SECURE CLOUD DATA ANALYTICS WITH TRUSTED PROCESSORS

by

Fahad Shaon

Dedicated to my family.

SECURE CLOUD DATA ANALYTICS WITH TRUSTED PROCESSORS

by

FAHAD SHAON, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2019

ACKNOWLEDGMENTS

First, I would like to thank Dr. Murat Kantarcioglu for showing me the path in this incredible world of research and for teaching me how to push the boundaries of human knowledge. I would especially like to thank him for being there for all the ups and downs of my life for the last few years.

I would like to thank Dr. Latifur Khan, Dr. Bhavani Thuraisingham, and Dr. Shuang Hao for their interest in my research, and serving on my PhD committee. Their valuable comments helped me greatly in improving this dissertation.

During my graduate study, I had a chance to work with brilliant friends and colleagues - Sazzadur Rahman, Cuneyt Akcora, Erman Pattuk, Huseyin Ulusoy, Imrul Anindya, Aref Asvadishirehjini, to name a few. I greatly value and appreciate their friendship. Special thanks to Sazzadur Rahman for being the go-to person for debating and validating new ideas. Also special thanks to Cuneyt Akcora for reviewing my pre-published manuscripts and helping me improve them.

I am also grateful to my parents, Kamal Uddin Ahmed and Sahanaz Begum, my sister, and brother for their constant encouragement and support during my graduate studies.

Finally and most importantly I would like to thank my wife Samira Farhin for her unwavering support in my pursuit of higher studies. Without her persistent support this long journey would be impossible.

November 2019

SECURE CLOUD DATA ANALYTICS WITH TRUSTED PROCESSORS

Fahad Shaon, PhD
The University of Texas at Dallas, 2019

Supervising Professors: Murat Kantarcioglu, Chair
Latifur Khan, Co-Chair

Over the last few years, data storage in cloud-based services has been very popular due to the easy management and monetary advantages of cloud computing. Recent developments showed that such data could be leaked due to various attacks. To address some of these attacks, encrypting sensitive data before sending to the cloud emerged as an important protection mechanism. Still, indexing, querying and running complex data analytics tasks on the encrypted data remained as important challenges. In this dissertation, we address some of the encrypted data processing challenges using two different but complementary approaches. First, we explore what kind of data querying functionality we can provide for encrypted data even if we have no support from the server. Later, we provide solutions for the use cases where the cloud server provides a trusted processor for processing some of the encrypted data.

For the cloud deployments where there is only limited support from the cloud server [1], we provide a new searchable encryption scheme, i.e., a type of encryption technique that allows querying on encrypted data. Unlike, most of the existing searchable encryption schemes that

[1]Cloud services such Dropbox, Box, Google Drive allow simple data retrieval and do not provide computational support (i.e., running an arbitrary code on the encrypted data)

are developed for keyword searches, our proposed scheme does not require running some code on the cloud servers. Furthermore, we provide an extensible framework for supporting complex search queries over encrypted multimedia data. Before any data is uploaded to the cloud, important features are extracted to support different query types (e.g., extracting facial features to support face recognition queries) and complex queries are converted to series of object retrieval tasks for the cloud service.

Later, we explore the setting where the cloud servers provide support for processing encrypted data using trusted processors. In this setting, we can execute code in a trusted processor in a secure manner, i.e, the adversary cannot temper with the code without detection, and data is always encrypted outside the trusted processor.

Over the past few years, efficient and secure data analytics tools (e.g., map-reduce framework, machine learning models, and SQL querying) that can be executed over encrypted data using the trusted processors 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 important security concepts such as "oblivious execution". 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. Also, we perform block size optimization and provide security guarantees for data obliviousness.

Similarly, systems to accesses encrypted inverted index using trusted processes have been developed before. However, none of these works proposed a mechanism to build the index in the cloud securely. All of these works assume that some form of unencrypted inverted index is already available. Building an inverted index can be very memory consuming task for big data on memory constraint platforms. So we propose a system to build the encrypted inverted index in the cloud using trusted processors for text as well as multimedia data in an oblivious and secure manner. We design our index to support TF-IDF based ranked document retrieval. Our system also supports indexing for answering complex queries such as face recognition.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Increasingly, individuals and organizations are adopting cloud computing for various computation needs because cloud computing provides a cost-effective alternative to traditional computational infrastructure. In general, there are no upfront cost and we pay only for the services based on usages. Moreover, we can develop and deploy software very fast using the advanced tools provided by cloud service vendors. However, there are several drawbacks as well. Adopting a public cloud increases chances of sensitive data exposure due to misconfiguration (Deahl, 2017) and poor security settings (Agarwal, 2014; Stadmeyer, 2014). Also, in public cloud computing setup, multiple users' computation is performed in a single machine for cost minimization, which exposes the user to insider attacks (Duncan et al., 2012). Finally, the cloud service provider can also observe stored unencrypted data and access patterns of the data. For some businesses, this might be detrimental as a large cloud provider can build competitive products (Novet, 2018).

To address these issues one solution is to encrypt data and code before sending these to the cloud, which allows the user to leverage cloud computing services securely. However, searching and performing data analytics on encrypted data is difficult. It is especially difficult to perform complex search queries, such as face recognition, on encrypted multimedia data without the support of advanced trusted processors. Even with the support of trusted processors we need to address several security challenges in building systems for data analytics and data indexing, such as side channels due to memory accesses. In this dissertation, we address some of these important challenges.

We start with the problem of searching encrypted multimedia data in cloud computing setting. Over the years, many searchable encryption techniques have been proposed (Curtmola et al., 2006; van Liesdonk et al., 2010; Cash et al., 2014; Kamara and Papamanthou, 2013; Cash et al., 2013; Naveed et al., 2014; Ostrovsky, 1990; Goldreich and Ostrovsky, 1996;

Bösch et al., 2015). Among those approaches, searchable symmetric encryption (SSE) (Curtmola et al., 2006; van Liesdonk et al., 2010; Cash et al., 2014; Kamara and Papamanthou, 2013; Cash et al., 2013; Naveed et al., 2014; Bösch et al., 2015) emerges as an efficient alternative for cloud based storage systems due to minimal storage overhead, low performance overhead, and relatively good security.

However, almost all searchable encryption techniques require executing some code on the cloud servers to enable efficient processing. On the other hand, popular commercial personal cloud storage providers (Dropbox, 2019a; Box, 2019a; Google, 2019a) only support basic file operations like reading and writing files that make it infeasible to apply traditional SSE techniques. Furthermore, complex queries on multimedia data may require running different and expensive cryptographic operations. These limitations create a significant problem for wide adoption of SSE techniques. Therefore, developing SSE schemes that can run on the existing cloud storage systems without requiring the cloud service providers' cooperation emerges as an important and urgent need. To our knowledge, only (Naveed et al., 2014) considered a setup without computational support from the cloud storage but the proposed solution does not support efficient complex querying over encrypted data.

Even though, one can wish that an alternative SSE as a service could be offered in the near future by the cloud service providers, due to network effects, many of the existing users may not want to switch their cloud service providers. Therefore, any new "secure" cloud storage with SSE providers may have a hard time in getting significant traction. So supporting SSE on the existing cloud storage platforms without requiring any support from the cloud storage service providers is a critical need.

In addition, adoption of multimedia (e.g., image, music, video, etc.) data for social communication is increasing day by day. KPCB analyst Mary Meeker's 2014 annual Internet Trends report (Meeker, 2014) states *1.8 billion* photos shared *each day*. However, indexing multimedia data is harder compared to text data. A significant pre-processing is required to convert raw multimedia data to a searchable format and queries made on multimedia data are complex as well. So building efficient cryptographic storage system that can easily handle multimedia content is a very important problem. In Chapter 4 we propose a generic searchable symmetric encryption scheme, where user can perform complex query on encrypted multimedia data using cloud server with only storage.

Alternatively, we can leverage trusted processors to perform computation on encrypted data in cloud settings as proposed in (Ohrimenko et al., 2016; Zheng et al., 2017; Schuster et al., 2015). 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 (Intel, 2015) 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 (Islam et al., 2012; Naveed et al., 2015) 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 Chapter 5 we propose efficient matrix computation system that can leverage SGX to perform data analytics operations securely on very large dataset.

Similarly, few systems has been proposed for encrypted index access using SGX. For instance, in Oblix (Mishra et al., 2018) authors propose few ways to access inverted index obliviously inside enclave. In HardIDX (Fuhry et al., 2017) authors propose building secure B+ index, which can later be used to build different application. In Rearguard (Sun et al., 2018) authors proposed system to retrieve list of documents obliviously from encrypted index. However, in these works, authors assume that the inverted index is already available and only focused on accessing the encrypted index securely. In contrast, we focus on building the index securely using SGX. Because building an inverted index might be trivial but it is very memory consuming computation. An inverted index is traditionally defined as a function that returns list of documents associated with input token. So, to build such index by reading a set of input documents, we need to maintain a hash map (or equivalent data structure) of the token to document lists. For a memory constraint system, such as smart phone, this computation might be infeasible. So in our design we push as much computation to the server as possible. Furthermore, these works mainly focus on building systems targeting text index. In contrast, we focus on building a secure index for text and image search in the cloud. Once we built the inverted index we can utilize existing index accessing systems to efficiently retrieve data. In Chapter 6 we propose an algorithm to build a search index for text data that can support ranked document retrieval using TF-IDF scoring. Also, we propose an algorithm to build an index on encrypted images to perform face recognition.

[1]It is worth to mention that, Intel also proposed a general dynamic memory allocation mechanism for the next version of SGX in (McKeen et al., 2016). However, to efficiently analyze very large datasets, we still need some form of memory allocation mechanisms.

The rest of the dissertation is outlined as follows. We start by discussing the related work to the solutions and problems mentioned in this dissertation in Chapter 2. Next, we review related tools and techniques that are used in this dissertation in Chapter 3. Then, we provide our solution for searching encrypted multimedia data without computation support from cloud providers in Chapter 4. Next, we discuss solutions for performing large scale encrypted data analysis leveraging trusted processors in Chapter 5. Finally, we focus on building an encrypted index in the cloud using the trusted processors in Chapter 6 and conclude the dissertation in Chapter 7

CHAPTER 2

RELATED WORKS

In this chapter, we summarize the literature studies that are relevant to our work in this dissertation.

2.1 Searchable symmetric encryption

Currently there are few ways to build encrypted cloud storage with content based search. Searchable symmetric encryption(SSE) is one of those, which allows users to encrypt data in a fashion that can be searched later on. Different aspects of SSE has been studied extensively as shown in an extensive survey of provably secure searchable encryption by Bösch at el. in (Bösch et al., 2015). Curtmola at el. (Curtmola et al., 2006) provided simple construction for SSE with practical security definitions, which was then adopted and extended by several others in subsequent work. Few works also looked into dynamic construction of SSE (van Liesdonk et al., 2010; Cash et al., 2014; Kamara and Papamanthou, 2013; Kamara et al., 2012) so that new documents can be added after SSE construction.

Another branch of study related to SSE is supporting conjunctive boolean query. Cash at el. (Cash et al., 2013) proposed such a construction, where authors used multi-round protocol for doing boolean query with reasonable information leakage. In the process they also claimed to build the most efficient SSE in terms of time and storage. Kuzu at el. (Kuzu et al., 2012) proposed an efficient SSE construction for similarity search, where they used locality sensitive hashing to convert similarity search to equality search. There are also work towards supporting efficient range query, substring matching query, etc. (Faber et al., 2015), where a rich query is converted to an exact matching query. However, these constructions require specialized server. Importantly, we can easily adopt such a conversion technique in our framework.

Naveed at el. (Naveed et al., 2014) proposed a dynamic searchable encryption schema with simple storage server similar to our setup. The system also hides certain level of access pattern. However, authors did not consider complex query problem in their work, which is one of the major challenges that we solved in this work.

Another way of querying encrypted database is oblivious RAM (ORAM) described by Ostrovsky (Ostrovsky, 1990) and Goldreich at el. (Goldreich and Ostrovsky, 1996), which also hides search access pattern and much secure. Despite recent developments (Pinkas and Reinman, 2010; Stefanov et al., 2013; Stefanov and Shi, 2013), traditional ORAM remains inefficient for practical usage in cloud storage system as described in (Bindschaedler et al., 2015; Naveed, 2015). Furthermore, our proposed system converts complex operations into sequence of key value read and write operations, which can easily be combined with ORAM technique to hide the access pattern.

Qin at el. (Qin et al., 2014) proposed an efficient privacy preserving cloud based secure image feature extraction and comparison technique. Similar construction for ranked image retrieval is proposed by (Xia et al., 2013; Lu et al., 2009; Raval, Pillutla, Bansal, Srinathan, and Jawahar, Raval et al.). These systems depend on highly capable cloud server for preforming image similarity query.

Finally, there are few commercial secure cloud storage systems, e.g., SpiderOak (SpiderOak, 2019), BoxCryptor (BoxCryptor, 2019). Even though these systems are easy to use and provide reliable security, these systems provide neither server based search nor complex query support. All these systems depend on either operating system or local indices to provide search functionalities. As a result, to provide search functionalities these systems need to download and decrypt all the data stored in cloud server, which might not be efficient solution in all circumstances.

2.2 Intel Software Guard Extension

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. (Schuster et al., 2015) 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. (Dinh et al., 2015) proposed random shuffling to protect *some* information leakage of VC3. Most recently, Chandra et al. (Chandra et al., 2017) 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 (Baumann et al., 2015) 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 (Xu et al., 2015). Recently, T-SGX (Shih et al., 2017) and SGX-LAPD (Fu et al., 2017) have attempted to defeat these controlled side channel attacks.

There are many other use cases of SGX. In (Arnautov et al., 2016), 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 (Krandle et al., 2017), the authors proposed protecting the confidentiality and integrity of systems logs with SGX. In (Bauman and Lin, 2016), the authors proposed using SGX for computer game protection. In (Brenner et al., 2016), authors used Intel SGX in building secure Apache Zookeeper (Zookeeper,

Zookeeper), which is a centralized service to manage configurations, naming, etc. in a distributed setting. Here authors provided transparent encryption to ZooKeeper's data.

In (Gupta, Mood, Feigenbaum, Butler, and Traynor, Gupta et al.), the authors theoretically analyzed the SGX system and proposed a mechanism to use SGX for efficient two-party secure function evaluation. In (Barbosa et al., 2016), the authors also theoretically analyzed isolated execution environments and proposed sets of protocols to secure communication between different parties.

In (Ohrimenko et al., 2016), 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 (Zheng et al., 2017), 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 (Bajaj and Sion, 2014), CipherBase (Arasu et al., 2013), and Monomi (Tu et al., 2013) 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.

2.3 Intel SGX based search index

In Rearguard (Sun et al., 2018) authors build search indexes for different types of keyword searches. However, authors assumed to have an initial inverted index built in client end. In HardIDX (Fuhry et al., 2017) authors proposed building secure B+ index and used it to build encrypted databases and searchable encryption schema. However, proposed algorithms are not made oblivious as a result leaks a lot of information via a side channel. Finally, in Oblix (Mishra et al., 2018) authors proposed building different type of oblivious data structures using oblivious ram (ORAM) techniques. We consider our work as a complement to these works one can build index using our techniques and use these systems later to access the inverted index.

CHAPTER 3

BACKGROUND

3.1 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) is one of the many mechanisms to enable search over encrypted data. In an SSE schema, we not only encrypt the input dataset, but also we create an encrypted inverted index. The index contains a mapping of an encrypted version of keywords (called trapdoors) to list of document ids that contain corresponding plain text keywords. Formally, an SSE schema is defined as the collection of 5 algorithms $SSE = (Gen, Enc, Trpdr, Search, Dec)$ Given security parameter Gen generates a master symmetric key, Enc generates the encrypted inverted index and encrypted data sets from the input dataset. $Trpdr$ algorithm takes keywords as input and outputs the trapdoor, which is used by the $Search$ algorithm to find the list of documents associated with input keywords. Finally, the Dec algorithm decrypts the encrypted document given the id and proper key. We refer the reader to (Curtmola et al., 2006) for a further discussion of SSE. Furthermore, in typical SSE settings, Gen, Enc, $Trpdr$, and Dec are performed in a client device and the $Search$ algorithm is performed in a cloud server.

3.2 Intel SGX

Intel SGX is a new CPU extension for executing secure code in Intel processors (Anati et al., 2013). In the 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 the intended code is loaded and securely provision the code with secret keys, which is called *attestation*.

Internally, the infrastructure uses Enhanced Privacy ID (EPID) (Brickell and Li, 2011) for hardware-based attestation. In addition, trusted and untrusted components communicate between each other using programmer-defined entry points. Entry points defined in a trusted code are called *ECalls*, which can be called by the untrusted part once the 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 (Costan and Devadas, Costan and Devadas; Pass et al., 2016).

3.3 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`[1] 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

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

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., (Ohrimenko et al., 2016; Rane et al., 2015)). The data oblivious approach of programming protects against attacks from access pattern leakage as described in (Islam et al., 2012; Naveed et al., 2015). 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.

CHAPTER 4

A PRACTICAL FRAMEWORK FOR EXECUTING COMPLEX QUERIES OVER ENCRYPTED MULTIMEDIA DATA[1]

4.1 Introduction

In this chapter, we propose an efficient searchable encryption scheme framework to query encrypted multimedia data using cloud storage service. Our proposed framework only requires file storage and retrieval support from cloud storage services. Furthermore, by leveraging the extensible extract, transform and load operations provided by our framework, users can build encrypted search systems to handle complex queries. As an example, we show how our framework could be used to run face recognition queries on encrypted images. To our knowledge, this is the first system that can support *complex queries* on encrypted multimedia data *without significant computational support* from the cloud service provider (i.e., without running customized code in the cloud). *Main contributions* of this work can be summarized as follows:

- We propose a generic outsourcing framework that enables secure and efficient querying on any data. Our framework supports complex querying on any encrypted data by allowing queries to be represented as a series of simple equality queries using the features extracted from the data. Later on, these extracted features are transformed into encrypted indexes and these indexes are loaded to the cloud and leveraged for efficient encrypted query processing.

- We prove that our system satisfies adaptive semantic security for dynamic SSE.

[1]© Shaon F., Kantarcioglu M. (2016) A Practical Framework for Executing Complex Queries over Encrypted Multimedia Data. In: Ranise S., Swarup V. (eds) Data and Applications Security and Privacy XXX. DBSec 2016. Lecture Notes in Computer Science, vol 9766. Springer, Cham. DOI: `https://doi.org/10.1007/978-3-319-41483-6_14`

- We show the applicability of our framework by applying it to state-of-the-art image querying algorithms (e.g., face recognition) on encrypted data.

- We implement a prototype of our system and empirically evaluate the efficiency under various query types using real-world cloud services. Our results show that our system introduces very little overhead, which makes it remarkably efficient and applicable to real-world problems.

The rest of the chapter is organized as follows: Section 4.2 provides the general setup and threat model of our system, Section 4.3 describes the internal details of each phase, Section 4.4 extends our initial framework making it dynamic, in Section 4.5 we formally prove the security of our system, Section 4.6 shows an application of our proposed framework, Section 4.7 shows the experimentations, and in Section 4.8 we conclude our work.

4.2 Threat Model

In this study, we consider a setup, where a user owns a set of documents that may include multimedia documents. The User wants to store these documents into a cloud storage server in encrypted form. User also wants to perform complex search queries over the encrypted data. Most importantly, the user wants to utilize the existing cloud storage service, which is not capable of executing any custom code provided by them. Formally cloud storage server $\mathcal{Z}$ can *only* perform *read* and *write* operations. This simple requirement of cloud storage server makes the system easily adaptable in several real-world scenarios. On the other hand, the user has devices with sufficient computation power that can perform modern symmetric cryptography algorithms and are called clients.

In our system, the communication between server and client is done over an encrypted channel, such as https. So eavesdroppers can not learn any meaning full information about the documents capturing the communication, apart from the existence of such communication. We also assume that the cloud storage server $\mathcal{Z}$ is managed by Bob, who is semi-honest. As such, he follows the protocol as it is defined but he may try to infer private information about the document he hosts. Furthermore, the system does not hide the search access pattern, meaning Bob can observe the trapdoors in search query. Based on the encrypted file accesses after subsequent search queries, Bob also can figure out trapdoor to document ids assignments. However, Bob can not observe the plain text keyword of trapdoors.

