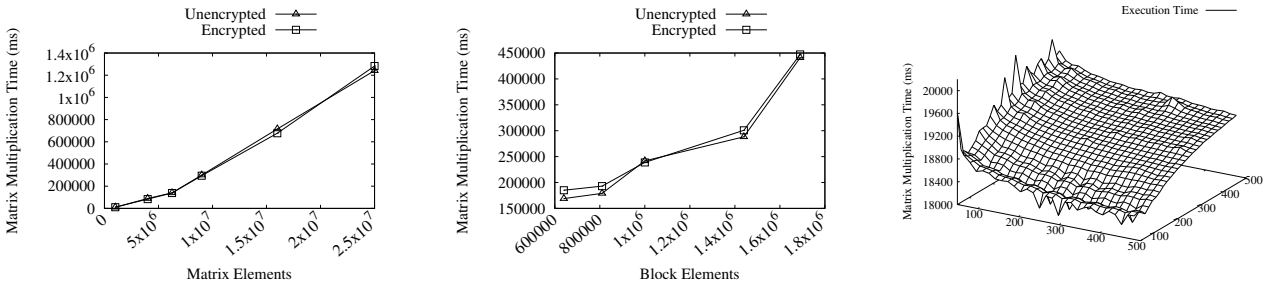
Transpose, Inverse, and Sort Operation. Next, we illustrate performance of transpose, inverse, and sort operations in Figure 7. Again, we observe that the required time is proportional to the size of input matrix. For the matrix inverse experiments, we take square matrix of different sizes and split it into 500×500 elements size blocks and perform the inverse according to our iterative matrix inverse algorithm described in Algorithm 1. We observe that the time increment is correlated with the size of the matrix. For the sort experiments, we generated three matrices one with random data, one in ascending sorted order, and one descending sorted, and ran our bitonic sort implementation. We observe that the required time is exactly the same for all three cases. This affirms our claim of data obliviousness as well.

Relational Operations. Finally, we perform the experiments that highlight the performance of relational operations. Similar to our previous experiments, we observe that relational operations also show linear growth in execution time with input matrix size as illustrated in Figure 9.



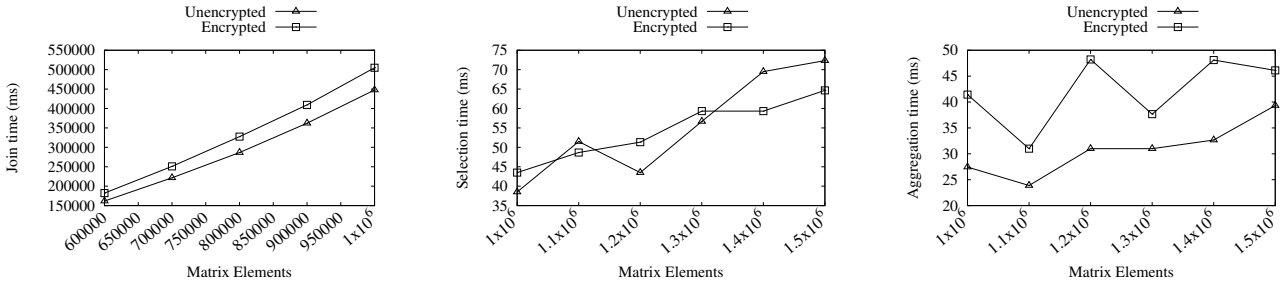
(a) Matrix transpose (b) Matrix inverse (c) Bitonic sort

Figure 7: Matrix transpose, inverse and sort operation performance.



(a) Matrix Size (b) Block Size (c) Block Variation

Figure 8: Matrix multiplication time encrypted vs. unencrypted for different matrix size (a) and block size (b). Surface plot of encrypted matrix multiplication execution time for varying block size (c).



(a) Join Operation (b) Selection Operation (c) Aggregation Operation

Figure 9: Relational operations performance encrypted vs. unencrypted.

5.2 Case Studies

In this subsection, we perform experiments to show the effectiveness of our overall framework to solve real-world complex problems and the potential information leakage in each case.

Linear Regression. We start with performing linear regression on random datasets. We chose linear regression because it is commonly used in many scientific studies [45, 51]. The time required for the execution is reported in Figure 10. We observe that the operation time is proportional to the input size. This is due to the fact each internal operation to compute θ exhibits a linear growth property. Next, we report the execution time to compute the θ on two real world machine learning datasets: USCensus1990 [43] and OnlineNewsPopularity [28] from UCI Machine Learning Repository [10]. In both cases, we take one column as the target variable and others as the input feature. The results are given in Table 1.