4.3 The Proposed System

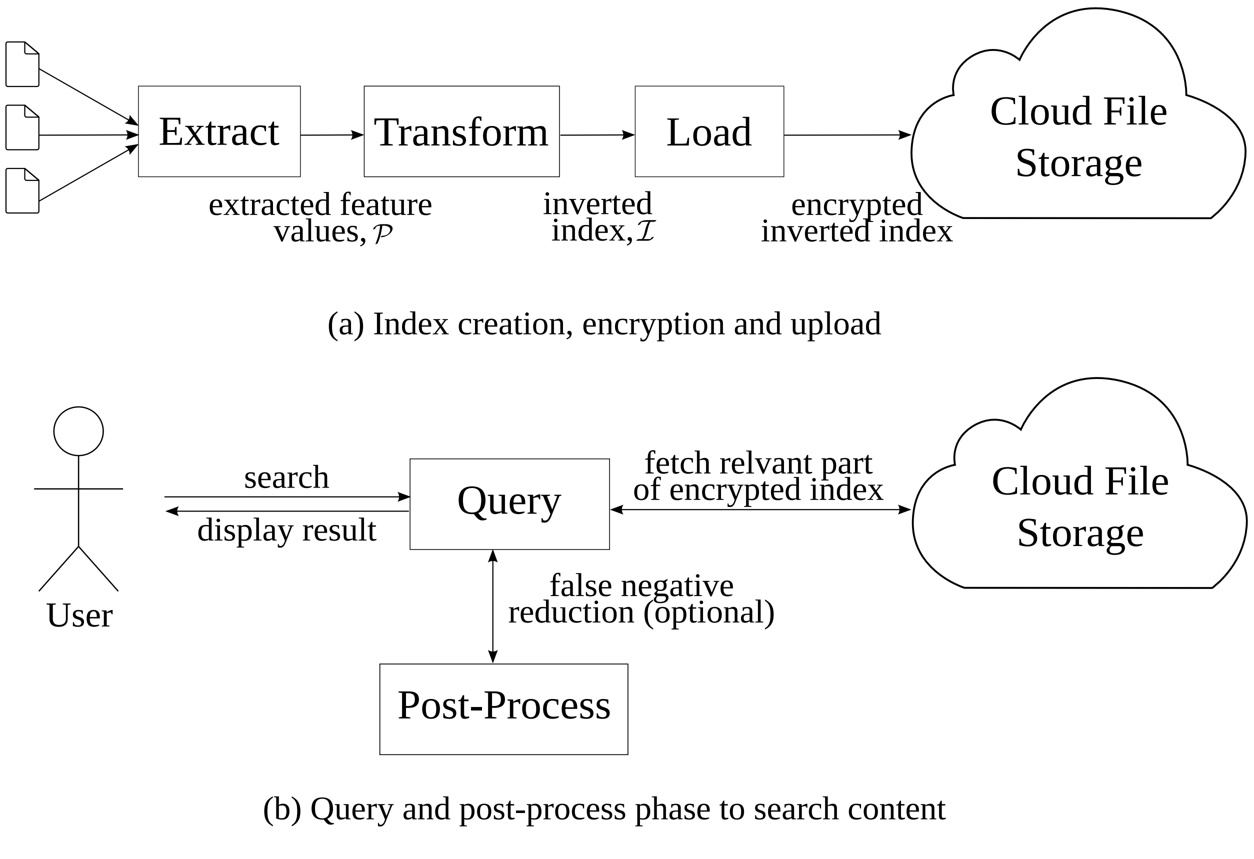
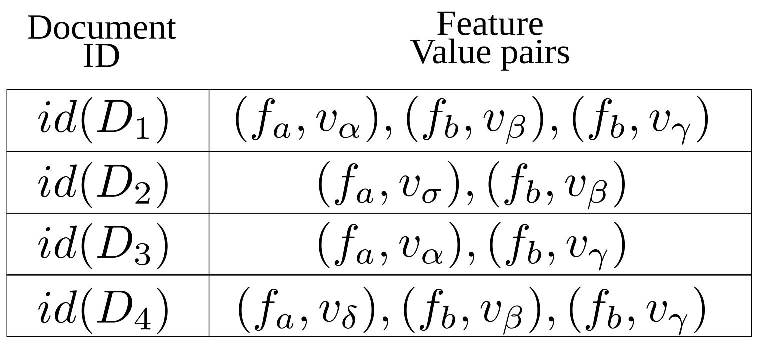

Extract
Transform
Load
Cloud File Storage
extracted feature values, $\mathcal{P}$
inverted index, $\mathcal{I}$
encrypted inverted index
(a) Index creation, encryption and upload
User
search
display result
Query
fetch relvant part of encrypted index
Cloud File Storage
false negative reduction (optional)
Post-Process
(b) Query and post-process phase to search content

Figure 4.1. Overall workflow of our proposed system. (a) Index creation consists of extract, transform and load phases. (b) Search consists of query and post-process phases.

Our main motivation is to build encrypted cloud storage that can support complex search query with support of a simple file storage server. We generalize the required computations into a five-phase *Extract, Transform, Load, Query, Post-Process (ETLQP)* framework. These five phases represent the chronological order of operations required to create, store encrypted index, and perform complex operations. Figure 4.1(a) and 4.1(b) illustrate an overview of different phases in our system.

4.3.1 Extract

In this phase we extract necessary features from a dataset. Let, $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ be a set of documents, $id(d_i)$ be the identifier of document d_i, $\Theta = \{\theta_1, \theta_2, ..., \theta_m\}$ be a set of m feature extractor functions. Functions in Θ can extract a set of feature and value pairs (f, v) from documents. We build a list U_i with all the feature value pairs extracted from d_i. For all the feature extractors $\theta_j \in \Theta$ we compute $(f, v) \leftarrow \theta_j(d_i)$ and store (f, v) in U_i. Finally we organize the result in $\mathcal{P}$, such that $\mathcal{P}[id(d_i)] \leftarrow U_i$. Such an example $\mathcal{P}$ is illustrated in Figure 4.2. Here, we have four documents $\{D_1, ..D_4\}$. D_1 has feature value pairs U_1 =$\{(f_a, v_\alpha)$,(f_b, v_β), $(f_b, v_\gamma)\}$, etc.



Document ID	Feature Value pairs
$id(D_1)$	$(f_a, v_\alpha), (f_b, v_\beta), (f_b, v_\gamma)$
$id(D_2)$	$(f_a, v_\sigma), (f_b, v_\beta)$
$id(D_3)$	$(f_a, v_\alpha), (f_b, v_\gamma)$
$id(D_4)$	$(f_a, v_\delta), (f_b, v_\beta), (f_b, v_\gamma)$

(c) Extracted feature-values, $\mathcal{P}$

Figure 4.2. $\mathcal{P}$, output of extract phase that maps document ids to feature value pairs

To clarify further, let us assume that, we want to build an encrypted image storage application that can perform location-based queries over the encrypted images. In other words, the system is capable of answering queries, such as *find images taken in Italy.* To support such a query, we implement a feature extractor function θ_l, where θ_l extracts location information from image meta data. Output of θ_l is defined as a feature value pair (*"LOCATION", "longitude and latitude of image"*). We define as many feature extractors necessary based on the application need and all feature extractor functions return values in a similar format. In Section 4.6 we discuss in detail how we defined more feature extractors and use those to answer much more complicated queries. Algorithm 1 describes the *extract* phase for building the inverted index.

Algorithm 1 Extract algorithm for building the inverted index

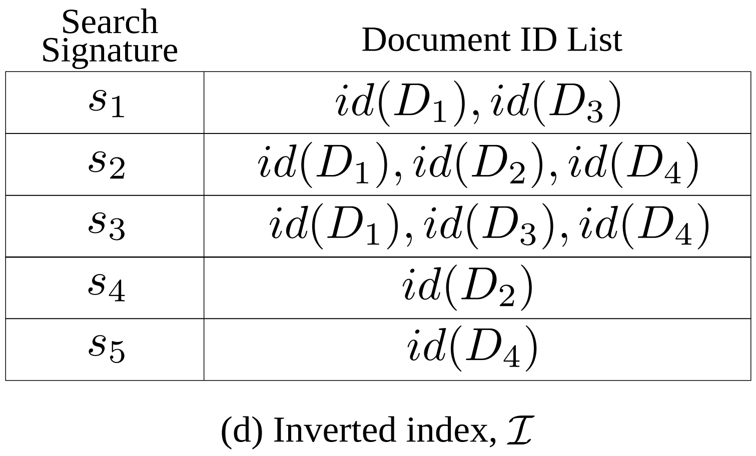
```
1:  Require: $\mathcal{D}$ = Document set, $\Theta$ = Feature extractor function set.
2:  $\mathcal{P} \leftarrow$ empty hash table.
3:  for all document $d$ in $\mathcal{D}$ do
4:      $U \leftarrow$ empty list
5:      for all feature extractor $\theta$ in $\Theta$ do
6:          $(f, v) \leftarrow \theta(d)$ and add to list $U$
7:      end for
8:      $\mathcal{P}[id(d)] \leftarrow U$
9:  end for
10: return $\mathcal{P}$
```

4.3.2 Transform

In this phase we transform the extracted feature values into a simpler form so that complex search operations can be expressed as a series of equality searches. We compute search signatures s form feature-value pairs and associate corresponding documents with s. This association at the query stage can be used to infer the existence of a feature-value pair in a document. Essentially here we define sets of transform functions $\mathcal{T} = \{t_1, .., t_p\}$, where each transform function is designed to generate search signatures from a feature value pair (f, v) and $\mathcal{T}_f$ defines subset of transformation functions that can be applied to feature f.

With these transform functions $\mathcal{T}$, we generate an inverted index $\mathcal{I}$ that is indexed by search signatures and contains the list of document ids. For all the feature value pairs in

$\mathcal{P}$, we generate search signature $s^t_{f,v} \leftarrow t(f,v)$ where $t \in \mathcal{T}_f$. We build document id list V_s for all the unique search signature s that contains $id(D_i)$ if and only if there exists a feature value pair (f,v) that is in U_i and at least one transformation function t that generates search signature s. Finally we fill the inverted index $\mathcal{I}$ such that $\mathcal{I}[s] \leftarrow V_s$. In Figure 4.3 we show such an example $\mathcal{I}$, which is created from $\mathcal{P}$ of Figure 4.2. Here, search signature s_1, s_2, s_3, s_4, s_5 are generated from feature value pairs (f_a, v_α), (f_b, v_β), (f_b, v_γ), (f_a, v_σ), (f_a, v_δ) accordingly.



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

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

Figure 4.3. Inverted index $\mathcal{I}$, that maps search signatures to document ids.

Similarly, in our encrypted image storage application example, we define a transform function t_l that takes geographic location and document id as input and converts the location information to mailing address using reverse address lookup service, takes the country information and document id to construct a search signature using a collision-resistant hash function.

Using such an extract transform model has several benefits over the ad-hoc model. The proposed model helps us to organize the necessary computation into modules, which intern increase development efficiency. The feature extractor functions can be reused in other projects. Algorithm 2 describes the *transform* phase for building the inverted index.

Algorithm 2 Transform algorithm for building the inverted index

```
1: Require: $\mathcal{P}$ = Extracted feature-value hash table, $\mathcal{T}$ = Transform function set
2: $\mathcal{I} \leftarrow$ empty hash table
3: for all document id $id(d)$ in $\mathcal{P}$ do
4:     for all feature-value pair $(f, v)$ in $\mathcal{P}[id(d)]$ do
5:         for all transformation function $t$ in $\mathcal{T}_f$ do
6:             $s \leftarrow t(f, v)$ and add $id(d)$ to $\mathcal{I}[s]$
7:         end for
8:     end for
9: end for
10: return $\mathcal{I}$
```

4.3.3 Load

In this phase, we set up our encryption schema, encrypt the inverted index, and upload the encrypted version into a file storage server $\mathcal{Z}$. We initialize a master encryption key K, three random constants C_1, C_2, C_3, a secure pseudo random permutation function φ, and a keyed pseudo random function H. Given a key, φ encrypts data, φ^{-1} decrypts corresponding result, and H generates the authentication code of messages. The pseudo-random permutation φ takes an encryption key and an arbitrary length binary string as input and outputs a ciphertext. Given output ciphertext and corresponding encryption key, the inverse φ^{-1} will output the original message back. We are also assuming that the output of φ is indistinguishable under a non-adaptive and adaptive chosen ciphertext attack (IND-CCA1, IND-CCA2). The keyed pseudo-random function H also takes an encryption key and an arbitrary length binary string as input and outputs a fixed-length binary string.

In addition, we define a small synchronized cache $\mathcal{C}$ and an encryption key K_C for encrypting the cache. $\mathcal{C}$ is always synchronized with storage server $\mathcal{Z}$. Synchronization is achieved by updating the server's version after any change in the client's version and before updating the cache locally most recent version is downloaded from the server first. In $\mathcal{C}$, we store the document id list size of all search signatures of $\mathcal{I}$, which is notated by $\mathcal{C}.freq$.

Later, we also use this cache to store information related to individual files to make the query phase easier.

We divide all the document id lists in $\mathcal{I}$ into b length blocks and add padding to the last block if needed. The value of b is determined by defining and minimizing a cost function (described in Section 4.3.6). We generate trapdoors $T_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_1)$ and $K_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_2)$ for j^{th} block of document list of $\mathcal{I}[s]$. We use K_j^s to encrypt block contents and T_j^s as the key for encrypted inverted index $\mathcal{E}$. So $\mathcal{E}[T_j^s] \leftarrow \varphi(K_j^s,\ j^{th}\ block\ of\ \mathcal{I}[s])$. To query the inverted index, later on, our system will regenerate these two trapdoors and perform inverse operations to build the original document id list. In addition, we store number of documents associated with a signature s in $\mathcal{C}.freq[s]$, then encrypt and upload the cache. Algorithm 3 describes the operations necessary for *load* phase.

Algorithm 3 Load encrypted index

```
1:  Require: $K$ = Master key, $\mathcal{I}$ = Inverted index of search signatures, $\mathcal{C}$ = Synchronized
    cache, $K_C$ = encryption key for cache, $\mathcal{Z}$ = File storage server.
2:  $b \leftarrow optimize(\mathcal{I})$
3:  for all signature $s$ in $\mathcal{I}$ do
4:      $blocks_s \leftarrow \lceil \frac{|\mathcal{I}[s]|}{b} \rceil$
5:      for $j = 1 \rightarrow blocks_s$ do
6:          $T_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_1)$, $K_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_2)$
7:          $sub \leftarrow \mathcal{I}[s].slice((j-1) \times b, j \times b)$
8:          $\mathcal{E}[T_j^s] \leftarrow \varphi(K_j^s, pad(sub))$
9:      end for
10:     $\mathcal{C}.freq[s] \leftarrow |\mathcal{I}[s]|$
11: end for
12: for all trapdoor $t$ in $\mathcal{E}$ do
13:     $\mathcal{Z}.write(t, \mathcal{E}[t])$
14: end for
15: $C_{sig} \leftarrow H(K_C \mid\mid C_3, 1)$
16: $\mathcal{Z}.write(C_{sig}, \varphi(K_C, \mathcal{C}))$
```

4.3.4 Query

In previous phases, we have created an encrypted inverted index and uploaded into file storage server $\mathcal{Z}$. Query and post-process phases are dedicated to querying the index and returning proper output to the user. First, given a user query q, we extract and transform it into a set of search signatures $\mathcal{Q}$. We use the number of document ids per block, stored in $\mathcal{C}.freq$, to compute block counts, which in turn used to compute trapdoors K_j^s and T_j^s for each block of search signatures. Using these trapdoors we retrieve and decrypt document ids. Finally, the result is organized into a hash table $\mathcal{R}$ such that $\mathcal{R}[s] = \mathcal{I}[s]$ for all $s \in \mathcal{Q}$. Algorithm 4 contains the detail operations of the query phase.

Algorithm 4 Query

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

4.3.5 Post-process

In this step, we further process the result of the query phase to remove false positive entries. Given the result set $\mathcal{R}$ from the query phase for query q, we remove the id of the documents that do not match the original query. Therefore, $\mathcal{R}.remove(id(d))$ if $q(d) \neq True$. A query that only contains exact search features, this phase is optional.

4.3.6 Optimal block size analysis

Block size has a direct impact on the performance of our proposed system. Larger block size implies a waste of space for padding and smaller block size implies many blocks to process. So we need to find an optimal value of block size b that keeps the overall cost to minimal. In our construction for each block, we have a fixed cost and a dynamic cost that is related to block length. We define fixed cost as α and co-efficient of dynamic cost β. The cost can be in terms of time and size. Both linearly depends on block size in our construction. So cost for a b length block is $(\alpha + \beta \times b)$. Let, $\mathcal{J}(s)$ is $|\mathcal{I}[s]|$ meaning document id list size for search signature s and total cost $\mathcal{G}(b)$ for blocking and encrypting given inverted index $\mathcal{I}$ for block length b then

$$\mathcal{G}(b) = \sum_{s \in \mathcal{I}} \left\lceil \frac{\mathcal{J}(s)}{b} \right\rceil (\alpha + \beta \times b)$$

We want to minimize the above function for b. However, it contains a ceiling function, which can not be minimize by taking derivatives and equating to zero. So we approximate the probability distribution of $\mathcal{J}$, i.e., lengths of document id list in $\mathcal{I}$. We assumed that, distribution is Pareto distribution (Arnold, 1985), which is defined by probability density function (PDF)

$$f(x|\gamma, x_m) = \frac{\gamma x_m^\gamma}{x^{(\gamma+1)}}$$

and cumulative distribution function (CDF)

$$F(x|\gamma, x_m) = 1 - (\frac{x_m}{x})^\gamma$$

where x is the random variable, γ is distribution parameter, and x_m is minimum value of x.

In our total cost analysis for each $\mathcal{J}(s)$ smaller or equal b cost is exactly $(\alpha + \beta \times b)$ and number of elements where $\mathcal{J}(s) \leq b$ is equal to $F(b)$. For elements where $\mathcal{J}(s) > b$ we can approximate the total cost using expected value of $\mathcal{J}(s)$. Finally, the cost function

$$\mathcal{G}(b) = (\alpha + \beta b)F(b) + E[\mathcal{J}(s)]_{\mathcal{J}(s)>b}(\frac{\alpha + \beta b}{b})$$

where E is expectation of probability distribution. Now we can compute the expectation by integration.

$$\mathcal{G}(b) = (\alpha + \beta b)(1 - (\frac{x_m}{b})^{\gamma}) + (\frac{\alpha + \beta b}{b}) \int_b^{\infty} \frac{\gamma x_m^{\gamma} x}{x^{\gamma+1}} dx$$

After preforming integration and several algebraic simplification we get the final form

$$\mathcal{G}(b) = (\alpha + \beta b) - (\alpha + \beta b) x_m^{\gamma} b^{-\gamma} + (\gamma x_m^{\gamma} \frac{b^{-\gamma} + 1}{\gamma - 1})(\frac{\alpha}{b} + \beta)$$

And the first order derivative is

$$\mathcal{G}'(b) = \beta - x_m^{\gamma} \beta b^{-\gamma} + (\alpha + \beta b) x_m^{\gamma} \gamma b^{-\gamma - 1} - \gamma x_m^{\gamma} b^{-\gamma} (\frac{\alpha}{b} + \beta) - \frac{\gamma x_m^{\gamma}}{\gamma - 1} b^{-\gamma - 1} \alpha$$

Now we minimize b by setting $\mathcal{G}'(b) = 0$ and solving the equation for b. In experimentation, we observe that method of moments estimation for x_m and γ gives almost the correct value.

4.4 Dynamic Document Addition

Here we are going to improve our algorithms to support the dynamic addition of documents. Given a new document set D' for addition, we first perform extract and transform to build an inverted index $\mathcal{I}'$. Now we download and decrypt the cache $\mathcal{C}$ and compute the number of blocks x, the number of empty spaces in the last block y from $\mathcal{C}.freq$ information for signatures that are already in inverted index $\mathcal{I}$. On the other hand, assign zero to x and y for search signatures that we have not seen yet. If there is empty space meaning $y > 0$ then we fill the last block with new document ids. The rest of the document ids are divided into b length blocks and encrypted with the appropriate key. Algorithm 5 describes dynamic document addition in detail.

Algorithm 5 Dynamic document addition

```
1: Require: $D'$ = Documents to add, $K$ = Master key, $C_1, C_2, C_3$ = Constants, $b$ = block size, $K_C$ = Encryption key for cache, $Z$ = File storage server, $\Theta$ = Feature extractor function set, $\mathcal{T}$ = Transform function set.
2: $\mathcal{I}' \leftarrow Transform(Extract(D', \Theta), \mathcal{T})$
3: $C_{sig} \leftarrow H(K_C \mid\mid C_3, 1)$
4: $\mathcal{C} \leftarrow \varphi^{-1}(K_C, \mathcal{Z}.read(C_{sig}))$ // download and decrypt
5: for all signature $s$ in $\mathcal{I}'$ do
6:     if $s$ in $\mathcal{C}.freq$ then
7:         $x \leftarrow \lceil \frac{\mathcal{C}.freq[s]}{b} \rceil$, $y \leftarrow x \times b - \mathcal{C}.freq[s]$
8:     else $x \leftarrow 0$, $y \leftarrow 0$
9:     end if
10:    if $y > 0$ then
11:        $T^s_x \leftarrow H(K, s \mid\mid x \mid\mid C_1)$, $K^s_x \leftarrow H(K, s \mid\mid x \mid\mid C_2)$
12:        $L \leftarrow \varphi^{-1}(K^s_x, \mathcal{Z}.read(T^s_x))$
13:        Fill empty spaces in $L$
14:        $\mathcal{Z}.write(T^s_x, \varphi(K^s_x, L))$
15:    end if
16:    for $j = 1 \rightarrow \lceil \frac{|\mathcal{I}'[s]| - y}{b} \rceil$ do
17:        $k \leftarrow j + x$
18:        $T^s_k \leftarrow H(K, s \mid\mid k \mid\mid C_1)$, $K^s_k \leftarrow H(K, s \mid\mid k \mid\mid C_2)$
19:        $sub \leftarrow \mathcal{I}'[s].slice((k-1) \times b + y, j \times b + y)$
20:        $\mathcal{Z}.write(T^s_k, \varphi(K^s_k, pad(sub)))$
21:    end for
22:    $\mathcal{C}.freq[s] \leftarrow \mathcal{C}.freq[s] + |\mathcal{I}'[s]|$
23:    $\mathcal{Z}.write(C_{sig}, \varphi(K_C, \mathcal{C}))$ // encrypt and upload
24: end for
```

4.4.1 Bandwidth Requirement Analysis

One might argue that, since we are performing all the complex operations on client side, so encrypt the inverted index $\mathcal{I}$ like another document; then download, decrypt, and search in the local inverted index in time of query to avoid all the complexities. However, such an approach will increase bandwidth consumption for dynamically updating the index.