As we have proved in section 4, an attacker (e.g., a malicious operating system) can learn limited information due to the data analytics task execution over the encrypted data. In this case study, basically, regression is executed using a sequence of operations

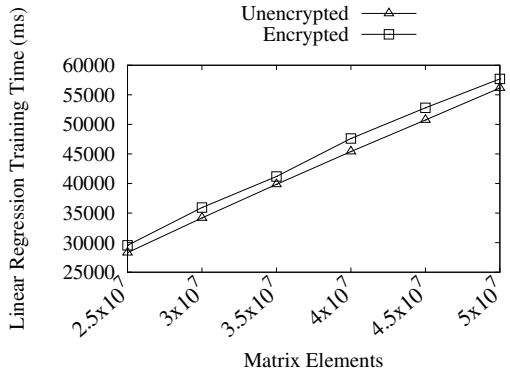


Figure 10: Linear Regression time encrypted vs. unencrypted.

Data Set	Rows	BigMatrix Encrypted
USCensus1990	2,458,285	3m 5s 460ms
OnlineNewsPopularity	39,644	2s 250ms

Table 1: Time results of linear regression on real datasets.

with fixed input, output, and block size. More specifically, for the `USCensus1990` case, an adversary can observe that we are performing a sequence of matrix operations on $n \times m$ and $n \times 1$ matrix, and we are publishing $m \times 1$ matrix, where $n = 2,458,285$, $m = 67$, and the sequence of operations are load, load, transpose, multiplication, inverse, multiplication, multiplication, and publish. The adversary can also observe the individual operation's input-output size. This information is trivially leaked based on the operation types and the input data set size. In addition, the adversary can know the block size used in each operation. In summary, an attacker *can only* infer that regression analysis is done over a matrix of size $n \times m$ for specific n and m values, nothing else.

PageRank. We chose PageRank as another case study, since it has been extensively used in link analysis. In our experiments, We use 3 directed graph datasets: Wikipedia vote network [38], Astro-Physics collaboration network [39] and Enron email network [40] from Stanford Network Analysis Project [9]. We generate the adjacency matrix of these networks and perform 40 iteration of PageRank. The execution time is reported in Table 2. We observe that as the dataset size increases the time increases significantly. That is because the total number of elements of a matrix increases quadratically as the number of nodes increases.

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 2: Page Rank on real datasets.

Information leakage in PageRank is a sequence of operations with input, output, and block sizes. In addition, the page rank algorithm (as described in Appendix B) has loop instructions, where it can leak the size of the loop and iteration count of the loop. Furthermore, the program uses a constant, i.e., the damping factor, which can be leaked too. On a side note, if a user needs to hide a value, our current implementation requires the user to input it as data rather a hard-coded constant in the program. Specifically, for `Wiki-Votes` example, an adversary can know that the user is performing a sequence of operations over a matrix of size $m \times m$ and output another $m \times 1$ matrix, where $m = 7,115$ and the sequence of operations are load, assign, assign, rand, norm, scalar, scalar, sub, div, ones, scalar, element_wise, loop, multiply, and publish. The adversary can also observe the size of input output of each operation. In addition, the adversary can also observe the block size used in each operation. In summary, the adversary *can only infer* that PageRank is executed over a $m \times m$ matrix, and nothing else.

Join oblivious vs. non-oblivious. We test the overhead of obliviousness in SQL JOIN query. We take the Big Data Benchmark [3] from AMP Lab and run a join query `SELECT * FROM Ranking r JOIN UserVisits uv (20) ON (r.pageURL = uv.destURL)` in oblivious and non oblivious mode for the small version of the dataset, where `Ranking` table contains 1,200 rows and 3 columns and `UserVisits` table contains 10,000 rows and 9 columns. We observe that the non-oblivious version takes 3min 46.3sec and the oblivious version takes 24min 12.47sec. The main reason behind the oblivious version being slower is that the value of K (i.e., the intermediate join size upper limit) is relatively high. In general, for join operation the overhead in oblivious version is mainly controlled by the parameter K. In this setting, an adversary *can only infer the input size* and the value of K, nothing else.