Let, $\{q_1,...,q_\varrho\}$ be ϱ consecutive queries that user like to perform on a dynamically updating index, (i.e., new documents are added in between each query), $|q_i|$ be the length of query q_i, $|\mathcal{E}(q_i)|$ be the size of blocks returned by query q_i, $|\mathcal{Y}|$ be the maximum among $\{|\mathcal{E}(q_1)|$,...,

$|\mathcal{E}(q_{\varrho})|\}$, $|\mathcal{I}|$ be the size of inverted index, $|\mathcal{C}.freq|$ be the size of cache required for storing frequency of all the buckets. Total bandwidth cost for performing n queries ignoring the addition cost

$$\varrho|\mathcal{C}.freq| + \sum_{i=1}^{\varrho}(|\mathcal{E}(q_i)| + |q_i|) \leq \varrho(|\mathcal{C}.freq| + |\mathcal{Y}| + |q_i|)$$

On the other hand, if we keep a local inverted index the bandwidth cost would be simply $\varrho|\mathcal{I}|$. Since after each update index is updated and we need to download the recent version. In practice $|\mathcal{C}.freq| + |\mathcal{Y}| + |q_i| \ll |\mathcal{I}|$. Also if we consider the cost of addition operation, our system will outperform. Because during addition we are only adding new blocks not updating the whole index. In contrast, the complete local inverted index needs to be sent to the server after each addition. So building an encrypted inverted index always saves bandwidth. However, the amount of savings depends on the dataset and query load.

4.5 Security

In this section, we formally prove the security of our proposed system. The cloud service is managed by semi-honest Bob, who follows the defined protocol but may try to infer private information about the document he hosts. Over the years, many security definitions have been proposed for searchable encryption for the semi-honest model. Among those simulation-based adaptive semantic security definition by Curtmola, at el., (Curtmola et al., 2006) is widely used in literature. Later it is customized to work under a random oracle model by Kamara, at el., in (Kamara et al., 2012). We adopt this definition to prove our security model.

In our proposed static model, we are leaking encrypted document size, block length, number of total blocks, trapdoor of blocks related to a search query information. In the dynamic model, in addition to this information, we are leaking the length of newly added encrypted documents, and associated search signatures. Also, note that the cache $\mathcal{C}$ can be considered as a document that is updated with a new length in every addition operation. This *does not leak any additional information* because in the cache we are storing (1) document id lists length information, which is some constant times number of search signatures and (2) few internal information about documents, which is some constant times number of documents. All of these atomic information are already leaked due to the index.

We will first define necessary patterns, history, trace, and view for our schema then prove this schema satisfies adaptive semantic security.

Search Signature Pattern (μ_p): Suppose $\{o_1, o_2, ..., o_\eta\}$ is a set of η consecutive operations on the encryption collection such that o_i is a search or addition request. Each operation o_i has a set of associated search signatures denoted as o_i^s. Specifically, if o_i is a search instance, it involves a single search signature $o_i^s = \{s_{i_1}\}$, If o_i is an addition, it involves a set of search signatures that are included in the whole dataset of new documents such that $o_i^s = \{s_{i_1}, ..., s_{i_\varsigma}\}$. Then μ_p is a function such that $\mu_p((i, \rho), (j, \ell)) = 1$ if $s_{i_\rho} = s_{j_\ell}$ and $\mu_p((i, \rho), (j, \ell)) = 0$ otherwise, for $1 \leq i, j \leq \eta$, $1 \leq \rho \leq |o_i^s|$, and $1 \leq \ell \leq |o_j^s|$

Search pattern ($\mathcal{N}_p$): Suppose o_i is a search request, $cnt(s_{i_1})$ be the number of times s_{i_1} occurs in the dataset. Then, $\mathcal{N}_p(o_i) = (cnt(s_{i_1}))$. Note that we are assuming that the adversary can infer this count. However, we are not disclosing this information directly.

Addition Pattern ($\mathcal{A}_p$): Suppose o_i is an addition request for a document collection $\{D_\iota, ..., D_\rho\}$, $|C_x|$ denotes the bit-length for the encrypted form of D_x, $\{s_{j_1}, ..., s_{j_\varsigma}\}$ is set of search signatures that are included in a new corpus, and $cnt(s_{j_\iota})$ denotes the number of documents associated with s_{j_ι} in modified dataset. Then $\mathcal{A}_p(o_j) = (\{|C_\iota|, ..., |C_\rho|\}, \{cnt(s_{j_1}), ..., cnt(s_{j_\varsigma})\})$

History ($\mathcal{H}_\eta$): Let $\mathcal{D}$ be the document collection and $OP = \{o_1, ..., o_\eta\}$ be the consecutive search or addition requests that are issued by user. Then $\mathcal{H}_\eta = (\mathcal{D}, OP)$ is defined as η query history.

Trace (λ): Let $C = \{C_1, ..., C_n\}$ be the collection of encrypted data items, $|C_i|$ be the size of C_i, $\mu_p(\mathcal{H}_\eta)$, $\mathcal{N}_p(\mathcal{H}_\eta)$, $\mathcal{A}_p(\mathcal{H}_\eta)$, b be the search signature, search, addition pattern for

$\mathcal{H}_\eta$, length of each block in encrypted inverted index respectively. Then $\lambda(\mathcal{H}_\eta) = \{(|C_1|, ..., |C_n|), \mu_p(\mathcal{H}_\eta), \mathcal{A}_p(\mathcal{H}_\eta), \mathcal{N}_p(\mathcal{H}_\eta), b\}$ is defined as the trace of $\mathcal{H}_\eta$. Trace can be considered as the maximum amount of information that a data owner allows its leakage to an adversary.

View (v): Let $C = \{C_1, ..., C_n\}$ be the collection of encrypted data items, $\mathcal{E}$ be the encrypted inverted index, and $\Pi = \{\Pi_{o_1}, ..., \Pi_{o_\eta}\}$ be the trapdoors and encrypted values for η consecutive requests in $\mathcal{H}_\eta$. Then, $v(\mathcal{H}_\eta) = \{C, \mathcal{E}, \Pi\}$ is defined as the view of $\mathcal{H}_\eta$. The view is the information that is accessible to an adversary.

Adaptive Semantic Security for Dynamic SSE: SSE schema satisfies adaptive semantic security in random oracle model, if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ that can adaptively simulate the adversary's view of the history from the trace with probability negligibly close to 1 through interaction with random oracle. Intuitively, this definition implies that all the information that is accessible to the adversary can be constructed from the trace. Formally, let $\mathcal{H}_\eta$ be a random history from all possible history, $v(\mathcal{H}_\eta)$ be the view, $\lambda(\mathcal{H}_\eta)$ be the trace of $\mathcal{H}_\eta$. Then, scheme satisfies the security definition in the random oracle model if one can define a simulator $\mathcal{S}$ such that for all the polynomial-size distinguishers $Dist$, for all polynomial $ploy$ and a large Λ:

$$Pr[Dist(v(\mathcal{H}_\eta)) = 1] - Pr[Dist(\mathcal{S}(\lambda(\mathcal{H}_\eta))) = 1] < \frac{1}{poly(\Lambda)}$$

where probabilities are taken over $\mathcal{H}_\eta$ and the internal coins of key generation and encryption.

Theorem 1. *The proposed scheme satisfies the adaptive semantic security.*

Proof. We will show the existence of 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 $v_R(\mathcal{H}_\eta) = \{C, \mathcal{E}, \Pi\}$ be the real view. Then $\mathcal{S}$ adaptively generates the simulated view $v_S = \{C^*, \mathcal{E}^*, \Pi^*\}$

$\mathcal{S}$ chooses n random values $\{C_1^*, ..., C_n^*\}$ such that $|C_1^*| = |C_1|$, ..., $|C_n^*| = |C_n|$. In this setting, C_i is the output of a secure encryption scheme. By the pseudo-randomness of the applied encryption, C_i is computationally indistinguishable from C_i^*.

Given the documents per search signature (e.g., $cnt(s)$) and block length b, $\mathcal{S}$ computes number of entries in $\mathcal{E}$ and generates that many (k_i^*, v_i^*). Note that for every (k_i, v_i) in real encrypted inverted index $\mathcal{E}$ there exists a (k_i^*, v_i^*) in simulated encrypted inverted index $\mathcal{E}^*$. Here length of k_i and k_i^* is equal to the output length of pseudo-random function H. Similarly, length of v_i and v_i^* is equal to b. Here, encrypted keys and blocks are computationally indistinguishable from random values by pseudo-randomness of the applied encryption.

$\mathcal{S}$ simulates requests $\Pi_{o_1}, ..., \Pi_{o_\eta}$ according to their types

1) Π_{o_i} is a search request: We define $\mathcal{X}_s = \{\pi_{s^1}, ..., \pi_{s^{\lceil \frac{cnt(s)}{b} \rceil}}\}$ be the trapdoors generated for search signature s. Suppose, $\Pi_{o_i} = (\mathcal{X}_{s_{i_1}}, cnt(s_{i_1}))$ is a search request. Then $\mathcal{S}$ copies $cnt(s_{i_1})$ from $\mathcal{N}_p(o_i)$ to $cnt(s_{i_1})^*$. Then if $\mu_p((i,1), (j,\ell)) = 1$ for any $1 \leq j < i$ and $1 \leq \ell \leq |o_j|$ then $\mathcal{X}_{s_{i_1}}^* = \mathcal{X}_{s_{j_\ell}}^*$. Otherwise $\mathcal{X}_{s_{i_1}}^*$ is set to $\lceil \frac{cnt(s_{i_1})}{b} \rceil$ number of random row-key from simulated encrypted inverted index $\mathcal{E}^*$ such that those was not previously selected during the simulation. In this setting, components of simulated and real requests are computationally indistinguishable by the pseudo-randomness of the applied encryption. Hence Π_{o_i} and $\Pi_{o_i}^*$ are computationally indistinguishable.

2) Π_{o_i} is an addition request: Suppose, $\Pi_{o_i} = ((\mathcal{X}_{s_{i_1}}, cnt(s_{i_1})), ..., (\mathcal{X}_{s_{i_\varsigma}}, cnt(s_{i_\varsigma})))$ is an addition pattern, $|\Pi_{o_i}|$ be the number of pairs.

For each of the pair individually (iterated with ρ, where $1 \leq \rho \leq |o_i^s|$) simulator $\mathcal{S}$ does the following. First copy $cnt(s_{i_\rho})$ from $\mathcal{A}_p(o_i)$ to $cnt(s_{i_\rho})^*$. Next, if $\mu_p((i,\rho),(j,\ell)) = 0$, for all $1 \leq j < i$ and $1 \leq \ell \leq |o_j^s|$ meaning new search signature so copy $\lceil \frac{cnt(s_{i_\rho})}{b} \rceil$ new random row keys and values from $\mathcal{E}^*$ to $\mathcal{X}_{s_{i_\rho}}^*$ such that those was not used earlier. However, things get little complicated when there is at least one $\mu_p((i,\rho),(j,\ell)) = 1$ meaning this search signature has been seen earlier. Note that during addition phase client only needs to update the last block and add more blocks if necessary. For this simulator $\mathcal{S}$ needs to search in revers find the largest j that is smaller than i where $\mu_p((i,\rho),(j,\ell)) = 1$ That is the place where s_{i_ρ} was last used. Now find $cnt(s_{j_\ell})$ either from $\mathcal{A}_p(o_j)$ or $\mathcal{N}_p(o_j)$ depending on the j^{th} operation. Now $cnt(s_{i_\rho}) - cnt(s_{j_\ell})$ is the number of new documents that have search signature s_{i_ρ} and $\mathcal{S}$ assigns $\lfloor \frac{cnt(s_{i_\rho})-cnt(s_{j_\ell})}{b} \rfloor$ new k^* from $\mathcal{E}*$ that are not already used and corresponding random v^*. Also $\mathcal{S}$ has to add one more row key-value pair (k^*, v^*) to for the last block that's being updated. $\mathcal{S}$ picks last element of $\mathcal{X}^*_{s_{j_\ell}}$ from so far generated Π^* and randomly generate a new value $v*$ for that element too. In this setting, the constructed $\Pi^*_{o_i}$ is computationally indistinguishable from Π_{o_i}.

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

□

4.6 Application of ETLQP framework

As an application of our ETLQP framework, we built an image storage system that saves encrypted images in cloud storage and built an encrypted index to search later on. Before going into further detail of our ETLQP framework implementation we briefly describe Fuzzy Color and Texture Histogram (FCTH) (Chatzichristofis and Boutalis, 2008), Eigenface (Turk and Pentland, 1991), Locality Sensitive Hashing(LSH) (Indyk and Motwani, 1998), and range query to exact query conversion mechanism (Faber et al., 2015). FCTH and Eigenface are used for image similarity search and face recognition respectively and LSH is used for dimension reduction. Finally, as the name suggests range query to an exact query conversion mechanism is used to convert a range query in a defined range to a sequence of matching query. These concepts are vital to the development of our system.

Fuzzy Color Texture Histogram (FCTH) (Chatzichristofis and Boutalis, 2008) is an histogram of image that combines texture and color information. It is widely used in content-based image retrieval systems (CBIR) (Lux and Chatzichristofis, 2008; Chatzichristofis et al., 2009; Yang et al., 2009; Chatzichristofis et al., 2010; Chatzichristofis and Boutalis, 2010). In FCTH the texture information is represented by an eight-bin histogram derived via the fuzzy system that uses the high-frequency bands of the Haar Wavelet transform. The color is represented by a 24-bin color histogram computed like in the CEDD descriptor. Overall, the final histogram includes 192 regions. Each of the 1600 image blocks is processed and assigned to a region as in the CEDD. The final 192-bin histogram is also normalized and quantized such that each bin value is an integer between 0 to 7 inclusive. FCTH of an image can be considered as a vector with 192 dimensions and distance between FCTH vector of images can be used to determine similarity among images.

Eigenface (Turk and Pentland, 1991) is a very well studied, effective yet simple technique for face recognition using static 2D face image. It consists of three major operations - finding eigenvectors of faces, finding weights of each faces, and recognition tasks.

Finding Eigenvectors. We start with M face-centered upright frontal images that are represented as $N \times N$ square matrices. Let, $\{\Gamma_1, \dots, \Gamma_M\}$ are $N^2 \times 1$ vector representation of these square matrices, $\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$ is the average of these vectors, and $\Phi_i = \Gamma_i - \Psi$ is computed by subtracting average Ψ from ith image vector.

Now eigenvectors u_i of co-variance matrix $C = AA^T$, where $A = [\Phi_1\ \Phi_2 \dots \Phi_M]$, can be used to approximate the faces. However, there are N^2 eigenvectors for C. In practice N^2 can be a very large number, thus computing eigenvectors of C can be very difficult. So instead of AA^T matrix we compute eigenvectors of A^TA and take top K vectors for approximating eigenvectors u_i, where $\left\Vert u_i \right\Vert = 1$. The selection of these eigenvectors is done *heuristically.*

Finding Weights. Φ_i can be represented as a linear combination of these eigenvectors $\Phi_i = \sum_{j=1}^{K} w_j u_j$ and weights can be calculated as $w_j = u_j^T \Phi_i$. Each normalized image is represented in this basis as a vector $\Omega_i = \begin{bmatrix} w_1 \ w_2 \ \ldots \ w_k \end{bmatrix}^T$ for $i = 1, 2, \ldots M$. This is essentially projecting face images into new eigenspace (the collection of eigenvectors).

Recognition Task. Given a probe image Γ, we first normalize $\Phi = \Gamma - \Psi$ then project into eigenspace such that $\Omega = \begin{bmatrix} w_1 \ w_2 \ \ldots \ w_K \end{bmatrix}^T$, where $w_i = u_i^T \Phi$. Now we need to find out nearest faces in this eigenspace by $e_r = min \left\| \Omega - \Omega_i \right\|$. If $e_r <$ a threshold, chosen heuristically, then we can say that the probe image is recognized as the image with which it gives the lowest score.

In summary, face images are considered as a point in a high dimensional space. An eigenspace consisting few significant eigen vectors are computed for approximating faces in a training face dataset. Next, test face images are projected into the computed eigenspace. Distances of test face images and all training faces images are computed. If any distance is bellow a pre-determined threshold then those faces are considered a match for associated test face.

Locality sensitive hashing is a technique widely used to reduce dimensions. The core concept of LSH is to define a family of hash functions such that similar items belong to same bucket with high probability. More specifically we utilized LSH in euclidean space and adopted widely accepted projection over the random line technique described in (Andoni and Indyk, 2008). Let, r be a random projection vectors, v be an input vector, o be a random number used as offset, and w be bucket length parameter fixed by user. The bucket id is computed by $Round(\frac{v.r+o}{w})$ function. Finally, several such projection vectors are used to generate several bucket ids for a single input vector. In this setting, nearby items will share at least the same bucket with very high probability. In practice value of w and the number of random projections are controlled to achieve the required success rate.

Range query to exact query conversion. We adopt the range query mechanism described in (Faber et al., 2015). Let, a be a discrete feature that has a value ranging from 0 to 2^{t-1}, meaning it requires t bits to represent in binary. We first create a binary tree of t depth representing the complete range. Each leaf node (at depth t) represents an element in the range and we level all left edge as 0 and right edge as 1. So, the path from the root to a leaf node essentially represent the binary encoding of that leaf. In the transform phase, we convert an input value of the range to t feature-value tuples, where the feature is the concatenation of field name, depth i and value is the binary encoding of inner node at depth i. During the query phase given a range, we first find the cover as described in (Faber et al., 2015), create the corresponding search signatures and perform the query.

4.6.1 ETLQP for image storage

To build an application using ETLQP framework described system section, the programmer has to define proper extract and transformation functions. Load, Query, and Post-Process phases remain the same. For our image storage software we consider four features *location* - where the picture was take, *time* - when the picture was taken, *texture and color* - for searching similar pictures, and *faces* - for face recognition. In our implemented system queries of the first two features are equality search and later two are similarity search. Similarity searches are difficult to perform since result not only contains exact matches but also contains results that are similar. So, we need to have a similarity measure for the feature in question. To accomplish such a similarity query we utilize LSH, which essentially helps us to convert the query to a sequence of equality search. In addition, results of the LSH can contain false positives. We need extra post-processing to remove those.

Extract. Location and time data are extracted from Exif (JEITA, 2002) meta-data. Exif is a very popular standard for attaching image meta-data into image used by all popular camera manufacturers. Camera with Global Positioning System (GPS) module can store longitude and latitude of a picture taken into Exif data, which can be extracted easily using available libraries (Noakes, 2019). We use FCTH for similarity analysis and used open-source implementation of the FCTH analyzer (Lux and Chatzichristofis, 2008). Finally, for face recognition using Eigenface, we extract frontal faces from images using haar cascade (Viola and Jones, 2001; Lienhart and Maydt, 2002) frontal face pattern classifier.

Transform. Now we define appropriate transformations for extracted features. The main idea behind the definition of transformation functions is to make the query easier later on. So definition of transformation functions is mainly guided by the query demand.

- **Location.** Location information in terms of longitude and latitude is difficult to use in practice. We use OpenStreetMap's reverse geolocation service (OpenStreetMap, 2019) to determine the address of latitude and longitude associated with the image. To make queries easier later, we generate search signatures of six sub-features of the address - full address, city, county, country, state, and zip.

- **Time.** Similarly, we break the created date of an image into five sub-features - complete date, year, month, day of month, and day of the week. We generate search signatures based on these sub-features. In addition, to support range query based on date we convert the time into Unix timestamp that essentially represents seconds passed from 1 *January* 1970 without considering the leap second. Then we divide the time stamp by the number of seconds in a day (86400), which gives us the number of days passed from the epoch. Finally, we build the range query binary tree with depth 20, which essentially is capable of covering dates till year 4840. Then we create the feature value list as described earlier.

- **Texture and Color.** In the extract phase we extracted FCTH of the provided image, which is a 192-dimensional vector. We can treat each dimension as different sub-features but that will make it difficult to perform similarity search later on. Instead, we define a euclidean LSH schema that puts near elements into the same bucket and use the bucket ids to generate search signatures.

- **Face.** We built an eigenface schema with extracted face images. Again to preserve similarity we built a euclidean LSH schema with weight vectors of faces and store the eigenspace related information into synchronized cache $\mathcal{C}$. In particular, we store the average face, selected top eigenfaces, and weights of all faces. Storing such information is the major reason for defining the cache $\mathcal{C}$.

Query and Post-Process. With previously defined extract and transform functions, we can perform *time queries*, such as find images that are taken on specific year, month, day of the week, day of the moth, or in a range of dates, etc. We can also perform *location queries*, such as find pictures that were taken in a country, state, city, etc. In both of these cases, we transform a query into encrypted search signatures and retrieve associated encrypted document ids from the cloud storage server. Finally, we decrypt and display the result directly to the user. On the other hand, for face recognition and image-similarity query, we extract appropriate feature values from a query image and transform these values into LSH bucket ids of previously defined LSH schema. We generate encrypted search signature, retrieve encrypted document ids, and decrypt the result like date and time queries. However, before showing results to user we remove false-positive results introduced by the LSH schema.

4.7 Experimental Evaluation

In our proposed design we have two components *client* and *server*. Client processes images, performs cryptographic operations, and produces an encrypted inverted index that is stored in the server. In query phase client retrieves partial index from the server based on the user query.

ETQLP client is written in Java using several other libraries for image feature extraction. Cryptographic operations are performed using the Java Cryptographic Extension (JCE) implementation. During our experimentation, we execute the client program in a computer with *Intel(R) Core(TM) i7-4770 3.40GHz* CPU, *16GB* RAM running *Ubuntu 14.04.4 LTS*. Our implemented client can store encrypted inverted index into different types of servers.

- File storage server in local network. We developed a very simple web-based storage service that has two endpoints file read and file write. Our server is written in Python (v2.7.6) using Flask (v0.10.1) microframework and files are stored in a MongoDB (v3.2.0). We deployed our local storage server in a machine with *Intel(R) Xeon(R) CPU E5420 2.50GHz* CPU, *30GB* of RAM running *CentOS 6.4*. In addition, our client computer is also in the same network. Here, file path is search signature of encrypted inverted index $\mathcal{E}$ and file content is encrypted document id list.

- Amazon S3 (Amazon, 2019) is a very popular commercial object and file storage system, which provides easy to use representational state transfer (REST) application program interface (API) for storing, retrieving and managing arbitrary binary data or file. Amazon also provides a very extensive software development kit (SDK) for building applications to utilize its services. In our implementation, search signatures of encrypted inverted index $\mathcal{E}$ are keys of S3 objects and content of the objects are associated encrypted document id list.

- Personal file storage services. Among the popular commercial personal file storage services, we implemented capabilities to store inverted index into Dropbox (Dropbox, 2019a), Box (Box, 2019a), and Google Drive (Google, 2019a) because of available open-source SKD on these platforms. Here, each entry in encrypted inverted index $\mathcal{E}$ is saved as separate file, where file name is encrypted search signature and file content is encrypted document id list. *Due to rate limitations* (Dropbox, 2019b; Box, 2019b; Google, 2019b) *we could not perform extensive analysis on these platforms.* However, a typical user adding images from time to time will not have any trouble using any of these platforms as a cloud file storage server. We reached the rate limit due to the repeated nature of our experiment.

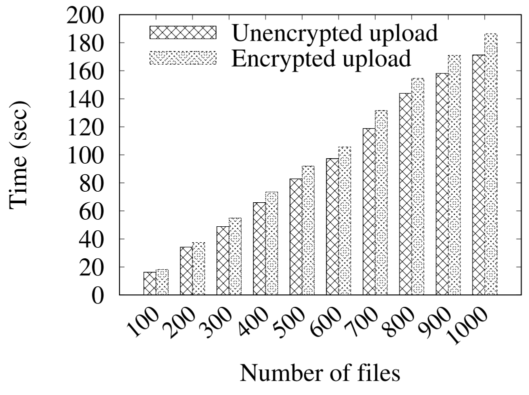
In Table 4.1, we list the throughput of each of the servers. We compute the system throughput by upload and downloading 100MB in the storage servers. We observe that local server performs extremely well in case of download because of MongoDB's advanced cache management, which keeps the recently used data in RAM to improved performance. In addition, in our smaller-scale experiments we observed that the performance of personal storage server scales according to this throughput ratio.

Dataset. We use Yahoo Flickr Creative Commons 100 Million Dataset (YFCC100M) presented in (Thomee et al., 2015), which contains basic information of 100 million media objects, of which approximately 99.2 million are photos and 0.8 million are videos, all of which carry permissive creative commons license. We have randomly selected 20109 images and downloaded the original version of these images. Size of this random dataset is 42.3GB, average file size is 2.15MB, the number of faces detected 7027, and 4102 images have latitude and longitude embedded in EXIF data.

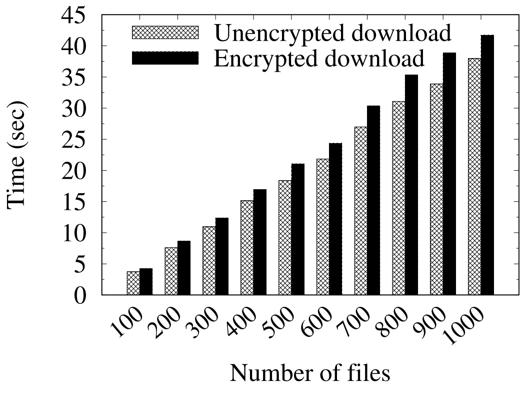
Face Detection Accuracy. Our constructed dataset is randomly selected and unlabeled. As a consequence the correctness of our face detection system remains unmeasured. So we perform face detection on two know face datasets *Caltech Faces* (Weber, Weber) and *Color FERET* (Phillips et al., 1998, 2000). *Caltech Faces* dataset contains 450 frontal face images each containing picture of an individual. Our system detected 431 of those correctly, yielding a 95.78% accuracy. We also observed that most of the failed images are too dark to detect any face. *Color FERET* dataset contains a total of 11338 facial images, which were collected by photographing 994 subjects at various angles. Since our face detection system

Table 4.1. Throughput of different servers

Data Set	Download throughput	Upload throughput
Local	66.049	13.589
Amazon-S3	17.096	20.646



(a) Overhead Upload



(b) Overhead Download

Figure 4.4. Overhead of encryption and decryption during upload and download.

detects frontal faces only, we extract frontal face images with `fa` and `fb` suffixes. We found that there are a total 2722 such images. Our face detection system successfully detected 2459 images, yielding 90.33% accuracy. Almost all the failed images are too dark or subjects wore glasses.

Experiments. Before performing any experiment for the proposed ETLQP framework, we first compare the performance of an encrypted image storage system with an unencrypted one to observe the overhead of encryption. We randomly select a few images, encrypt and upload those images. Then we download, decrypt and save those files again and measure the performance. We perform this experiment with local storage server and in the client we used a thread pool with 2 threads to parallelize the operations. Encrypted files are slightly larger than the unencrypted version because we padded the input file and added a 256-bit message authentication code. So overall size overhead is very negligible. We observe on average 10.09% increment in execution time for encrypted upload compare to unencrypted upload, illustrated in 4.4(a). Similarly we observe on average 13.06% increment in execution time for downloading encrypted file and decrypt, compare to unencrypted download, illustrated

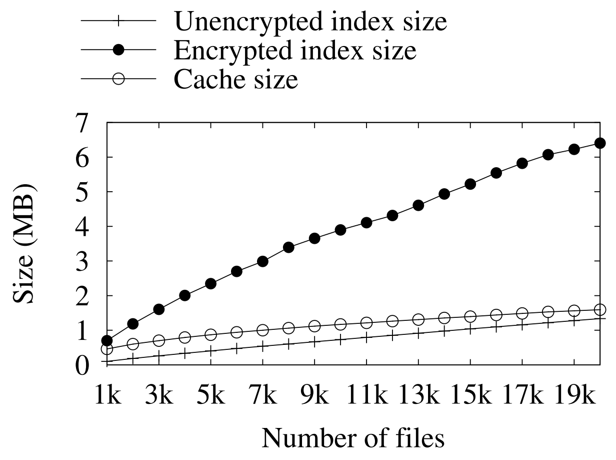
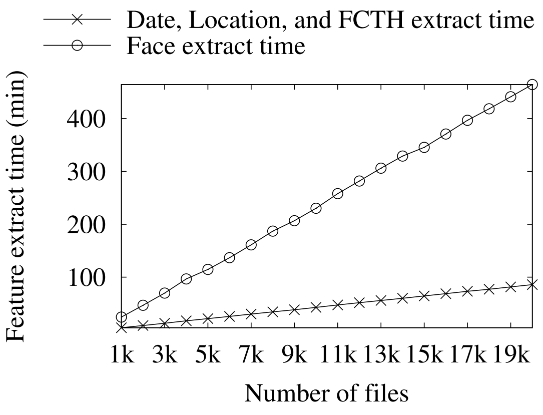


Figure 4.5. Size of the unencrypted index, encrypted index, and cache vs. number of files

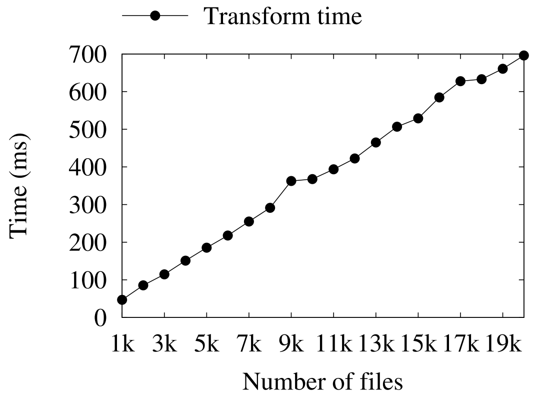
in 4.4(b). So we conclude that encryption *does not* add significant overhead for an efficiently implemented client.

Now, we measure the performance of different phases of our framework for varying the number of randomly selected images from the above dataset. We measure the performance of different phases of our framework for varying numbers of randomly selected images from the above dataset. The horizontal axis of most of the reported graphs is the number of randomly selected images used to build the index and the vertical axis is the observation. We repeat each experiment for at least 3 times and report the average observation value.

We extracted four features of the images *created date*, *location*, *FCTH vector*, and *faces*. For *created date* feature we generated search signatures of *day of the week*, *day of the month*, *month*, *year*, *week of week year*, and a combination of *year, month, and date.* Also we generated range query related signature to perform arbitrary range query on date. For *location* feature, we first reverse looked up the address of latitude, longitude extracted from images using open street map (OpenStreetMap, 2019)Next we created search signatures based on *city*, *state*, *county*, *country*, *zip code* and *full address. FCTH vector* is extracted from all the images with Lucene image retrieval (Lux and Chatzichristofis, 2008) implementation. We



(a) Extract time



(b) Transform time

Figure 4.6. Time required for building the index for different number of images.

detected *frontal faces* using OpenCV implementation of haar cascade classifier, converted all the face images to median face size, built an eigenface classifier on the detected faces, and store the computed weight vector of all the faces as an image feature. In Figure 4.5, we illustrate the size growth of unencrypted inverted index, encrypted inverted index, and synchronized cache. The growth is linear, which implies index size increment is proportional to the number of files added. Moreover, in our experiment, we observed that for 20000 images encrypted inverted index size is only 7.05MB, which is about four average size images in our dataset. So the size overhead of our proposed system is very low.

We also observe that feature extraction is the most time-consuming phase of our system. Figure 4.6(a) illustrates required time for extracting features. We observe that face detection and extraction time is the dominating factor in this phase. It requires 464.54 minutes to detect and extract faces from 20000 images in a sequential manner, averaging about 1.39 seconds per image. In addition, the other three features take 85.87 minutes for 20000 images, averaging 0.26 seconds per image. Even though it looks like a long time for a lot of images but the time required for individual images is very little. Furthermore, these experiments are done in a sequential manner. A multi-threaded implementation will certainly reduce the overall time. In addition, in this prototype, we implemented a separate program to call

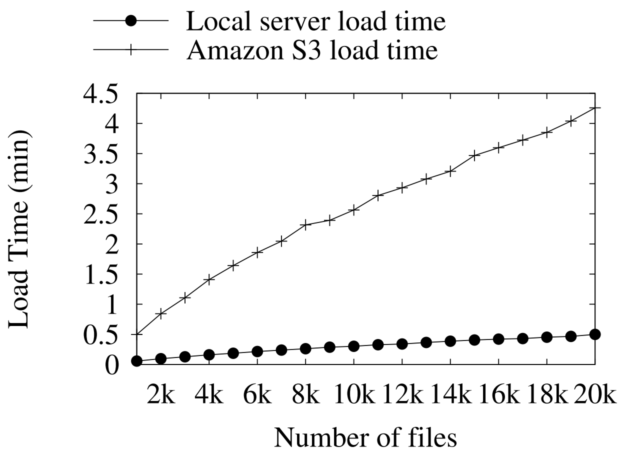


Figure 4.7. Time required for uploading the index for different number of images.

native OpenCV API to detect faces and communicate the results back to the main process, which added extra overhead. In contrast, the transform phase is one of the fastest phase in our implementation. Here, extracted feature values are transformed into an inverted index of search signatures and document ids. We observed that the growth is almost linear and for 20000 images it only requires 696 milliseconds, shown in Figure 4.6(b).

The next phase in our framework is load, where we encrypt and load the inverted index into a cloud storage server. In our experiments, we load the encrypted index into (1) Local server and (2) Amazon S3. We also split the encrypted index into $1MB$ splits and during query processing we downloaded the block that contains the required search signature. We observed that this blocking makes the loading process a bit faster but random query processing time remained almost the same. In Figure 4.7, we show the time required for encrypting and loading the inverted index into the local and Amazon S3 servers. For 20000 images it requires 20.52 seconds to encrypt and load the entire inverted index into the local storage server and 5.65 minutes to complete in the Amazon S3 server. Furthermore, the time growth is linear due to the linear growth of index size.

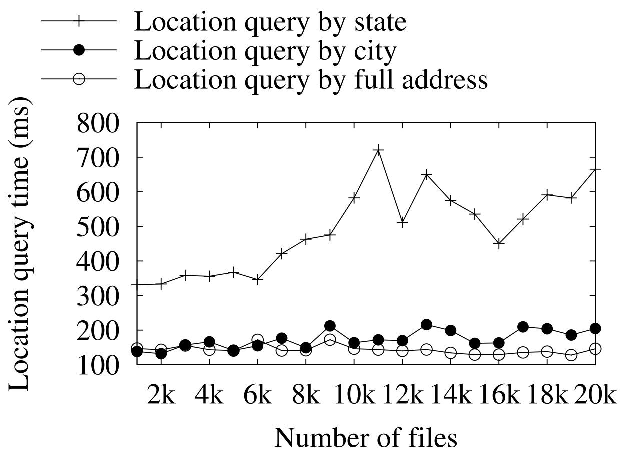
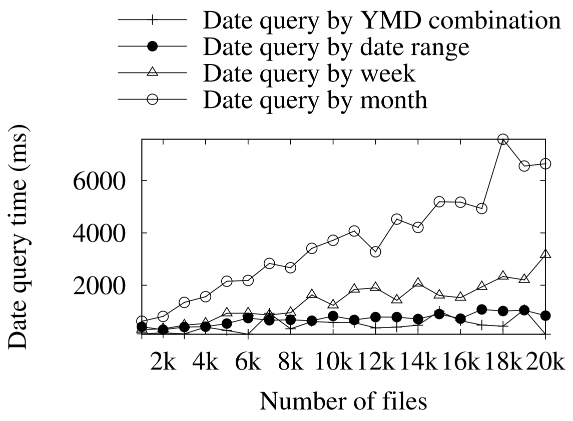


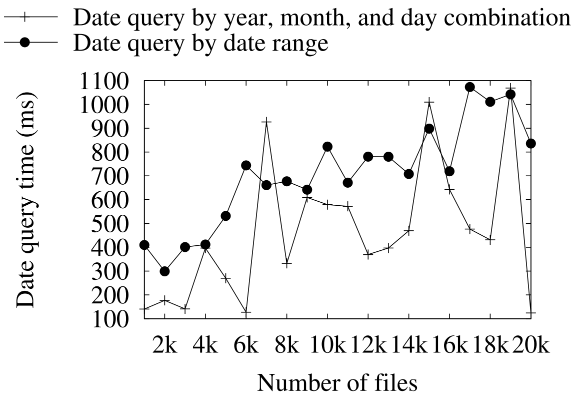
Figure 4.8. Location query execution time

After loading the data into cloud storage server we perform queries on extracted features. For the location feature, we perform a query with five randomly selected states, cities, and full addresses. In Figure 4.8, we show the performance of location queries on different numbers of randomly selected images from the dataset. It is interesting to observe that the query by full address performs fastest among all three query categories. Query by state takes longest to finish and query by city performs in between. This is because time requires to finish a query is proportional to the number to blocks fetched and processed. Very few images are like to have the same full address however more images likely to have common state or city. As a consequence, we observe the above performance. Similarly, for date feature we randomly select five year, month, date(YMD) combinations, date range, months, and weeks. Query by year, month, date combination and range query by date takes the smallest amount time. In contrast, query by month takes the longest and query by the week in between. In Figures 4.9(a) and 4.9(b) we illustrate the performance of different types of date query vs the number of files.

For the FCTH feature, we randomly select five images among input images, get FCTH vectors, then perform the same euclidean LSH transformation defined in the transform phase



(a) Date Query

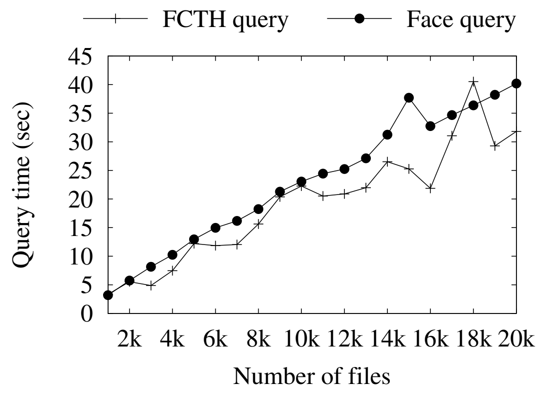


(b) Date Query with YMD

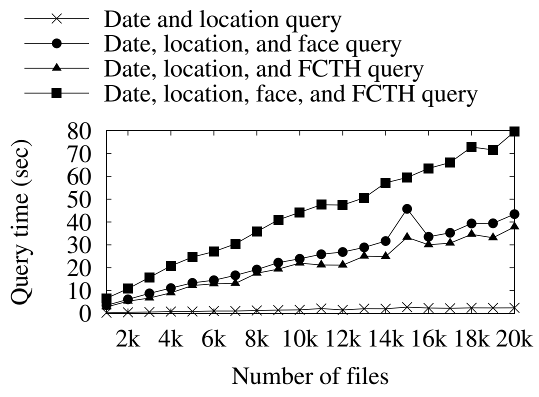
Figure 4.9. Time required for date query vs number of files.

to get the search signatures. For face feature, we randomly select five faces and compute weights with eigenvector information stored in the cache, then perform euclidean LSH transformation and get search signatures. Using these search signatures we get matching images. Finally, we remove images that are too far from the query image. Determining the accuracy of our proposed system for these two features is difficult since the dataset is unlabeled. However, we can estimate the performance with experimentation as shown in Figure 4.10(a). FCTH and face query both takes significantly longer than location and date query, this is due to the nature of these features, extra LSH transformations, and result post-processing. In our experiment we set up a euclidean LSH schema with 4 random projections. For the FCTH feature, each random projection line is divided into 20 unit length buckets and during query time we search we query for images that have distance less than 8 units. For 20000 images we observed 78.4% precision and 16.6% recall. LSH parameters can be adjusted according to the needs of the application. Our experiments give a general idea of performance overhead of different types of complex queries.

We also perform four conjunctive combinations of queries. We perform different types of queries individually then intersect the result to get the final result. First combination is



(a) FCTH and Face Query



(b) Combination Queries

Figure 4.10. Time required for different type of complex queries vs number of files.

date and location query combination, where we combine location queries with date queries. The second combination is date, location, and FCTH query, where we combine three types of queries together. The next combination is date, location, and face query, which is also three type queries. The fourth combination is date, location, FCTH, and face query, which combines all the features our system can extract. As shown in Figure 4.10(b) fourth combination takes longer to perform and the first combination takes the smallest amount of time. This is because location and date queries are individually very fast compared to the other two types of queries.

Finally, we compared the performance of our framework with a naive implementation. In the naive implementation the extract and transform phases remain the same. However, the load and query phase is different. In the naive implementation, we encrypt and upload the inverted index as a single file. During the query phase, we download and decrypt the whole encrypted index to perform queries.