Comparison with a SMC Implementation. Finally, for the sake of completeness, we also compare our result with a popular multi-party computation programming abstraction ObliVM [41]. Here we perform matrix multiplication for varying size matrices using ObliVM generated code and our BigMatrix construct. As expected, we observe that the ObliVM takes significant amount of time compared to our solution with Intel SGX in Table 3. A solution using traditional multi-party circuit evaluation technique will always incur high overhead compared to a hardware assisted solution, because of the intensive communication and complex cryptographic operations. Due to the huge performance difference, we did not conduct more complex comparisons involving ObliVM.

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 3: Two-party matrix multiplication time in ObliVM vs. BigMatrix.

6 RELATED WORK

Because of the availability and sound security guarantees, Intel SGX is already used in many studies to build secure systems. For instance, Schuster et al. [50] proposed a data analytics system named VC3 that can perform Map-Reduce programs with the protection from SGX. However, VC3 does not provide any side channel information leakage protection and the authors used a simulator to report the result. Therefore, Dinh et al. [25] proposed random shuffling to protect *some* information leakage of VC3. Most recently, Chandra et al. [23] proposed using data noise to further mitigate these side channel leakages. One can argue that with Map-Reduce some of the operations proposed in our framework can be performed but it is very well known that different matrix operations such as matrix multiplication performs poorly in Map-Reduce based system. In practice, matrix multiplication using map-reduce is only feasible for sparse matrix. In contrast, our framework is data oblivious and we do not use any data specific assumption.

Haven [19] is another system that described the ways to adopt SGX to run ordinary application in a secure manner. However, the way of running legacy binaries as in Haven can introduce a controlled side channel attacks with SGX [55]. Recently, T-SGX [52] and SGX-LAPD [29] have attempted to defeat these controlled side channel attacks.

There are many other use cases of SGX. In [14], the authors proposed a secure container mechanism that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. In [36], the authors proposed protecting the confidentiality and integrity of systems logs with SGX. In [18], the authors proposed using SGX for computer game protection. In [21], authors used Intel SGX in building secure Apache Zookeeper [2], which is a centralized service to manage configurations, naming, etc. in a distributed setting. Here authors provided transparent encryption to ZooKeeper's data.

In [30], the authors theoretically analyzed the SGX system and proposed a mechanism to use SGX for efficient two-party secure function evaluation. In [16], the authors also theoretically analyzed isolated execution environments and proposed sets of protocols to secure communication between different parties.

In [46], the authors proposed oblivious multi-party machine learning using SGX based analysis. Here authors proposed mechanism to perform different machine learning algorithm using SGX. For each algorithm authors proposed a different mechanism to handle large dataset. No centralized data handling method was mentioned in the work. In contrast, our work is focused on building a generic system that can easily be extended and used for large scale data analytics task that may involve data processing, querying and cleaning in addition to machine learning tasks. Furthermore, we consider our work as complimentary to this work since some of these machine learning techniques could be provided as library functions in our generic language.

For SQL query processing in a distributed manner in [56], the authors proposed a package for Apache Spark SQL named Opaque, that enables very strong security for DataFrames. Opaque offers data encryption and access pattern hiding using Intel SGX. However, this work does not provide a general language that can be used to do other computations in addition to SQL queries. Our proposed framework supports SQL query capabilities in addition to more generic vectorized computations.

In addition to SGX based solutions, there has been a long line of research on building systems using secure processors. TrustedDB [15], CipherBase [13], and Monomi [54] uses different types of secure hardware to process queries over encrypted database. Again, these systems mainly focused on sql type processing and do not provide a generic language for handling data analytics tasks.

7 CONCLUSION

In this work, we proposed an effective, transparent, and extensible mechanism to process large encrypted datasets using secure Intel SGX processor. Our main contribution is the development of a framework that provides a generic language that is tailored for data analytics tasks using vectorized computations, and optimal matrix based operations. Furthermore, our framework optimizes multiple parameters for optimal execution while maintaining oblivious access to data. We show that using such abstractions, we can perform essential data analytics operations on encrypted data set efficiently. Our empirical results show that the overhead of the proposed framework is significantly lower compared to existing alternatives.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. The research reported herein was supported in part by an NIH award 1R01HG006844 and NSF awards CNS-1111529, CNS-1228198, CICI-1547324, IIS-1633331, CNS-1564112, and CNS-1629951.

REFERENCES

[1] Apache Spark - Lightning-Fast Cluster Computing. http://spark.apache.org/. Accessed 5/16/2017.
[2] Apache ZooKeeper. https://zookeeper.apache.org/. Accessed 5/16/2017.
[3] Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/. Accessed 5/16/2017.
[4] GNU Octave. https://www.gnu.org/software/octave/. Accessed 5/16/2017.
[5] Matlab. https://www.mathworks.com/products/matlab.html. Accessed 5/16/2017.
[6] Numpy. http://www.numpy.org/. Accessed 5/16/2017.
[7] Pandas - Python Data Analysis Library. http://pandas.pydata.org/. Accessed 5/16/2017.
[8] R: The R Project for Statistical Computing. https://www.r-project.org/. Accessed 5/16/2017.
[9] Stanford Network Analysis Project. https://snap.stanford.edu/. Accessed 5/16/2017.
[10] UCI Machine Learning Repository: Data Sets. https://archive.ics.uci.edu/ml/datasets.html. Accessed 5/16/2017.
[11] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. 2006. Sovereign joins. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 26–26.
[12] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.
[13] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase.. In *CIDR*. Citeseer.
[14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel OâĂŹKeeffe, Mark L Stillwell, et al. 2016. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symp. Operating Systems Design and Implementation*.
[15] Sumit Bajaj and Radu Sion. 2014. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on* 26, 3 (2014), 752–765.
[16] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. 2016. Foundations of hardware-based attested computation and application to

SGX. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 245–260.
[17] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 307–314.
[18] Erick Bauman and Zhiqiang Lin. 2016. A Case for Protecting Computer Games With SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*. Trento, Italy.
[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.
[20] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.
[21] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 16th Annual Middleware Conference (Middleware)*.
[22] Ernie Brickell and Jiangtao Li. 2011. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2* 1, 1 (2011), 3–33.
[23] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. 2017. Securing Data Analytics on SGX With Randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security*. Oslo, Norway.
[24] Victor Costan and Srinivas Devadas. *Intel sgx explained*. Technical Report. Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.
[25] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2r: Enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*. 447–462.
[26] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf. Accessed 5/16/2017.
[27] Ramez Elmasri. 2008. *Fundamentals of database systems*. Pearson Education India.
[28] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. 2015. A Proactive Intelligent Decision Support System for Predicting the Popularity of Online News. In *Progress in Artificial Intelligence*. Springer, 535–546.
[29] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. 2017. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17)*. Atlanta, Georgia. USA.
[30] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *Proceedings of the 2016 FC Workshop on Encrypted Computing and Applied Homomorphic Cryptography*.
[31] Charles L Hamblin. 1962. Translation to and from Polish Notation. *Comput. J.* 5, 3 (1962), 210–213.
[32] Franz E Hohn. 2013. *Elementary matrix algebra*. Courier Corporation.
[33] Intel. Product Change Notification - 114074 - 00. https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf. Accessed 5/16/2017.
[34] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *NDSS*, Vol. 20. 12.
[35] John King and Roger Magoulas. 2016. 2016 Data Science Salary Survey. http://www.oreilly.com/data/free/2016-data-science-salary-survey.csp. (September 2016).
[36] Vishal Krandle, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. Securing System Logs with SGX. In *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security*. Abu Dhabi, UAE.
[37] Tze Leung Lai, Herbert Robbins, and Ching Zong Wei. 1978. Strong consistency of least squares estimates in multiple regression. *Proceedings of the National Academy of Sciences of the United States of America* 75, 7 (1978), 3034.
[38] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 1361–1370.
[39] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2.
[40] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[41] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 359–376.
[42] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 10.
[43] Christopher Meek, Bo Thiesson, and David Heckerman. 2002. The Learning-Curve Sampling Method Applied to Model-Based Clustering. *Journal of Machine Learning Research* 2 (2002), 397.
[44] Muhammad Naveed, 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, 644–655.
[45] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. 1996. *Applied linear statistical models*. Vol. 4. Irwin Chicago.
[46] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 619–636. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko
[47] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.
[48] Rafael Pass, Elaine Shi, and Florian Tramer. 2016. Formal Abstractions for Attested Execution Secure Processors. Cryptology ePrint Archive, Report 2016/1027. (2016). http://eprint.iacr.org/2016/1027.
[49] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*. 431–446.
[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 38–54.
[51] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 936. John Wiley & Sons.
[52] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.
[53] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*. 299–310. https://doi.org/10.1145/2508859.2516660
[54] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 289–300.
[55] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 640–656. https://doi.org/10.1109/SP.2015.45
[56] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. 2017. Opaque: A Data Analytics Platform with Strong Security. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

A BIGMATRIX API DESIGN

In this Appendix section we discuss more implementation details of different important BigMatrix operations. In addition to the description of the operations, we also include the discussion of the *trace* and the *cost*. We call the information leakage to the adversary the *trace*. In general, the *trace* contains any information that an adversary can observe from the inputs, and also entering calls (ECalls) and out calls (OCalls) made by an enclave. The *cost* is the computation and communication cost of each operation. Since, individual operations are data independent and these costs will be the same for every possible input and given only the trace as input, we will be able to compute the cost. During our programming language construction, we use these cost functions to find the optimal execution plan.