In Figure 4.11, we illustrate the data transfer required to perform a year-month-date(YMD) query using our proposed framework and naive implementation. As the number of query increases data transfer requirement increases liner to the index size. On the other hand,

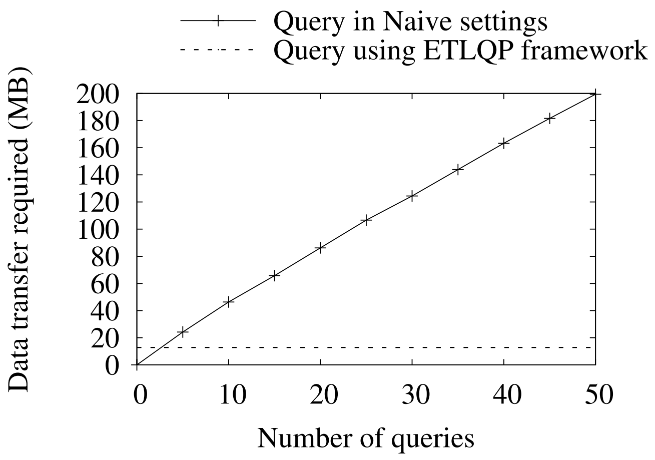


Figure 4.11. Naive query time

in our framework initial index loading phase requires loading the encrypted inverted index than on subsequent query the data transfer is very little.

4.8 Conclusion

In this chapter, we addressed the problem of searchable encryption with a simple server that can support complex queries with multimedia data type. We made several contributions including an extensible general framework with security proof and its implementation. Our defined extract, transform, load, query, and post-process (ETLQP) framework can build an efficient searchable encryption scheme for complex data types (e.g, images). With this framework we can perform very sophisticated queries, such as face recognition, without needing cryptographic computational support from the server. Our implementation shows small overhead for building encrypted search index and performing such complex queries. In addition, we also show that the overhead of general cryptographic operations is negligible compared to other necessary operations of a cloud-based file storage system.

CHAPTER 5

SGX-BIGMATRIX: A PRACTICAL ENCRYPTED DATA ANALYTIC FRAMEWORK WITH TRUSTED PROCESSORS[1]

5.1 Introduction

In this chapter 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 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,

[1]© Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). ACM, New York, NY, USA, 1211-1228. DOI: `https://doi.org/10.1145/3133956.3134095`

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[i].age >= 50:
        count++
        sum += Person[i].income
print sum / count
```

With our framework, such a computation needs to be done using Python NumPy (NumPy, 2019) or pandas (Pandas, 2019) 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., (Stefanov et al., 2013)) 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 chapter 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 (Arasu et al., 2013; Bajaj and Sion, 2014), map-reduce (Schuster et al., 2015), sql execution on spark (Zheng et al., 2017), 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.

5.2 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 Section 3.3, 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 (R, 2019), Octave (Octave, 2019), Matlab (Matlab, 2019), Apache Spark (Spark, 2019), 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 (King and Magoulas, 2016) 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.

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 (Matlab, 2019), Octave (Octave, 2019), NumPy (NumPy, 2019), and Pandas (Pandas, 2019), 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.

5.2.1 Setup 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.

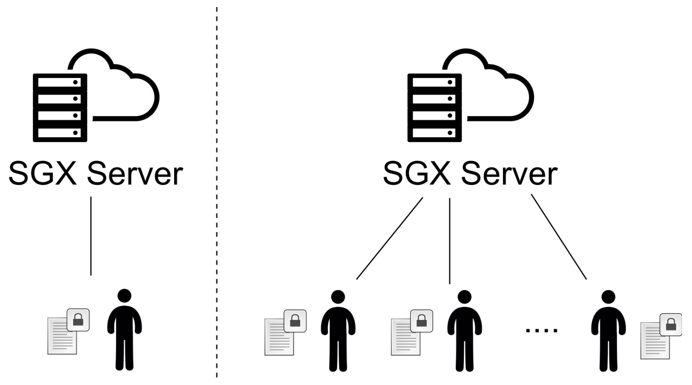


Figure 5.1. General setup of our system - one or more users using SGX based server for data analytics.

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.

5.2.2 Communication Protocols

Code agreement and loading phase To facilitate communication among multi-parties, we assume that the participants know each other' and the 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, $\mathcal{C}$ 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 $\mathcal{C}$ with a randomly generated nonce r_i as follows,

$$\sigma_i = \langle s_i, r_i \rangle = \langle Sign(K^{(i)}_{pri}, \mathcal{C}||r_i), r_i \rangle$$

- 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 $Verify(K^{(j)}_{pub}, s_j, \mathcal{C}||r_j)$ 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 $\mathcal{C}$. Now we are ready to start the SGX loading

- One of the participant uploads $\mathcal{C}$ 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

  $K_{pub}^{enclave}$, $K_{pri}^{enclave}$

- The server generates the following λ_i for all the participants P_i and send to participants

$$\lambda_i = \langle Sign(K_{pub}^{enclave}, \mathcal{C}), ESig, \phi(K_{pub}^{(i)}, K_{pub}^{enclave}||K_i||r_i), r_i\rangle$$

- 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.

5.2.3 Overview

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

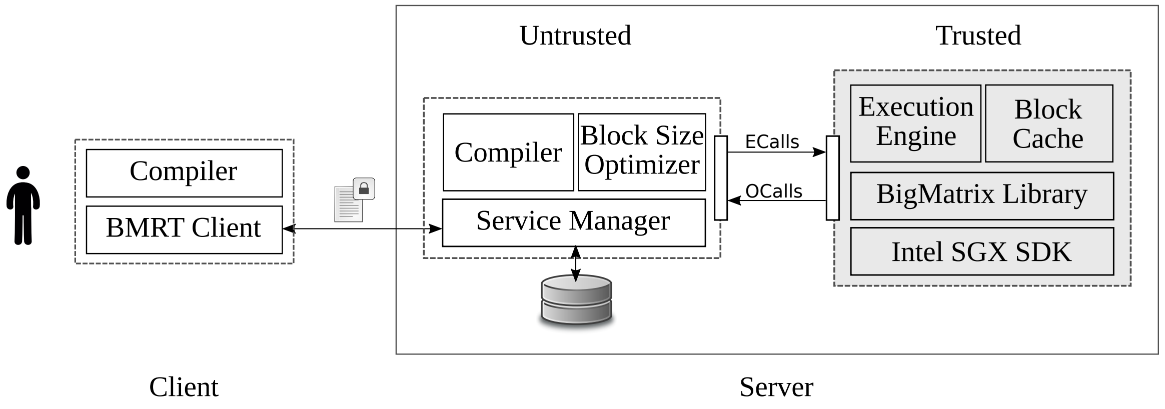


Figure 5.2. Framework Overview.

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.

5.2.4 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 Section 5.2.9, 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 Section 5.3.9. 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 Section 5.3 for more internal details including *trace* and cost of important BigMatrix operations.

5.2.5 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 (Hohn, 2013), 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 Section 5.2.8), 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 Section 5.2.9.

Linear Regression

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 (Lai et al., 1978). 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. We discuss the security guarantees of our framework in more detail in Section 5.4.

PageRank

PageRank is a popular algorithm to measure the relative importance of a node in a connected graph (Page et al., 1999). 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.

5.2.6 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 (Belady, 1966). It is particularly possible in our case since the work load is known and most importantly the code is data oblivious meaning data access 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.

5.2.7 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, $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ 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, $\texttt{Op}(\rho_i) \in \mathcal{O}$ 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:

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

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 5.3. 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

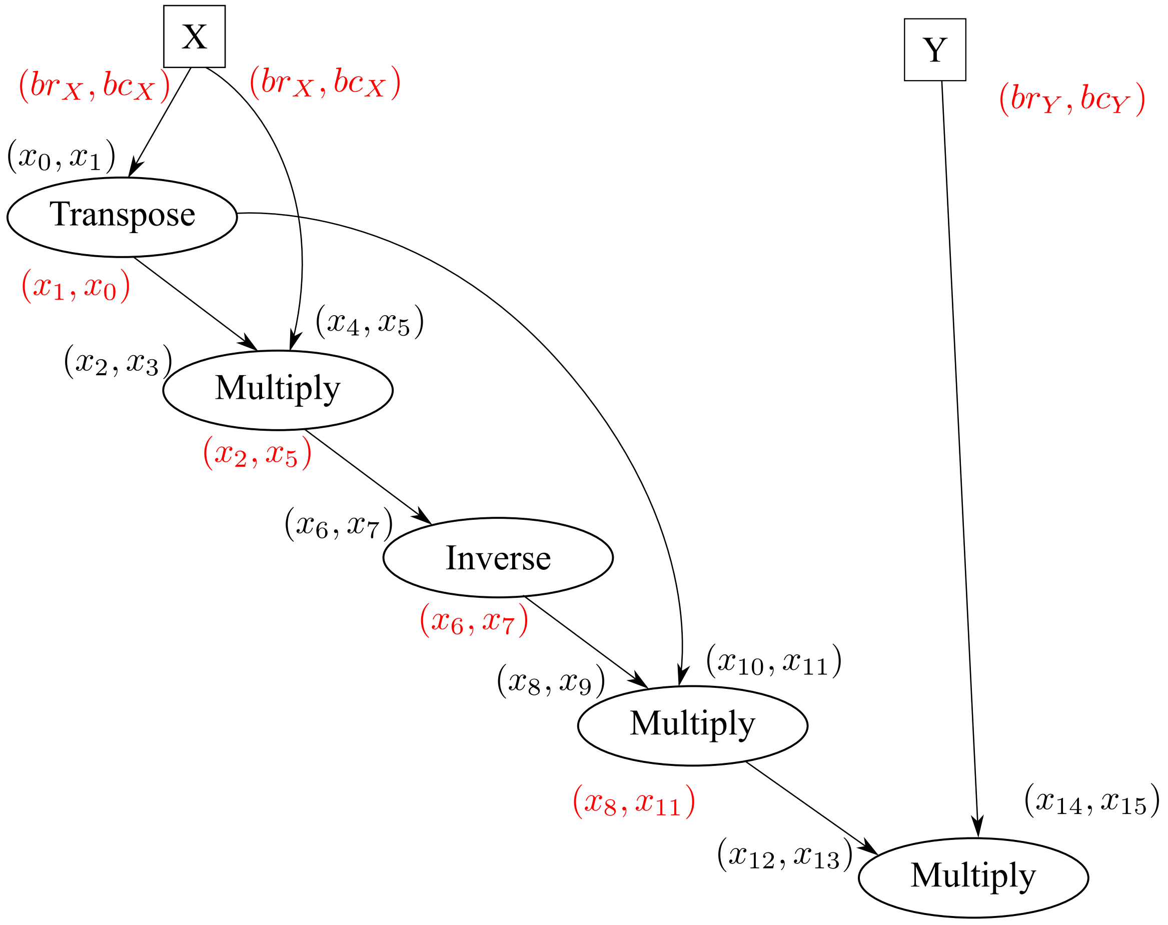


Figure 5.3. Linear regression execution tree for block size optimization.

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.

5.2.8 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 (Hamblin, 1962). 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 (Elmasri, 2008). 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.

5.2.9 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 (Dworkin, 2007), 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. In Figure 5.4 we illustrate the serialization process.

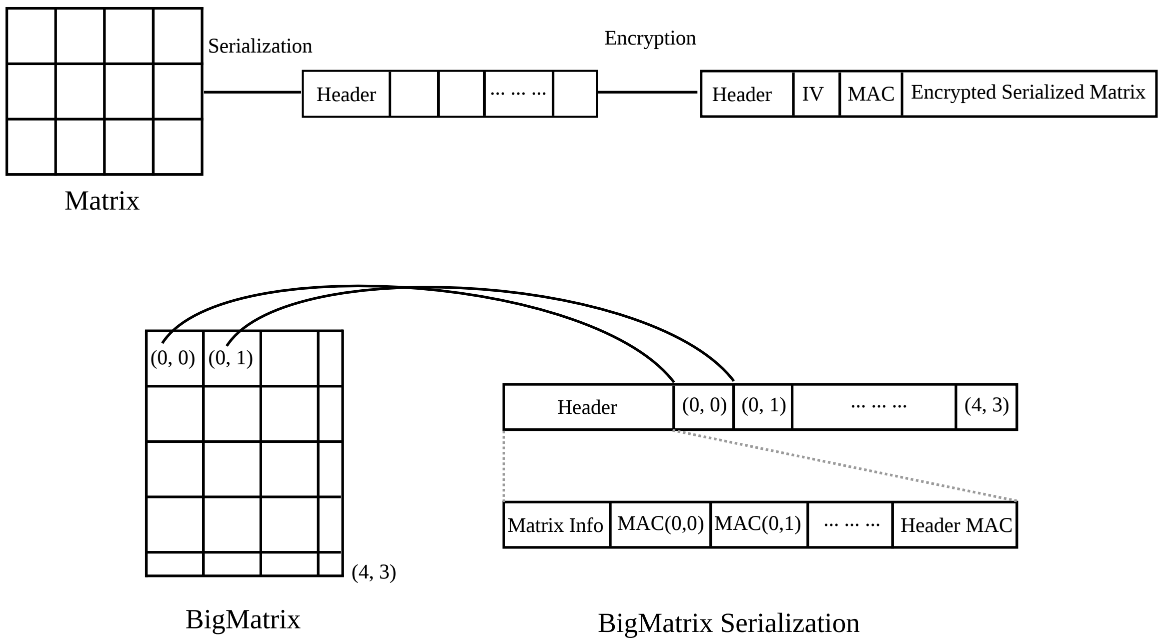


Figure 5.4. Serialization of a Sample BigMatrix.

Storing and Loading mechanism As we explained in Section 3.2, 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.

5.2.10 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.

5.3 BigMatrix API Design

In this section, we discuss more implementation details of different important BigMatrix operations, with 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.

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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 6. 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 6 Matrix inverse by block iterative method.

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

8:     $A_f = -1 * A_e * A_f$
9:     $A_h = A_h + A_g * A_f$
10:    $A_h = inverse(A_h)$
11:    $A_g = A_h * A_e$
12:    $A_g = -1 * A_h * A_g$
13:    $A_e = A_e + A_f * A_g$
14:    $A_f = -1 * A_f * A_h$
15: **end for**
```

The **trace** of the matrix inverse performed in blocks consists of the trace of individual operations in sequences mentioned by Algorithm 6. Similar to previous operations, this operation does not perform any data dependent execution so it is data oblivious.

5.3.5 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 $1 \leq \alpha \leq p$ and $1 \leq \beta \leq q$.

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.

5.3.6 Sort and Top k

We use Bitonic Sort (Batcher, 1968) 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.

5.3.7 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 $1 \leq i \leq m$. 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.

5.3.8 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 (Ohrimenko et al., 2016; Rane et al., 2015). 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.

5.3.9 Join

We only considered a simple join without any special optimizations. We adopted (Agrawal et al., 2006) 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 7 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

Algorithm 7 Data oblivious join algorithm for BigMatrix

```
1: Require: $A$, $B$ input BigMatrix, that has only one column block, $\varphi$ matching condition,
   $k$ = maximum row matches from $A$ to $B$
2: Output: $C$ output BigMatrix.
3: for $u = 1$ to $row\_blocks(A)$ do
4:     $load\_block$ $A_u$
5:     for $i = 1$ to $rows(A_u)$ do
6:         $X = 2k$ dummy block array
7:         $t = K$
8:         for $v = 1$ to $row\_blocks(B)$ do
9:             $load\_block$ $B_v$
10:            for $j = 1$ to $rows(B_v)$ do
11:                if $match(A_u[i, :], B_v[j, :], \varphi)$ then
12:                    $X[t] = A_u[i, :], B_v[j, :]$
13:                else
14:                    $X[t] = dummy, dummy$
15:                end if
16:                $t = t + 1$
17:                if $t >= 2k$ then
18:                    Sort $X$ with bitonic sort such that dummy blocks at the end.
19:                end if
20:            end for
21:        end for
22:        Write first $k$ elements to $C$
23:    end for
24: end for
```

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 7.

5.4 Security Analysis

In this section, we give an overview of the oblivious execution guarantees provided by our system. As we discuss in Section 5.2.5, 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.

5.4.1 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. $\mathcal{C}_\eta = \{\mathcal{F}_1, ..., \mathcal{F}_\eta\}$ be what the code participants agreed on. Here, $\mathcal{C}_\eta$ 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 $\mathcal{C}_\eta$ 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 $\mathcal{C}_\eta$ 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 $\mathcal{C}_\eta$ 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 $\mathcal{H}_\eta = (D, R, O, \mathcal{C}_\eta)$.

- **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 2. *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 $\mathcal{C}_\eta$, $\mathcal{S}$ 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. □

5.4.2 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 (Zheng et al., 2017), 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 5.2.8 could be turned off to prevent this type of leakage.

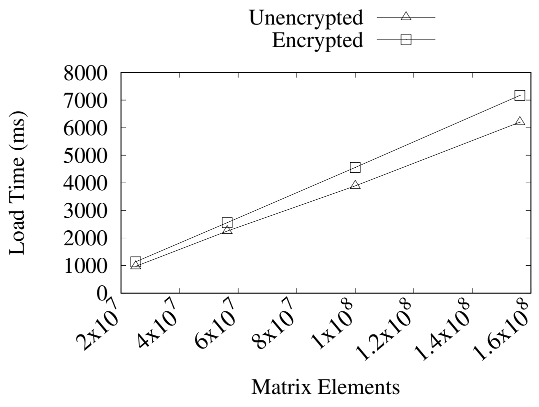
In other cases, intermediate results may reveal some sensitive information. For example, consider the statement `s = count( where(X[1]>=0))` discussed in subsection 5.2.5 where we learn the number of tuples in `X` that has column `1` value bigger or equal than `0`. If `s` value is used in an operation 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 5.2.5. 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.

5.5 Experimental Evaluations

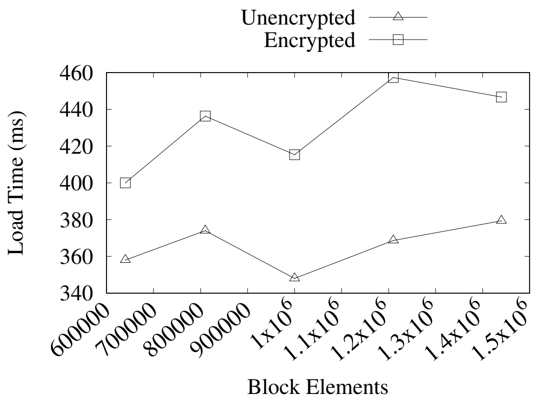
In this section, we perform experimental evaluations to show the effectiveness of our proposed system. We developed a prototype application using *Visual Studio 2012* and *Intel Software Guard Extensions Evaluation SDK 1.0* for Windows. We perform the experiments on a *Dell Precision 3620* computer with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

5.5.1 Individual Operation Performance

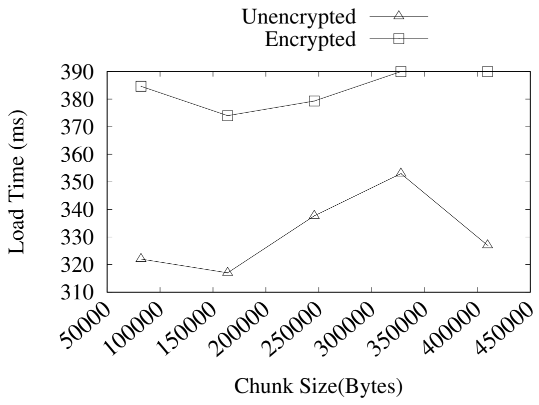
Experiment Setup To understand the performance of the individual operations, we generated *random data sets* with varying sizes and observe the time it takes to perform important operations. However, we acknowledge that the time is sensitive to other events occurring on the operating system. So we rerun 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



(a) Matrix Size



(b) Block Size



(c) Chunk Size

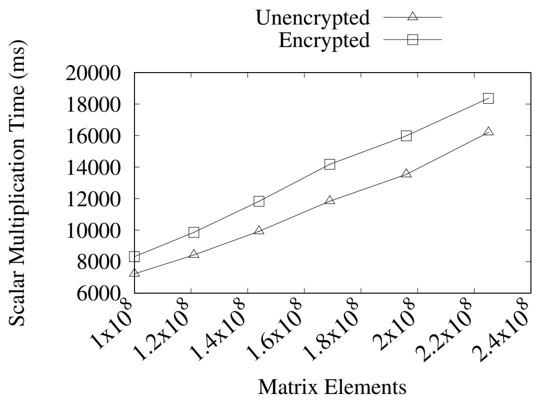
Figure 5.5. Load time encrypted vs. unencrypted

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.