Notations. We use $A[i, j]$ to mean the element of matrix A at i^{th} row j^{th} column. $A[i, j : y]$ indicates y number of elements of i^{th} row from j^{th} column to $(j + y)^{th}$ column. $A_{(p,q)}$ represents the block at p^{th} row and q^{it} column. $A_{(p:x,q:y)}$ means a sub-matrix of

x row blocks and y column blocks of A starting at p^{th} row block and q^{th} column block.

A.1 Matrix scalar operation

Let, A be a $m \times n$ matrix that is split into $p \times q$ blocks, $\odot$ be a binary operation, v be a value, and C be the output matrix of same dimensions. So the scalar operation can be defined as $C[i,j] = A[i,j] \odot v$. Using BigMatrix abstraction we perform,

$$C_{(\alpha,\beta)} = A_{(\alpha,\beta)} \odot v$$

for all the $1 \leq \alpha \leq p$ and $1 \leq \beta \leq q$ to compute desired output.

The **trace** of this operation consists of size of the matrices, the block size, the sequence of read block requests of A, and the sequence of write block request for C. After loading a block we access all the elements once and we do not perform any data dependent operations. As a result, this operation is data oblivious, i.e., the adversary will not be able to distinguish two datasets from the traces.

A.2 Matrix element-wise operation

Let, A and B be two matrices of $m \times n$ dimension, and $\odot$ be a binary operation such as multiplication, addition, subtraction, division, bit-wise and, bit-wise or, etc., C be the output of o operation applied element-wise between A and B. Meaning, $C[i,j] = A[i,j] \odot B[i,j]$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$, where $A[i,j]$ means i^{th} row and j^{th} column element in matrix A.

Now, let's assume that A, B and C is too large to fit into the enclave memory and A, B, C are split into $p \times q$ number of blocks. Using BigMatrix abstraction we perform

$$C_{(\alpha,\beta)} = A_{(\alpha,\beta)} \odot B_{(\alpha,\beta)}$$

for all the $1 \leq \alpha \leq p$ and $1 \leq \beta \leq q$ to compute desired output.

The **trace** for this operation consists of the size of matrices, the block size, the sequence of read requests block-by-block for A, B, and the sequence of write request for C. Once in memory each element is touched only once. Furthermore, we are not performing any data dependent operations.

A.3 Matrix multiplication

Let, A be a $m \times p$ matrix, B be a $p \times n$ matrix, A be split into $q \times s$ blocks, B be split into $s \times r$ blocks, and C be the output of AB. We can compute C with

$$C_{(\alpha,\beta)} = \sum_{\sigma=1}^{s} A_{(\alpha,\sigma)} B_{(\sigma,\beta)}$$

where $M_{(x,y)}$ indicates (x,y) block of matrix M.

The **trace** of this operation contains the size and block size of A, B, and C, the sequence of read requests for matrix A, B, and the sequence of write request for C. Similar to previous operations we do not perform any data dependent operations so this operation is data oblivious.

A.4 Matrix inverse

Performing matrix inverse is comparatively complicated than other operations. Let A be a square matrix split into four blocks

$$A = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where E and H are square matrices with dimensions $m \times m$ and $n \times n$, respectively. So, F and G are $m \times n$ and $n \times m$ dimension array. The inverse can then be computed

$$A^{-1} = \begin{pmatrix} E^{-1} + E^{-1}FS^{-1}GE^{-1} & -E^{-1}FS^{-1} \\ -S^{-1}GE^{-1} & S^{-1} \end{pmatrix}$$

where, $S = H - GE^{-1}F$. Also, E and S must have non-zero determinants. This format requires several multiplications and inverses. In a naive implementation, we will need a large amount of temporary memory. We can perform the following sequence of operations to inverse a matrix with manageable memory overhead.

- We perform E^{-1} in place and our BigMatrix internal state is as follows
$$\begin{pmatrix} E^{-1} & F \\ G & H \end{pmatrix}$$
- We multiply E^{-1} times block F and negate the result and replace F with the result. BigMatrix internal state is as follows
$$\begin{pmatrix} E^{-1} & -E^{-1}F \\ G & H \end{pmatrix}$$
- Next, we multiply G with $-E^{-1}F$ and subtract from H and replace H, leading to BigMatrix internal state of
$$\begin{pmatrix} E^{-1} & -E^{-1}F \\ G & H - GE^{-1}F \end{pmatrix}$$
Here, $H - GE^{-1}F$ is S.
- We compute S^{-1} and replace S, so we have
$$\begin{pmatrix} E^{-1} & -E^{-1}F \\ G & S^{-1} \end{pmatrix}$$
- Next, we compute GE^{-1} and replace G, so we have
$$\begin{pmatrix} E^{-1} & -E^{-1}F \\ G & S^{-1} \end{pmatrix}$$
- Now we compute $S^{-1}GE^{-1}$ by multiplying the last two results. We negate the result and replace G, so our BigMatrix looks like
$$\begin{pmatrix} E^{-1} & -E^{-1}F \\ -S^{-1}GE^{-1} & S^{-1} \end{pmatrix}$$

- We multiply the off diagonal elements and add it to E^{-1} block, so that we have

$$\begin{pmatrix} E^{-1} + E^{-1}FS^{-1}GE^{-1} & -E^{-1}F \\ -S^{-1}GE^{-1} & S^{-1} \end{pmatrix}$$

- Finally we multiply $-E^{-1}F$ with S^{-1}, replace $-E^{-1}F$ and we get the intended result.

$$\begin{pmatrix} E^{-1} + E^{-1}FS^{-1}GE^{-1} & -E^{-1}FS^{-1} \\ -S^{-1}GE^{-1} & S^{-1} \end{pmatrix}$$

We can perform these operations with temporary memory equal to the size of input BigMatrix. Now, we have built an iterative algorithm to perform the inverse. It starts with block (0, 0) and in each iteration it expands inverse by one block as described in Algorithm 1. In this algorithm we need to inverse 1×1 blocks. To achieve that we use a traditional LU decomposition technique with a fixed number of rounds depending on the size of matrix not on the data.

Algorithm 1 Matrix inverse by block iterative method.

Require: A = Square matrix split into blocks
$A_{(0:1,0:1)} = inverse(A_{(0:1,0:1)})$
for $i = 1$ to number of blocks in A **do**
  $e = (0 : i, 0 : i)$
  $f = (0 : i, i : 1)$
  $g = (i : 1, 0 : i)$
  $h = (i : 1, i : 1)$

  $A_f = -1 * A_e * A_f$
  $A_h = A_h + A_g * A_f$
  $A_h = inverse(A_h)$
  $A_g = A_h * A_e$
  $A_g = -1 * A_h * A_g$
  $A_e = A_e + A_f * A_g$
  $A_f = -1 * A_f * A_h$
end for

The **trace** of the matrix inverse performed in blocks consists of the trace of individual operations in sequences mentioned by Algorithm 1. Similar to previous operations, this operation does not perform any data dependent execution so it is data oblivious.

A.5 Matrix Transpose

Let, A be a matrix of dimension $m \times n$, which is split into $p \times q$ blocks, C be the transpose of A. $C[i, j] = A[j, i]$ for all elements of A. To compute C in our BigMatrix abstraction we compute

$$C_{(\alpha,\beta)} = transpose(A_{(\beta,\alpha)})$$

for $1 \le \alpha \le p$ and $1 \le \beta \le q$.

The **trace** of the transpose operation is the size of the matrix, the block size, the sequence of read requests for blocks of A, and the sequence of block write requests for block of B. Furthermore, while in memory each element value is touched only once and we do not perform any data dependent operation. As a result, the transpose operation is data oblivious.

A.6 Sort and Top k

We use Bitonic Sort [17] that performs exactly the same number of comparisons for the same size dataset. However, the comparison function in bitonic sort needs special attentions in order to make it data oblivious. In particular, we used registers to determine the comparison result of two rows and swap the values accordingly. To make our framework more practical we allow users to mention a list of column numbers and the direction of sort for each column. To make the overall sort operation oblivious, for each row, we read the full column and touch all the columns, compute a flag value and swap two rows based on the flag. For top k results, we perform the full sort and keep only the top k results based on the given criteria.

The **trace** of the sort function consists of the size of input matrix, the block size, and the sequence of read and write request for the matrix. We take input of the sorting direction as a row vector where each element belongs to $\{0, 1, -1\}$, 0 meaning no sorting direction, 1 meaning ascending order, and -1 meaning descending order sorting. As a result, there is no leakage through sorting order input.