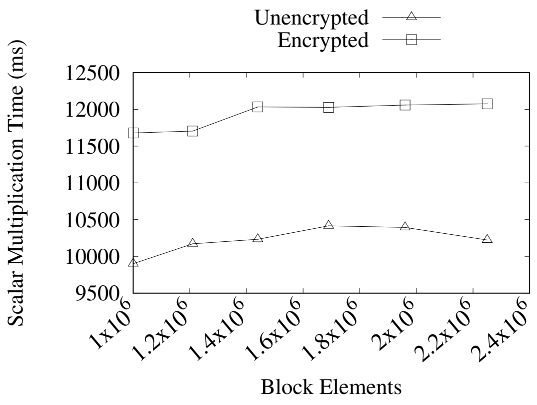
Load Operation We start with load operation, which consists of loading data encrypted with user key, decrypt it, and store again with session key for further use (e.g., the key stored for writing intermediate results to the disk during the operation). As explained in Section 5.2, we break a BigMatrix 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 break the block to smaller chunks. Figure 5.5 illustrates the performance of load operation for randomly generated data. We report three different experiments. In Figure 5.5(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 5.5(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 5.5(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 5.6. 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 5.6(a). However, the block size change does not affect the operation time in most cases as illustrated in Figure 5.6(b). In Figure 5.6(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.

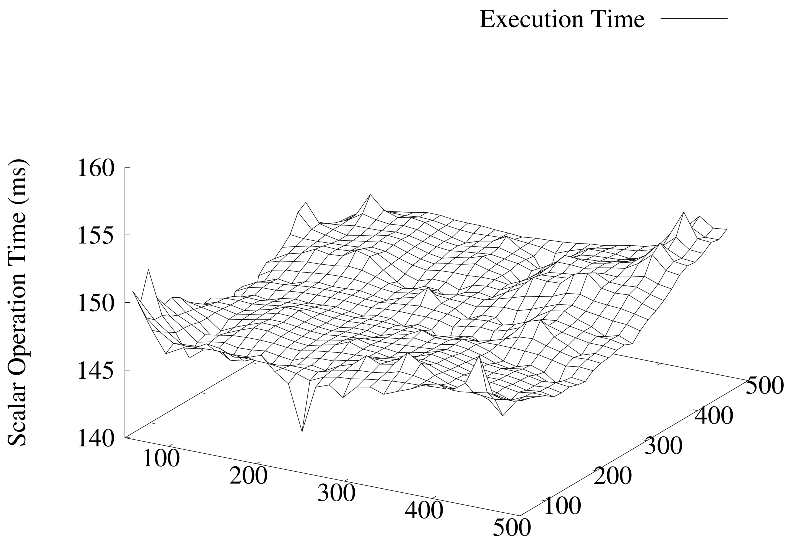
Element-wise Operation Next, we report the performance of element-wise operations. For an element-wise operation, we take two randomly generated matrix and perform an



(a) Matrix Size



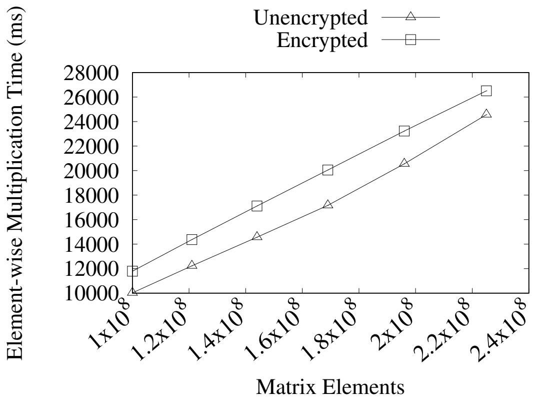
(b) Block Size



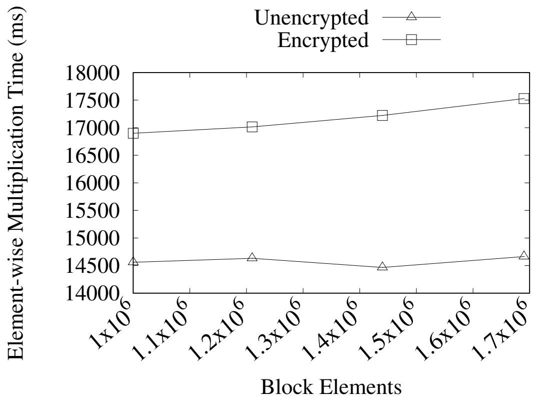
(c) Block Variation

Figure 5.6. 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).

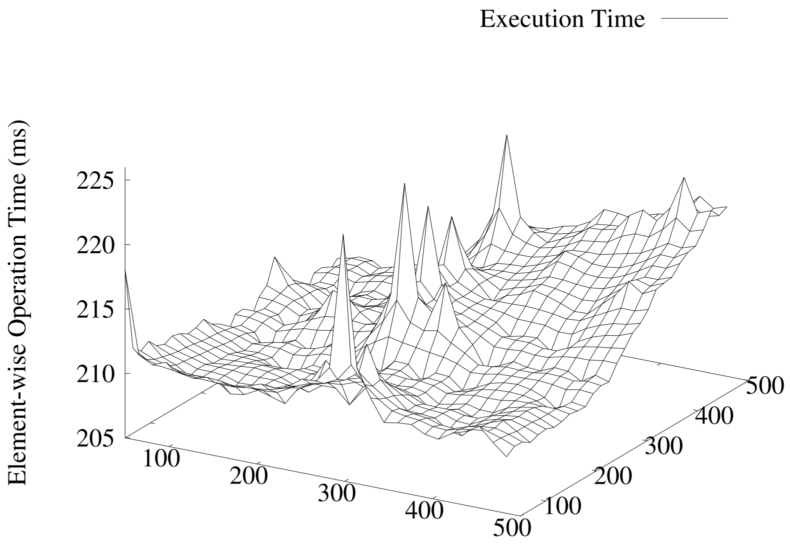
element-wise multiplication and store the result. Similar to the scalar operation, we observe that the operation time is almost linearly proportional to the matrix size (in Figure 5.7(a)). Also we observe that the block size does not have huge effect on the operation time (in Figure 5.7(b)).



(a) Matrix Size



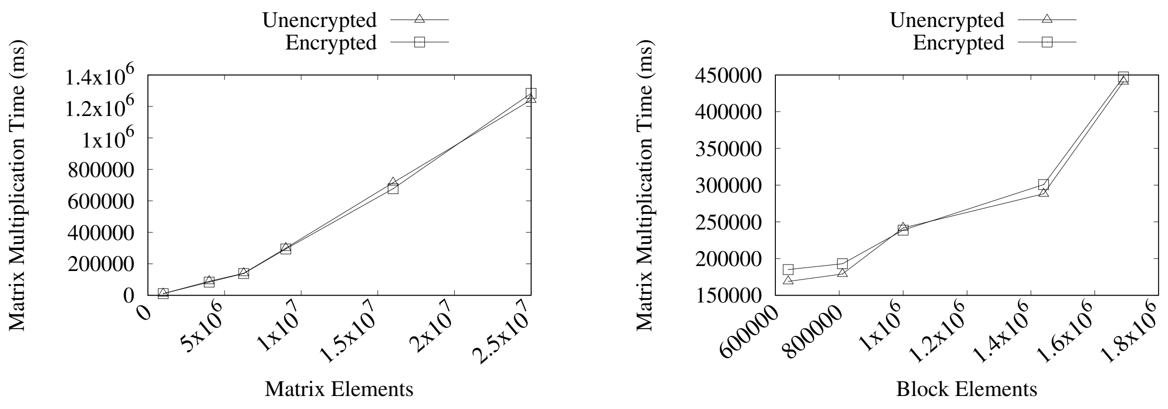
(b) Block Size



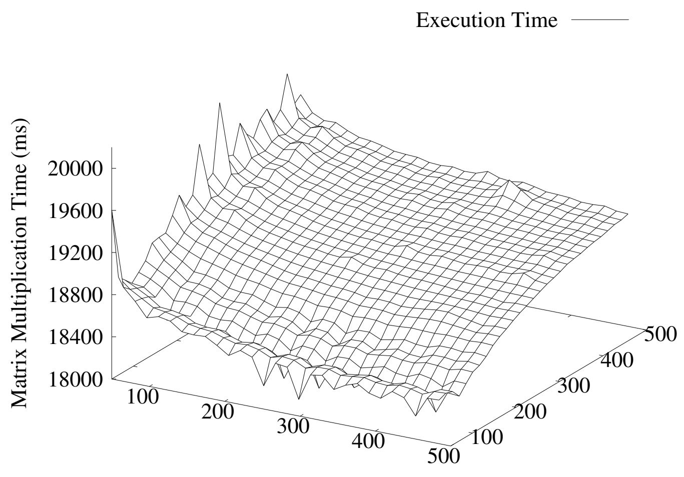
(c) Block Variation

Figure 5.7. 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).

Matrix Multiplication Operation In Figure 5.8, 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 5.8(a)). However, here we also observe that the overhead of



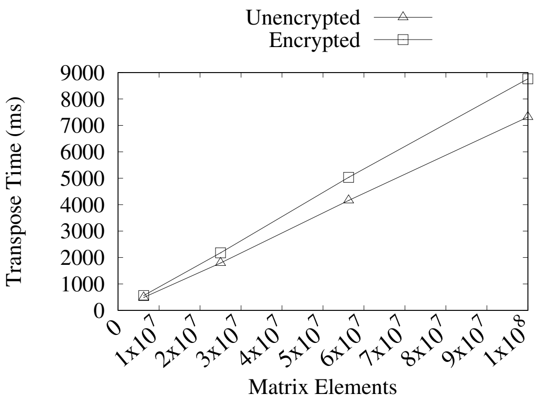
(a) Matrix Size (b) Block Size



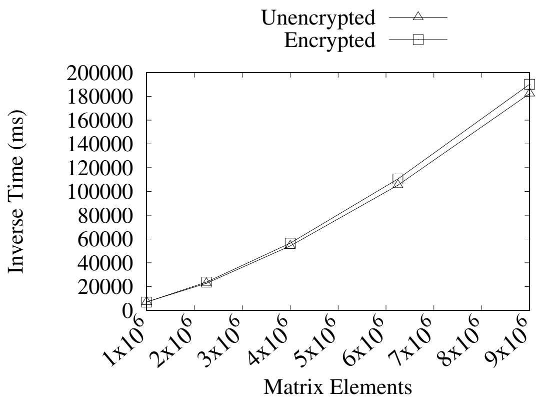
(c) Block Variation

Figure 5.8. 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).

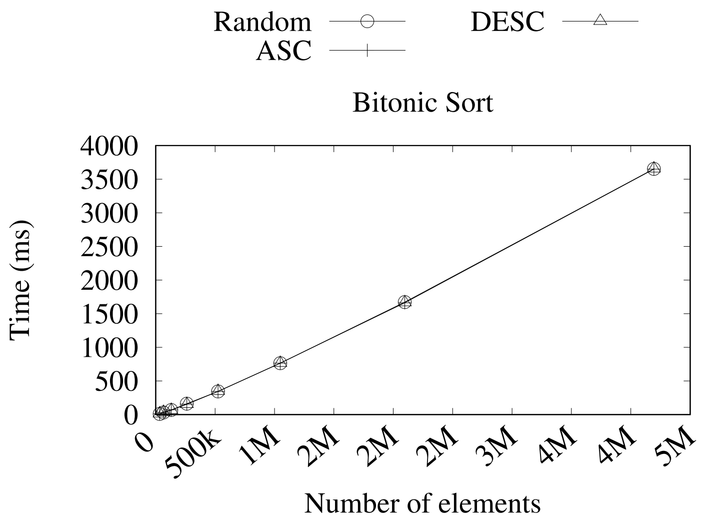
encryption is very low due to the intensive computation required for matrix multiplications. In addition, we observe a big difference in various block sizes as illustrated in Figure 5.8(b). Here we observe a steady growth in the operation time with the block size increment. This can be attributed to 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



(a) Matrix transpose



(b) Matrix inverse

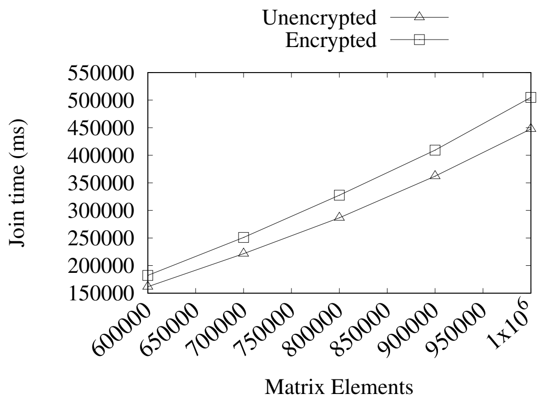


(c) Bitonic sort

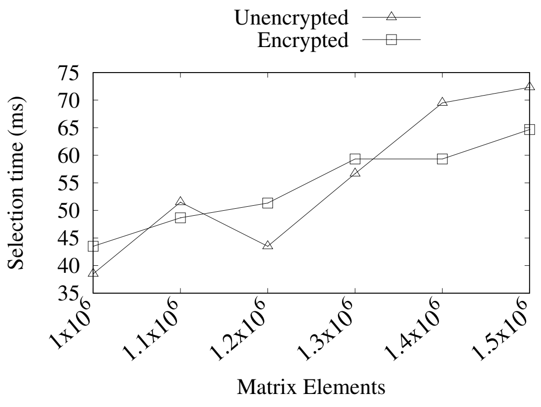
Figure 5.9. Matrix transpose, inverse and sort operation performance.

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.

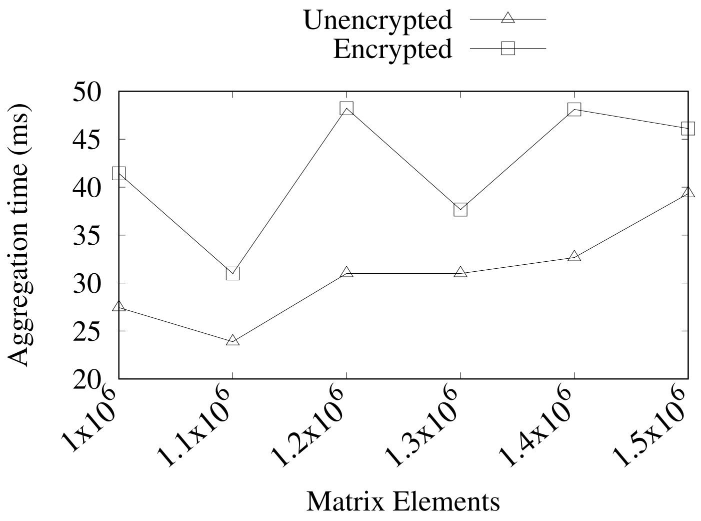
From these sets of experiments, we observe that the operation time is almost always linearly proportional to the size of the matrix. However, block size has an important and varying impact on the execution time. Each operation behaves differently based on these



(a) Join Operation



(b) Selection Operation



(c) Aggregation Operation

Figure 5.10. Relational operations performance encrypted vs. unencrypted.

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 5.9. 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 $\times$ 500 elements size blocks and perform the inverse according to our iterative matrix inverse algorithm described in Algorithm 6. 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 5.10.

5.5.2 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 (Neter et al., 1996; Seber and Lee, 2012). The time required for the execution is reported in Figure 5.11. 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 (Meek et al., 2002) and OnlineNewsPopularity (Fernandes et al., 2015) from UCI Machine Learning Repository (Dua and Graff, 2017). In both cases, we take one column as the target variable and others as the input feature. The results are given in Table 5.1.

As we have proved in Section 5.4, an attacker (e.g., a malicious operating system) can learn limited information due to the data analytics task execution over the encrypted data.

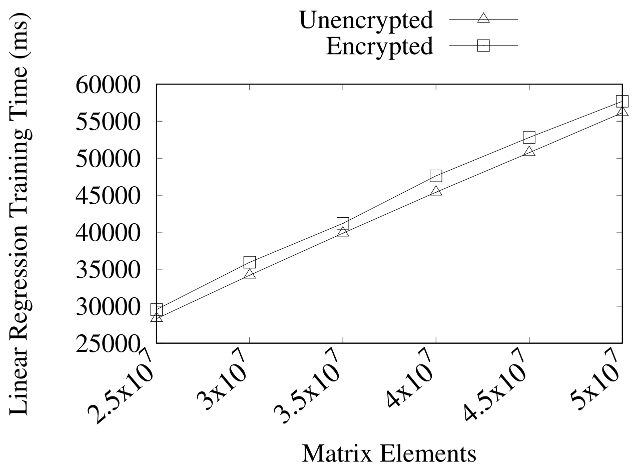


Figure 5.11. Linear Regression time encrypted vs. unencrypted.

In this case study, basically, regression is executed using a sequence of operations 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

Table 5.1. Time results of linear regression on real datasets.

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

vote network (Leskovec et al., 2010), Astro-Physics collaboration network (Leskovec et al., 2007) and Enron email network (Leskovec et al., 2009) from Stanford Network Analysis Project (Leskovec and Krevl, 2014). We generate the adjacency matrix of these networks and perform 40 iteration of PageRank. The execution time is reported in Table 5.2. 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.

Table 5.2. Page Rank on real datasets.

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

Information leakage in PageRank is a sequence of operations with input, output, and block sizes. In addition, the page rank algorithm (as described in Section 5.2.5) 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 (AMPLab, 2014) 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 (Liu et al., 2015). 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 5.3. A solution using traditional multi-party circuit evaluation technique will always incur high overhead compared to a hardware assisted solution, because of the intensive com-

Table 5.3. Two-party matrix multiplication time in ObliVM vs. BigMatrix.

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

munication and complex cryptographic operations. Due to the huge performance difference, we did not conduct more complex comparisons involving ObliVM.

5.6 Conclusion

In this chapter, we proposed an effective, transparent, and extensible mechanism to process large encrypted datasets using a 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.

CHAPTER 6
SGX-IR: SECURE INFORMATION RETRIEVAL WITH TRUSTED PROCESSORS

6.1 Introduction

In this chapter, we discuss secure text and image index building using trusted processors. We proposed a secure index building for text and image data. We build text index to support TF-IDF and different variants (Christopher D. Manning and Schtze, 2008), such as, log, augmented, boolean term frequency with cosine normalization. To do that efficiently, we first do document level summarization and create a stream of tuples of token id, document id, and count. Next, we encrypt and send it to the server. In the server, we compute different TF-IDF values. For face recognition, we encrypt the face images and send them to the server. In the server, we scale all the face images to the same size and calculate eigenfaces (Turk and Pentland, 1991) of the input images. We adopt Jacobi's eigenvector calculation algorithm to compute the eigenfaces. Our contributions in this chapter can be summarized as follows:

- We propose algorithms to build a search index for text data that supports TF-IDF based ranked information retrieval.
- We propose data oblivious algorithm for computing eigenvectors for a given matrix, and show how to use it for face recognition on encrypted image data.
- We build a prototype of the system and show its practical effectiveness.

6.2 System

In this section, we outline our system details including setup and core building blocks.

6.2.1 Setup and Threat Model

Our system has two components: client and server. We briefly discuss these components and the threat model below:

Client To organize and properly utilize our server, we need a client program that runs in users device. It is capable of encrypting user data and send it to server for further computation. We also assume that computational capability of our users' systems are significantly limited. Our primary motivation is to off-load the index creation step for encrypted search to cloud. So that users with smaller capability machines can perform very large privacy preserving computation using secure server.

Server Our server has hardware based trusted execution environment (i.e. Intel SGX) and services that manages and monitors secure enclave life-cycle.

Threat model We follow standard threat model of trusted processor base systems. Specifically, we are considering a scenario, where a user has large number of documents on which she wants to build search index in a secure cloud server. Our user do not trust the server completely. User expects that server will follow the given protocol but server will want to infer information form the data. User only trusts the trusted component of the server, e.g. Intel SGX. Apart for the trusted component, all other components of the server, such as, hyper-visor, operating system, main memory, etc., are not trusted by the user. We are assuming that user can verify that server is executing proper code using proper attestation mechanism. In addition, we assume that communication between client and server is done over secure channel (TLS/SSL).

6.2.2 Storage

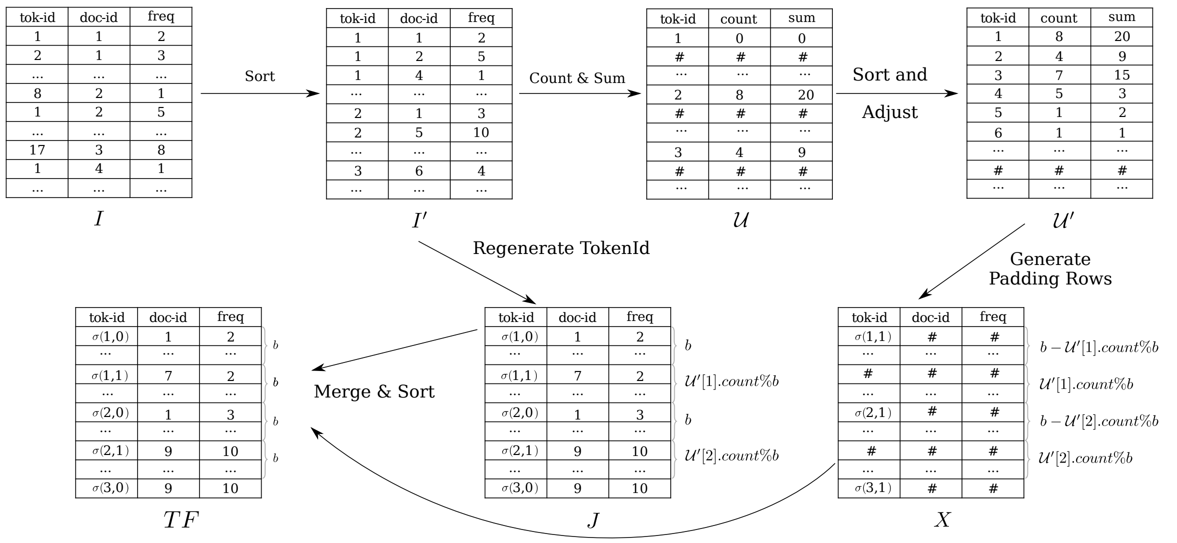
We adopted techniques outlined in Chapter 5 to store large dataset in our system. In short we break a large matrix into smaller blocks and load only blocks that are required to perform the intended operation. Once done we remove the block and encrypt the block again with session key and store in disk. We use least recently used (LRU) technique to manage the block caching. In addition, we also keep the IV and MAC of all the blocks into a header file, which is integrity protected.

6.2.3 Notation

We represent all of our data in two dimensional matrices. In some cases, we also define column names of matrices. We use $A[i]$ to denote i^{th} row of a 2D matrix or table, $A[i].col\_name$ to denote value of the column col_name on i^{th} row.

6.2.4 Secure text indexing in Server

We start by creating token and document pair and encrypting them in the client side. We perform initial tokenization in the client to achieve privacy (i.e., cloud only sees the encrypted data). In addition, we can perform tokenization using traditional algorithms, such as, Porter stemming (Porter, 2006), in one pass over the data. So we tokenize before data encryption. In addition, our client has limited memory so we use hash function to generate token id from lexical token. Let, $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ be a set of input documents, $id(d_i)$ be the document id, Θ_d be set of tokens in document d, $\mathcal{H}$ be a collision resistant deterministic hash function that generates token-id from lexical, $tf_{t,d}$ be the number of times token t occurred in document d. We start by extracting tokens from all the documents. We build matrix I with three columns - $token\_id$, doc_id, and $frequency$. For all t in Θ_d we add $\langle \mathcal{H}(t), id(d), tf_{t,d} \rangle$ to I. We can perform this step easily on client, because in most of cases, text datasets consists of a lot of small files. Furthermore, if we need to process a large



I

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

I'

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

$\mathcal{U}$

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

$\mathcal{U}'$

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

TF

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

J

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

X

tok-id	doc-id	freq
$\sigma(1,1)$	#	#
...	...	...
#	#	#
...	...	...
$\sigma(2,1)$	#	#
...	...	...
#	#	#
...	...	...
$\sigma(3,1)$	#	#

Figure 6.1. Text indexing example

document we can split it into smaller files then process as usual. Next we encrypt I and send it to server.

On server we decrypt I inside enclave. We sort I in ascending order of token id and assign result to I' obliviously, as listed in Algorithm 8 in line 3. We define two matrices $\mathcal{U}(token\_id, count, sum)$ to store the number of documents that a token occurred and summation of total occurrences of all tokens. We iterate sequentially over I, calculate condition $c = I'[i].token\_id \neq I'[i-1].token\_id$, in line 8, if c is true, this implies that we are now reading a new token's information otherwise we are reading current token's information. Based on c we obliviously fill *count* and *sum* column of $\mathcal{U}$ with the count and summation of the token or a dummy value, in lines 9 to 13. We sort $\mathcal{U}$ based on *token_id*, to move the dummy values to end, in line 15. At this stage, in $\mathcal{U}$, i^{th} row has the count and sum of $(i-1)^{th}$ token. So, we shift the count and sum one row in lines 15 and 16. To access a token information obliviously, we need to create equal length for all the tokens. One approach can found the maximum count of any token and allocate that number of elements per token. However, based on the experimental results, we observe that token frequencies follow pareto distribution or power law, i.e., a large number to token will appear in relatively small number

Algorithm 8 Text index building

```
1: **Require:** $I = n \times 3$ input matrix
2: **Output:** $\mathcal{T}$ = Index $p \times 3$ and $DF = p \times 2$ matrices
3: $I' \leftarrow obliviousSort(I, token\_id)$
4: $\# \leftarrow -1$                                                    ▷ dummy value
5: $sum \leftarrow 0, count \leftarrow 0$
6: $\mathcal{U}[0] \leftarrow \langle I'[0].token\_id, 0, 0\rangle$
7: for i = 1 to $I'.length$ do
8:     $c \leftarrow I'[i].token\_id \neq I'[i-1].tok\_id$
9:     $\mathcal{U}[i].token\_id \leftarrow obliviousSelect(I[i].tok\_id, \#, 1, c)$
10:    $\mathcal{U}[i].count \leftarrow obliviousSelect(count, \#, 1, c)$
11:    $\mathcal{U}[i].sum \leftarrow obliviousSelect(sum, \#, 1, c)$
12:    $count \leftarrow obliviousSelect(count, 0, 1, c) + 1$
13:    $sum \leftarrow obliviousSelect(sum, 0, 1, c) + I[i].frequency$
14: end for
15: $\mathcal{U}' \leftarrow obliviousSort(\mathcal{U}, token\_id)$
16: $\mathcal{U}'[i].count = \mathcal{U}'[i+1].count$
17: $\mathcal{U}'[i].sum = \mathcal{U}'[i+1].sum$
18: Remove rows with #
19: Generate inverse document frequency from $\mathcal{U}'$
20: $b \leftarrow optimizeBlockSize(\mathcal{U}.count)$
21: $count \leftarrow 0$
22: for i = 0 to $I'.length$ do
23:    $J[i].token\_id \leftarrow \sigma(I'[i].token\_id, \lfloor \frac{count}{b} \rfloor)$
24:    $count \leftarrow obliviousSelect(count + 1, 0, 1, count < b)$
25: end for
26: for i = 0 to $numToken$ do
27:    for j = $b - 1$ to 0 do
28:        $c \leftarrow \mathcal{U}'[i].count \% b < j$
29:        $t \leftarrow \sigma(\mathcal{U}'[i].token\_id, \lfloor \frac{\mathcal{U}'[i].count}{b} \rfloor)$
30:        $X[i * b + j].token\_id \leftarrow obliviousSelect(t, \#, 1, c)$
31:    end for
32: end for
33: $\mathcal{T} \leftarrow mergeAndSort(J, X, doc\_id)$
```

of documents. So if we block based on maximum token count, we will have lots of dummy entries. To reduce the storage overhead, we split large token into smaller blocks. We use optimization strategies outlined in Section 4.3.6 to find the optimal size b. We adopt this specific technique because it assumes a distribution of frequencies rather than relying on real data. In our scenario, the block size will be revealed to adversary so such an approach will help us reduce information leakage.

Next, we define a deterministic collision-resistant hash function σ that returns a new *token_id* given *token_id* and relative block number. For all the entries in I', we apply σ on token id and generate J matrix with *token_id'*, *doc_id*, and $frequency$, in lines 21 to 25. Next, for all tokens in $\mathcal{U}'$ we add $\mathcal{U}[i]'.count\%b$ dummy entries into X obliviously, in lines 26 to 32. Also, we merge X with J and sort the resulting matrix. Finally, we remove the rows with only dummy entries. So that $\mathcal{T}$ contain m rows per token. Now we push m rows into *any* standard ORAM if we want to access specific token information obliviously in sub-linear time. Otherwise, we can read a token's information obliviously by reading the entire $\mathcal{T}$ matrix once. Figure 6.1 illustrates an example of secure text indexing.

obliviousSelect(a, b, x, y): Let, a and b are two integers to select from, x and y be two comparison variable. We return a if $x == y$, otherwise return b. We show the most important lines of the implementation in the following code listing. We start by copying value of x and y to `eax`, `ebx` then perform `xor`, in line 5, so if x and y are equal then zero flag is set. Next, we copy value of a and b into `ecx` and `edx`. Now we conditionally move, in line 9, based on zero flag, between `ecx` and `edx` registers. So if zero flag is set then `edx` will get value of `eax` otherwise the value will remain unchanged. Finally, we return the value of `edx` register. In our setup, adversary will only observe sequence of operations but will not know exactly which value was selected.

```
1 oblivousSelect(a, b, x, y):
2 ...
3 mov %[x],%%eax
4 mov %[y],%%ebx
5 xor %%eax, %%ebx
6 ...
```

```
mov %[a],%%ecx
mov %[b],%%edx
cmovz %%ecx,%%edx
...
mov %%edx, %[out]
```

Oblivious Bitonic Sorting We use bitonic sort (Batcher, 1968) for sorting in the server, because in bitonic sort we perform exactly the same comparisons irrespective of input data value. Specifically we use the arbitrary length version to save time and space. In our empirical analysis, we have observe that total number to rows in matrices I, J, etc. can go up to 2^{26} or more. So to utilize traditional bitonic sort, we need to pad the matrices with dummy entries to 2^{27} number of rows. In practice, we observe that this adds upto 40% overhead. In addition, we implement an iterative variant of the bitonic sort of arbitrary length. We avoid recursion to save stack space, since SGX is a memory constraint environment. In Algorithm 9 we define our iterative implementation. We utilize existing definition of iterative bitonic sort defined in (Christopher, 2019; Wikipedia contributors, 2019) for length 2^k. The core idea behind this implementation is that any number can be expressed as summation of one or more 2^k format numbers, such as, $N = 2^{x_1} + ... + 2^{x_m}$. For given N element array/matrix, we consider it as block of 2^{x_1}, ..., $2^{x_{m-1}}$ rows. We sort first $(m - 1)$ blocks in descending order and sort the final block m ascending order using existing definition of non-recursive bitonic sort. Next we merge last two blocks m and $(m-1)$ in ascending order. We iteratively continue to merge the result with previous block. So, finally we get a sorted array/matrix in ascending order. In addition, we also make the exchange step oblivious. So the adversary will not get any additional information from sequence of comparisons and successful swaps.

Algorithm 9 Non-recursive non-trivial bitonic sort for arbitrary length

```
1: for d = 0 to $\lceil log_2(N) \rceil$ do
2:     if $((N >> d)\&1) \neq 0$ then
3:         $start \leftarrow (-1 << (d+1))\&N$
4:         $size \leftarrow 1 << d$
5:         $dir \leftarrow (size\&N\& - N) == (N\& - N)$
6:         $bitonicSort2K(start, size, dir)$
7:         if !dir then
8:             $bitonicMerge(start, N - start, 1)$
9:         end if
10:     end if
11: end for
```

6.2.5 Image indexing for face recognition

Oblivious Eigenface To make the entire process data oblivious, we need oblivious eigen vector calculation and oblivious comparison of projected test image. We discuss oblivious eigen vector calculation in a separate subsection. For oblivious distance calculation, we compute the distance function for all input training faces. We create $M \times 2$ matrix $\mathcal{F}$, where first column is face id and second column contains 1 if that face's distance is bellow the threshold and 0 otherwise. Finally, we sort $\mathcal{F}$ based on second column in descending order to get the matching face id.

6.2.6 Oblivious eigen vector calculation

We adopted Jacobi method (Golub and Van der Vorst, 2001) of eigen vector computation for oblivious calculation. In Jacobi eigenvalue method, as outlined in Algorithm 10, we start with finding maximum value and index of maximum value (k, l) in input symmetric matrix. Next we compute few values based on max, $A_{k,l}$, $A_{l,k}$ (in lines 13 to 20). In line 21, we assign zero to $A_{k,l}$ and $A_{l,k}$, and compute $A_{k,k}$ and $A_{l,l}$. Then, we perform rotations on k^{th} column and l^{th} column. However, since this is a symmetric matrix we can perform same computation only on upper triangular matrix as described in lines 22 to 31. We also perform

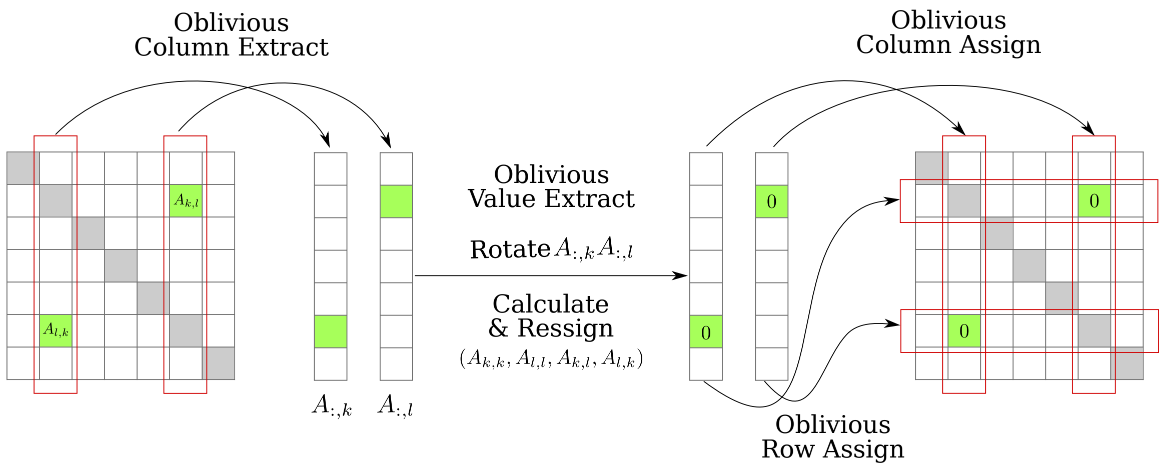


Figure 6.2. Oblivious Eigen Matrix calculation

the same rotation on eigen matrix E, which is initialized with identity matrix (in lines 32 to 34). We repeat the process until the input matrix becomes diagonal. The values in the main diagonal approximates the eigen values and normalized version of the the eigen matrix E consists of all the eigen vectors of matrix A. In practice, for eigenface, we need top n largest eigen vectors. To extract top n eigen vectors we sort the eigen vectors based on eigen values.

We define a few additional oblivious functions, which we use later in the oblivious eigenvector calculation algorithm.

obliviousValueExtract(U, k): Given an array U and an index k, we extract the value of U_x obliviously. We initialize $v \leftarrow U_0$ then we iterate over all the elements in the array and run *obliviousSelect(v, U_i, i, k)* and assign the return value to v. As a result, when i is equal to k the return value will be U_k, otherwise it will always return existing value of v. Similarly we define *obliviousValueAssign(U, k, a)*, where we assign value a to k^{th} location of input array U.

obliviousColumnExtract(A, k): Given a 2D matrix A and a column index k, we extract k^{th} column of matrix A. We utilized previously defined oblivious select. For a given row, r, we iterate over all the columns c and assign U_r to output of $obliviousSelect(A_{r,c}, U_r, r, k)$. As a result, when r is equal to k then we get the value of $A_{r,c}$ in U_r, otherwise value of U_r remains the same. Similarly, we define *obliviousColumnAssign()* and *obliviousRowAssign()*, where we assign the input column in a specific column or row of input matrix. In addition, we also define *obliviousConditionalColumnAssign()* and *obliviousConditionalRowAssign()*, with one more boolean parameter, where we perform the assignment if and only if the boolean parameter is true.

obliviousMaxIndex(A): We copy current element and current maximum (initially zero) into floating-point stacks in `st(1)` and `st(0)`. Next, we perform floating-point comparison, as shown in line 3 in the following code listing, which sets appropriate registers. Next, we perform conditional move operation which swaps values in the floating-point stack if the proper flag is set, in line 4. As a result, the maximum value will be at the top of the floating-point stack, which we assign back to the maximum variable.

Next, we move row of maximum and current element into two registers, `eax` and`ebx` respectively, in lines 6 and 7. Then we again conditionally swap these two registers and the flag was set during the initial float comparison, in line 8. So, the index of the largest value will be in `eax`, which read back to *maxRow* variable. Similarly, we also perform a conditional move for maximum column index in lines 10, 11, and 12.

```
1 obliviousMaxIndex(m, e, mR, mC, eR, eC):
2 ...
3 fucomi %%st(1), %%st
4 fcmovb %%st(1), %%st
5 ...
6 mov %[mR],%%eax
7 mov %[eR],%%ebx
8 cmovb %%ebx, %%eax
9 ...
```

```
mov %[mC],%%eax
mov %[eC],%%ebx
cmovb %%ebx, %%eax
...
```

Now, we create the data oblivious version of Jacobi's eigenvalue calculation listed in Algorithm 11. First, we fix the number of iterations and do not return early based on convergence. As a result, the adversary can not learn information about data based on the iteration count. Next for finding the maximum, max and index of maximum elements (k, l) using *obliviousMaxIndex* operation in line 6. We extract k^{th} and l^{th} columns, U and V using *obliviousColumnExtract*, in lines 8 and 9. Next, we extract values of kk and ll using *obliviousValueExtract* operation in lines 10 and 11. Next, we calculate value τ, t, s in lines from 12 to17. For t we first calculate both $\frac{max}{d}$ and $|\frac{1}{|p|+\sqrt{p^2+1}}|$ then we obliviously choose correct version using *obliviousSelect*. Next, we perform the rotation on extracted column and assign k^{th} and l^{th} value of U and V appropriately in lines 18 to 24. Then, we assign the column back to A if the algorithm is not converged, in lines 25 to 28. We determine convergence condition $\mathcal{C}$ in line 7 by check whether the current maximum is smaller than a predefined small value. We perform similar rotation on matrix E, in lines 29 to 33. We iterate the entire process fix number of times. Finally, we normalize and sort the eigen matrix based on eigenvalues in lines 35 to 37. The normalization process is naturally data oblivious, i.e. we execute same instructions irrespective of input data. For sorting we use our previously defined bitonic sort.

6.3 Experimental Evaluations

In this section, we discuss the performance of our proposed system. We developed a prototype of the proposed system SGX-IR, using *Intel Software Extensions SDK 2.6* for Linux. We perform the experiments on a *Intel LR1304SPCFSGX1* server with *Intel® Xeon®*

CPU E3-1270 @ 3.80GHz CPU, *64GB* main memory, *128MB* enclave memory, and running *Ubuntu Server 18.04*.

6.3.1 Bitonic sort

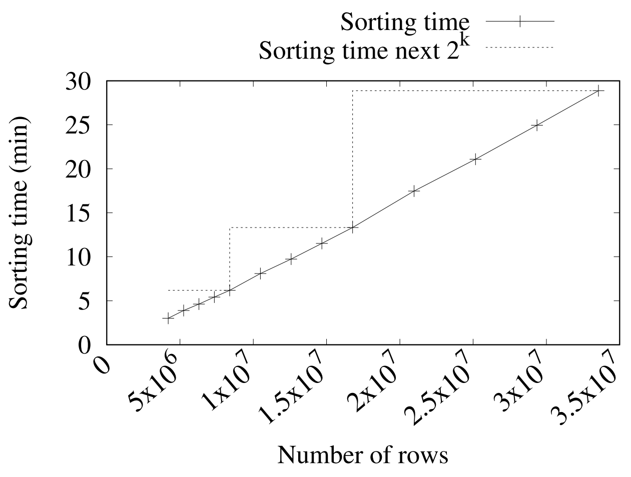
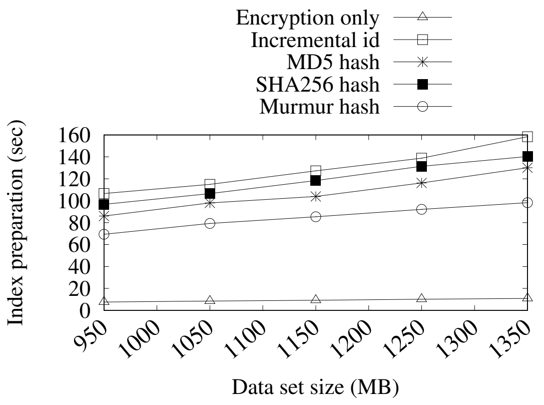


Figure 6.3. Bitonic sort time

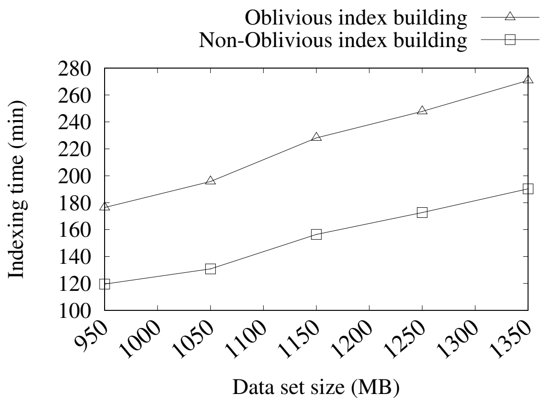
We first briefly discuss the impact of arbitrary length bitonic sort. In Figure 6.3 we show sort time of different size matrices with 3 columns. The solid line represents sorting duration and the dotted line represents sorting duration for the next 2^k element matrix. With the arbitrary length algorithm, the growth of the required time is linear. On the other hand, if we were to use 2^k version our required time would be the dotted line, where we have about 50% overhead in extreme cases.