A.7 Selection

Our framework also supports a number of most commonly used relational algebra operators. However, these operations are not data oblivious by nature. Therefore, we have to modify these operations to make them data oblivious.

Let, A be a matrix of $m \times n$ dimensions, φ be a propositional formula consisting one or more *atoms*, *match* be a function that takes input a row of the matrix A, a propositional formula φ and outputs 0 or 1 based on the result of the conditional predicate on the row, and C be the output. In our framework C is defined as a column vector (matrix of $m \times 1$ dimension) and computed as

$$C[i, 0] = match_{\varphi}(A[i, 0 : n])$$

for all $1 \le i \le m$. In this way, the output size is always the same, so no information leakage through output size. Next, we focus on building the *match* function in a data oblivious manner. First, we argue that we have to leak the size and type of the operation in our propositional formula. If we want to hide it then we always have to execute a constant number of conditional operations in every possible case, anything other than that would leak information about the φ. Furthermore, φ can be arbitrarily large and complex. So hiding φ for security will make the framework very inefficient. On the other hand, we can easily hide the columns that are used in φ. We simply touch all the values in input row in each match execution.

The **trace** of the selection operation consists of the size of input matrix, the block size, the sequence of read requests for input matrix, and the matching expression size. Here we perform data dependent operations but we do exactly same operations for the same number of input expressions and input rows. We hide the selection expression content by touching all the element of input matrix row and evaluating the selection expression to find whether current row matches or not. So we argue that our implementation is data oblivious.

A.8 Aggregation

In our framework, we support four aggregation commands, `sum`, `average`, `count`, `min`, and `max`. Each of these aggregation operations requires different types of processing. By definition `sum`, `average`, `count` are oblivious since the number of operations does not depend on the data in anyway. However, `min` and `max` depend on the data. In a trivial implementation min of max computation between two number reveals branch of the code that is executed by a processor. As a result, the adversary can distinguish between two different datasets. To remedy that we used techniques described in [46, 49]. Specifically, we load the values into a register (that is not observable by the adversary), compute the condition that set a flag, based on the flag we swap, and return value from one fixed register. In this process the number of operation remains the same, and the same path of the code is executed regardless of the input data.

The **trace** of our aggregation operation is the size of input matrix, the block size, the number of aggregation operation, and the type of aggregation operation.

A.9 Join

We only considered a simple join without any special optimizations. We adopted [11] technique to perform join between two BigMatrix. Similar to their constructs, we require users to supply the maximum number of matches in B with A, without this information the implementation of join operation will become data dependent. Let, A be matrix of dimension $m \times n$, B be matrix of dimension $x \times y$, φ be propositional formula consisting of atom, *match* be a function that takes one row from A and another row from B, outputs 1 if rows matches on given columns and 0 otherwise, and k be the number of maximum rows in B that matches with any row of A. We use Algorithm 2 to compute join. For simplicity and efficiency we are considering only BigMatrix that have one column blocks. It makes it easier to compute the matching condition obliviously. In case, if input matrix is not in this format we can run *reshape* operation to make it into this shape. Since we are considering only BigMatrix with single column we will use A_p to indicate p^{th} block. The details of this join algorithm is given in Algorithm 2.

Algorithm 2 Data oblivious join algorithm for BigMatrix

Require: A, B input BigMatrix, that has only one column block, φ matching condition, k = maximum row matches from A to B
Output: C output BigMatrix.
for $u = 1$ to $row\_blocks(A)$ **do**
  $load\_block\ A_u$
  for $i = 1$ to $rows(A_u)$ **do**
    $X = 2k$ dummy block array
    $t = K$
    for $v = 1$ to $row\_blocks(B)$ **do**
      $load\_block\ B_v$
      for $j = 1$ to $rows(B_v)$ **do**
        if $match(A_u[i,:], B_v[j,:], \varphi)$ **then**
          $X[t] = A_u[i,:], B_v[j,:]$
        else
          $X[t] = dummy, dummy$
        end if
        $t = t + 1$
        if $t >= 2k$ **then**
          Sort X with bitonic sort such that dummy blocks at the end.
        end if
      end for
    end for
    Write first k elements to C
  end for
end for

B ADDITIONAL EXAMPLE : PAGERANK

PageRank is a popular algorithm to measure the relative importance of a node in a connected graph [47]. It was originally used to measure the importance of hyperlinked web pages in Word Wide Web. The simplified version of the algorithm can be expressed as

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

where u, v are nodes in a connected graph, $PR(v)$ is PageRank of v, B_u is a set of nodes that links to u, and $L(v)$ is number of links from v. Finally, we iterate multiple times until the values converge. Interestingly, we can express the computation in terms of basic matrix operations using a technique called *power method*. Also, to reduce the information leakage through iteration required to converge, we run the update step a fixed number of times. In our programming language, we can write the code as follows

```
M = load('path/to/adjacency_matrix')
d = 0.8            // damping factor
N = M.rows

v = rand(N, 1)
v = v ./ norm(v, 1)

M_hat = (M .* d) + ones(N, N) .* (1 - d) / N

for _ = 1 to 40:
        v = M_hat * v
publish(v)
```

The corresponding execution engine code is as follows. For simplicity we are skipping the `unset` methods here.

```
M = load(adjacency_matrix_id)
d = assign(0.8)
N = assign(M.rows)

v = rand(N, 1)
t1 = norm(v, 1)
v = scalar('/', v, t1)

t2 = scalar ('*', M, d)
t3 = sub(d, 1)
t4 = div(t3, N)
t5 = ones(N, N)
```

```
t6 = scalar('*', t5, t4)
M_hat = element\_wise('+', t2, t6)

_ = loop(1, 40, 1)
v = multiply(M_hat, v)

publish(v)
```

In this case, the leaked information to the adversary is the size of `M`, the loop iteration count 40, the looped instruction count 1, and the sequence of operations.

C PROTOCOL DESIGN DETAILS

Code agreement and loading phase. To facilitate the communication among multi-parties, we assume that the participants know each others' and also SGX server's public key. This can be achieved by participating in an already existing public key infrastructure. In addition, we are assuming there exists a broadcast mechanism, where any participants including the server can broadcast messages to every other participant.

Let, p be the number of participants, P_i be the i^{th} participant, $K^{(i)}_{pub}$ be the public key of participant i, $K^{(i)}_{pri}$ be the private key of participant i, $K^{(s)}_{pub}$ be the public key of the central server, $K^{(s)}_{pri}$ be the private key of the central server, C be the code that all the participants wants to execute, $H(k, m)$ be an authenticated hash (HMAC) function that creates MAC of a message m with key k, $Sign(K_{priv}, m)$ be a signing function that generates fixed length signature s of message m with a private key of an asymmetric key pair, and $Verify(K_{pub}, s, m)$ be a verification algorithm that verifies signature s of message m with a public key of an asymmetric key pair.

The sequence of operations that participant P_i performs in this phase is the following:

- Generate a signature for the code C with a randomly generated nonce r_i as follows,

$$\sigma_i =< s_i, r_i >=< Sign(K^{(i)}_{pri}, C||r_i), r_i >$$

- Broadcast σ_i to all other participants
- Next get all other participants signatures, i.e., get σ_j for $j = \{1, ..., p\}$ and $j \neq i$
- Verify by executing $Verify(K^{(j)}_{pub}, s_j, C||r_j)$ for all j except i. If any of the signature fails then abort the protocol and broadcast the abort message

At this stage all the participants have agreed on the same code C. Now we are ready to start the SGX loading

- One of the participant uploads C and $\{\sigma_1..\sigma_p\}$ to the SGX server
- The server verifies all the $\{\sigma_1..\sigma_p\}$ as previously
- Next, the server creates the enclave, i.e., loads the trusted part of the code into SGX
- Generates the signature of the enclave from `mrenclave` register call
- Inside the enclave generate asymmetric key pair $K^{enclave}_{pub}, K^{enclave}_{pri}$
- The server generates the following λ_i for all the participants P_i and send to participants

$$\lambda_i =< Sign(K^{enclave}_{pub}, C), ESig, \phi(K^{(i)}_{pub}, K^{enclave}_{pub}||K_i||r_i), r_i >$$

- The server also generates a random session key for this execution K_s, which will be used for further computation in this session
- Each participants gets λ that they decrypt with their private key and get K_i

Input data and encryption key provisioning. Once direct key establishment with SGX server is done, we are ready to send data to the server.

- Now participant i generates a random symmetric key K_i and encrypts the key with a key derived from nonce n from previous step
- Participant i then encrypts the data with K_i and uploads to the SGX server

Result distribution. Upon finishing the code execution the SGX server will distribute the result, which is encrypted with recipient's public key K_i.