6.3.2 Text Indexing

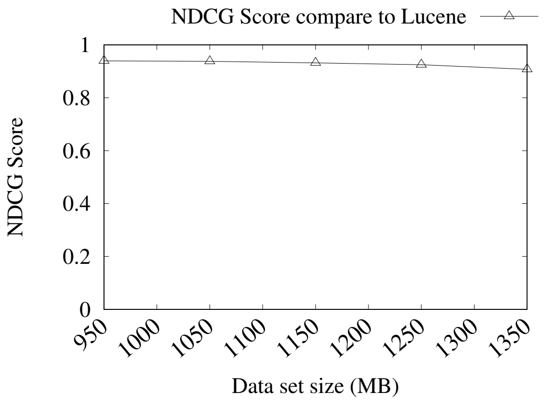
Dataset We use Enron dataset (Klimt and Yang, 2004) for text indexing experiments. We randomly select sub-set of files from the Enron dataset. Then we parse the data in the client end. We tokenize, stem (Porter, 2006), and build document token pairs. Next, we encrypt and send the data to server. In server we follow the algorithm outlined in Sec 6.2.4 to sort and generate index.



(a) Client side processing



(b) Index building time



(c) NDCG Scores compared to Apache Lucene

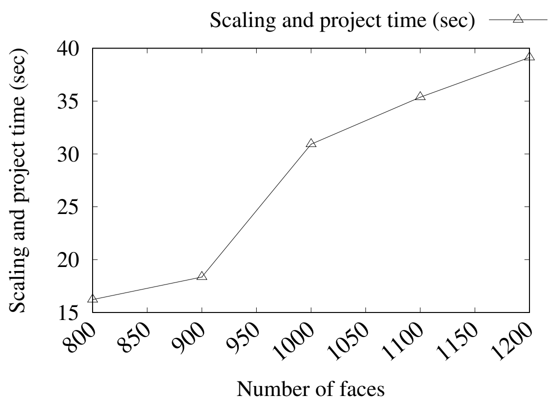
Figure 6.4. Enron experiment

Performance In Figure 6.4(a) we show the performance of client-side index pre-processing. We show time to build the input matrix using different types of cryptographic and non-cryptographic hashing functions and keeping an in-memory map for token id generation. We observe that incrementation token id generation is the most expensive and non-cryptographic hash, i.e., MurMur Hash, is the least expensive. In addition, we show the time required for only encrypting the data without performing any tokenization and token id generation, which shows the overhead of read-write and encryption. The gap between encryption only and a hashing token id generation signifies the overhead of our tokenization and matrix generation. Finally, in all theses cases we observe the growth is linear.

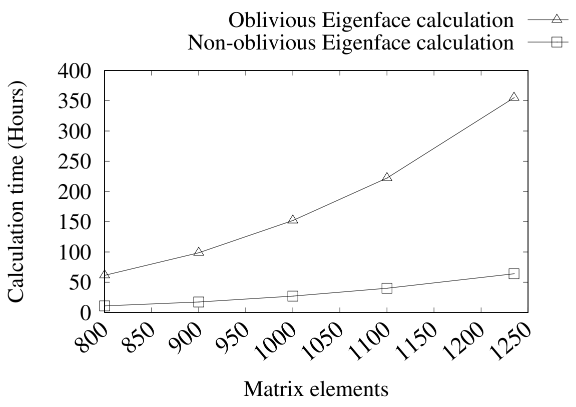
In Figure 6.4(b) we show server-side index processing cost. We compare our results with a non-oblivious version of a similar index building. For non-oblivious implementation, we sort the input matrix based on token id then build a separate matrix that is equivalent to $\mathcal{T}$ by iterating the sorted matrix. We observe that the obliviousness implementation is about 1.49 times more expensive. Finally, to show the effectiveness, of our information retrieval system we compare ranked results with Apache Lucene (ApacheLucene, 2019) library result. Apache Lucene library is the de-facto standard of information retrieval library and is used in numerous commercial and open-source search engine software, such as Apache Solr (ApacheSolor, 2019), Elasticsearch (Elastic, 2019). We adopt normalized discounted cumulative gain (NDCG) score (Christopher D. Manning and Schtze, 2008) to compare the ranked results of the information retrieval systems. In Figure 6.4(c) we report the NDCG score of our system compared to Apache Lucene for randomly selected $1,000$ tokens. We observe that our scores are about 0.92. In other words, our model works relatively well compared to the industry-standard information retrieval system. In addition, we allow different types of frequency normalizations. So users of the system can tune the normalization functions to improve the results as needed.

6.3.3 Face Recognition

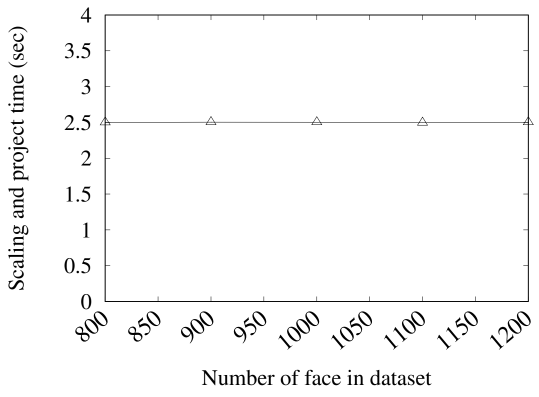
Dataset We use *Color FERET* (Phillips et al., 1998, 2000) dataset for testing face recognition. *Color FERET* dataset contains a total of 11338 facial images, which were collected by photographing 994 subjects at various angles. The dataset images contain face images in front of different background often containing other objects. So, wrote a face detection program using OpenCV implementation of haar cascade classifier (Viola and Jones, 2001; Lienhart and Maydt, 2002) for frontal face. We extract frontal face images with `fa` suffix from the dataset. We found that there are total 1364 such images. Our face detection system successfully detected 1235 images, yielding 90.33% accuracy. Here, most of the failed cases



(a) Eigen-face preparation



(b) Index building time



(c) Eigen-face recognition

Figure 6.5. Oblivious eigen-face experiment

has glass or similar face obstructing additions. We extract the frontal faces and scale to 100×100 faces. Then we randomly selected sub-set of images and build our face recognition dataset.

In Figure 6.5(a), we show the performance of face image preparation and in Figure 6.5(c) face finding overhead. Both of these operations are standard matrix operations. So overheads are very minimal under a minute. In Figure 6.5(b) we show the required time for building the eigenface index, which is dominated by the overhead of in eigenvector calculation. We compare our results with a non-oblivious version of the algorithm. For the non-oblivious version, we implement the Jacobi algorithm without accessing the matrix values obliviously.

We observe that both the cases the required times are quite large because of the large number required iteration for the Jacobi algorithm to converge. We observe that, the obliviousness adds around 5 times more overhead. We incur such large overheads because we read the entire matrix to extract and assign the required rows and columns.

6.4 Conclusion

In this chapter, we propose a secure information retrieval system for text and image data. Unlike other existing works, we focus on building the encrypted index in the cloud securely using trusted processors, such as Intel SGX. We address the information leakage due to memory access pattern issue by proposing data oblivious indexing algorithms. We build a text index to support ranked document retrieval using TF-IDF scoring mechanisms. Also, we build an image index to support the face recognition query. In addition, we also propose a non-recursive version of the bitonic sort algorithm for arbitrary input length.

Algorithm 10 Eigen vector with Jacobi method

```
1: Require: $A = n \times n$ diagonal matrix
2: Output: $E$ = eigen vectors, $V$ = eigen values
3: $E \leftarrow identity(n)$
4: $\epsilon_1 \leftarrow 10^{-12}$, $\epsilon_2 \leftarrow 10^{-36}$
5: for it = 0 to $n^2$ do
6:     $max \leftarrow max(A)$ in off-diagonal upper triangle
7:     $(k, l) \leftarrow maxIndex(A)$
8:     if $max < \epsilon_1$ then
9:         $V_i \leftarrow A_{i,i}, \forall i \in 0$ to $n$
10:        $normalize(E)$
11:        return
12:    end if
13:    $d \leftarrow A_{l,l} - A_{k,k}$
14:    if $|A_{k,l}| < \epsilon_2 |d|$ then
15:        $t \leftarrow \frac{A_{k,l}}{d}$
16:    else
17:        $p \leftarrow \frac{d}{2A_{k,l}}$
18:        $t \leftarrow |\frac{1}{|p|+\sqrt{p^2+1}}|$
19:    end if
20:    $c \leftarrow \frac{1}{\sqrt{t^2+1}}$, $s \leftarrow t \times c$, $\tau \leftarrow \frac{s}{1+c}$
21:    $A_{k,k} \leftarrow A_{k,k} - t \times A_{k,l}$, $A_{l,l} \leftarrow A_{l,l} + t \times A_{k,l}$, $A_{k,l} \leftarrow 0$
22:    $\mathcal{R} \leftarrow s. \begin{bmatrix} -\tau & -1 \\ 1 & -\tau \end{bmatrix}$
23:    for $i = 0$ to $k - 1$ do
24:        $\begin{bmatrix} A_{i,k} \\ A_{i,l} \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} A_{i,k} \\ A_{i,l} \end{bmatrix}$
25:    end for
26:    for $i = k + 1$ to $l - 1$ do
27:        $\begin{bmatrix} A_{k,i} \\ A_{i,l} \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} A_{k,i} \\ A_{i,l} \end{bmatrix}$
28:    end for
29:    for $i = l + 1$ to $n - 1$ do
30:        $\begin{bmatrix} A_{k,i} \\ A_{l,i} \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} A_{k,i} \\ A_{l,i} \end{bmatrix}$
31:    end for
32:    for $i = 0$ to $n - 1$ do
33:        $\begin{bmatrix} E_{i,k} \\ E_{i,l} \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} E_{i,k} \\ E_{i,l} \end{bmatrix}$
34:    end for
35: end for
```

Algorithm 11 Oblivious Eigen vector with Jacobi method

```
1: Require: $A = n \times n$ diagonal matrix
2: Output: $E$ = eigen vectors, $V$ = eigen values
3: $E \leftarrow identity(n)$
4: $\epsilon_1 \leftarrow 10^{-12}$, $\epsilon_2 \leftarrow 10^{-36}$
5: for it = 0 to $n^2$ do
6:     $max, k, l \leftarrow obliviousMaxIndex(A)$
7:     $\mathcal{C} \leftarrow max < \epsilon_1$
8:     $U \leftarrow obliviousColumnExtract(A, k)$
9:     $V \leftarrow obliviousColumnExtract(A, l)$
10:    $kk \leftarrow obliviousValueExtract(U, k)$
11:    $ll \leftarrow obliviousValueExtract(V, l)$
12:    $d \leftarrow ll - kk$
13:    $m \leftarrow |max| < \epsilon_2 |d|$
14:    $p \leftarrow \frac{d}{2 \times max}$
15:    $t_1 \leftarrow \frac{max}{d}$, $t_2 \leftarrow |\frac{1}{|p| + \sqrt{p^2+1}}|$
16:    $t \leftarrow obliviousSelect(t_1, t_2, m, 1)$
17:    $c \leftarrow \frac{1}{\sqrt{t^2+1}}$, $s \leftarrow t \times c$, $\tau \leftarrow \frac{s}{1+c}$
18:    $\mathcal{R} \leftarrow s. \begin{bmatrix} -\tau & -1 \\ 1 & -\tau \end{bmatrix}$
19:    $\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$
20:    $kk \leftarrow kk - t \times max$, $ll \leftarrow ll + t \times max$
21:    $obliviousValueAssign(U, k, kk)$
22:    $obliviousValueAssign(V, l, ll)$
23:    $obliviousValueAssign(U, l, 0)$
24:    $obliviousValueAssign(V, k, 0)$
25:    $obliviousConditionalColumnAssign(A, U, k, !\mathcal{C})$
26:    $obliviousConditionalColumnAssign(A, V, l, !\mathcal{C})$
27:    $obliviousConditionalRowAssign(A, U, k, !\mathcal{C})$
28:    $obliviousConditionalRowAssign(A, V, l, !\mathcal{C})$
29:    $U \leftarrow obliviousColumnExtract(E, k)$
30:    $V \leftarrow obliviousColumnExtract(E, l)$
31:    $\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$
32:    $obliviousConditionalColumnAssign(E, U, k, !\mathcal{C})$
33:    $obliviousConditionalColumnAssign(E, V, l, !\mathcal{C})$
34: end for
35: $V_i \leftarrow A_{i,i}, \forall i \in 0$ to $n$
36: $normalize(E)$
37: $sort(E)$ based on $V$
```

CHAPTER 7
CONCLUSION

In this dissertation, we provide efficient and secure solutions to indexing, searching, and analyzing encrypted data in cloud computing environment. We show that proposed solutions allow the user to utilize the cloud computing environment in a more secure manner.

First, we show that we can perform very complex search queries on encrypted image data, such as face recognition, without cryptographic computation support from the cloud server. We achieve this by converting complex search queries into a series of simple equality queries. We defined series of feature extraction and transformation functions that allows such simplification. Furthermore, we propose a generic framework for building and querying search index for any data type. As a result, our user can define their own extract and transform function to support other types of data and queries. We also theoretically prove that our system has very limited information leakage.

Next, we focus on the problem of performing data analytics on encrypted data using trusted processors. We observe that trusted processors have some sever shortcomings, such as memory access leakage, a very small amount of memory, and no out-of-the-box secure multi-party computation support. To that end, we propose an efficient and extensible mechanism to process large encrypted datasets using trusted processors, such as Intel SGX. 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.

Finally, we utilize a trusted processor to build a secure search index in the cloud. We propose algorithms to build a search index for text and image in data oblivious manner to reduce information leakage due to memory access. Our text index can support ranked document retrieval using TF-IDF scoring. Our system is extensible enough to support user preferred frequency normalization mechanisms. We also compare our results with the industry-standard information retrieval system and observe that our system can produce high quality ranked search results. In addition, to improve performance we defined a non-recursive version of the bitonic sort algorithm for arbitrary length input, which allows us to reduce the significant overhead from unnecessary padding. We also build search index to support face recognition queries using eigenface. We propose a data oblivious version Jacobi eigenvector calculation algorithm to build the eigenface index.

REFERENCES

Agarwal, A. (2014). Web vulnerability affecting shared links. `https://blogs.dropbox.com/dropbox/2014/05/web-vulnerability-affecting-shared-links/`.

Agrawal, R., D. Asonov, M. Kantarcioglu, and Y. Li (2006). Sovereign joins. In *22nd International Conference on Data Engineering (ICDE'06)*, pp. 26–26. IEEE.

Amazon (2019). Amazon s3 - object storage built to store and retrieve any amount of data from anywhere. `https://aws.amazon.com/s3/`.

AMPLab, U. B. (2014). Big data benchmark. `https://amplab.cs.berkeley.edu/benchmark/`. Accessed 5/16/2017.

Anati, I., S. Gueron, S. Johnson, and V. 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*, Volume 13.

Andoni, A. and P. Indyk (2008, January). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM 51*(1), 117–122.

ApacheLucene (2019). Apache lucene. `https://lucene.apache.org/`.

ApacheSolor (2019). Apache solor - blazing-fast, open source enterprise search platform built on apache lucene. `https://www.elastic.co/`.

Arasu, A., S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan (2013). Orthogonal security with cipherbase. In *CIDR*. Citeseer.

Arnautov, S., B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, et al. (2016). Scone: Secure linux containers with intel sgx. In *12th USENIX Symp. Operating Systems Design and Implementation*.

Arnold, B. C. (1985). *Pareto distribution*. Wiley Online Library.

Bajaj, S. and R. Sion (2014). Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on 26*(3), 752–765.

Barbosa, M., B. Portela, G. Scerri, and B. Warinschi (2016). Foundations of hardware-based attested computation and application to sgx. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pp. 245–260. IEEE.

Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314. ACM.

Bauman, E. and Z. Lin (2016, December). A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, Trento, Italy.

Baumann, A., M. Peinado, and G. Hunt (2015). Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*(3), 8.

Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal 5*(2), 78–101.

Bindschaedler, V., M. Naveed, X. Pan, X. Wang, and Y. Huang (2015). Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 837–849. ACM.

Bösch, C., P. Hartel, W. Jonker, and A. Peter (2015). A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR) 47*(2), 18.

Box (2019a). Box - a secure platform for content management, workflow, and collaboration. `https://www.box.com/`. Accessed 10/16/2019.

Box (2019b). Box - rate limiting. `https://developer.box.com/reference#rate-limiting`.

BoxCryptor (2019). Boxcryptor - security for your cloud. `https://www.boxcryptor.com/`. Accessed 10/16/2019.

Brenner, S., C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza (2016). Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 16th Annual Middleware Conference (Middleware)*.

Brickell, E. and J. Li (2011). Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2 1*(1), 3–33.

Cash, D., J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner (2014). Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium, NDSS*, Volume 14.

Cash, D., S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner (2013). Highly-scalable searchable symmetric encryption with support for boolean queries. Cryptology ePrint Archive, Report 2013/169. `http://eprint.iacr.org/`.

Chandra, S., V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham (2017, September). Securing data analytics on sgx with randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, Oslo, Norway.

Chatzichristofis, S. and Y. Boutalis (2008, May). Fcth: Fuzzy color and texture histogram - a low level feature for accurate image retrieval. In *Image Analysis for Multimedia Interactive Services, 2008. WIAMIS '08. Ninth International Workshop on*, pp. 191–196.

Chatzichristofis, S. and Y. Boutalis (2010). Content based radiology image retrieval using a fuzzy rule based scalable composite descriptor. *Multimedia Tools and Applications 46*(2-3), 493–519.

Chatzichristofis, S., Y. Boutalis, and M. Lux (2009, Aug). Img(rummager): An interactive content based image retrieval system. In *Similarity Search and Applications, 2009. SISAP '09. Second International Workshop on*, pp. 151–153.

Chatzichristofis, S. A., K. Zagoris, Y. S. Boutalis, and N. Papamarkos (2010). Accurate image retrieval based on compact composite descriptors and relevance feedback information. *International Journal of Pattern Recognition and Artificial Intelligence 24*(02), 207–244.

Christopher, T. W. (2019). Bitonic sort. `https://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm`. Accessed 10/28/2019.

Christopher D. Manning, P. R. and H. Schtze (2008). *Introduction to Information Retrieval.* Cambridge University Press.

Costan, V. and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

Curtmola, R., J. Garay, S. Kamara, and R. Ostrovsky (2006). Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 79–88. ACM.

Deahl, D. (2017). Verizon partner data breach exposes millions of customer records. `https://www.theverge.com/2017/7/12/15962520/verizon-nice-systems-data-breach-exposes-millions-customer-records`. Accessed 11/18/2019.

Dinh, T. T. A., P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang (2015). M2r: Enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*, pp. 447–462.

Dropbox (2019a). Dropbox. `https://www.dropbox.com`. Accessed 10/16/2019.

Dropbox (2019b). Dropbox - core api best practices. `https://www.dropbox.com/developers/core/bestpractices`.

Dua, D. and C. Graff (2017). UCI machine learning repository.

Duncan, A. J., S. Creese, and M. Goldsmith (2012). Insider attacks in cloud computing. In *2012 IEEE 11th international conference on trust, security and privacy in computing and communications*, pp. 857–862. IEEE.

Dworkin, M. (2007, November). 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.

Elastic (2019). Elasticsearch - open source search and analytics. `https://www.elastic.co/`.

Elmasri, R. (2008). *Fundamentals of database systems*. Pearson Education India.

Faber, S., S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner (2015). *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, Chapter Rich Queries on Encrypted Data: Beyond Exact Matches, pp. 123–145. Cham: Springer International Publishing.

Fernandes, K., P. Vinagre, and P. Cortez (2015). A proactive intelligent decision support system for predicting the popularity of online news. In *Progress in Artificial Intelligence*, pp. 535–546. Springer.

Fu, Y., E. Bauman, R. Quinonez, and Z. Lin (2017, September). 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.

Fuhry, B., R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi (2017). Hardidx: Practical and secure index with sgx. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 386–408. Springer.

Goldreich, O. and R. Ostrovsky (1996, May). Software protection and simulation on oblivious rams. *J. ACM 43*(3), 431–473.

Golub, G. H. and H. A. Van der Vorst (2001). Eigenvalue computation in the 20th century. In *Numerical analysis: historical developments in the 20th century*, pp. 209–239. Elsevier.

Google (2019a). Google drive - a safe place for all your files. `http://drive.google.com/`. Accessed 10/16/2019.

Google (2019b). Google drive api - resolve errors - user rate limit exceeded. `https://developers.google.com/drive/api/v3/handle-errors#resolve_a_403_error_user_rate_limit_exceeded`.

Gupta, D., B. Mood, J. Feigenbaum, K. Butler, and P. 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.*

Hamblin, C. L. (1962). Translation to and from polish notation. *The Computer Journal 5*(3), 210–213.

Hohn, F. E. (2013). *Elementary matrix algebra.* Courier Corporation.

Indyk, P. and R. Motwani (1998). Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, New York, NY, USA, pp. 604–613. ACM.

Intel (2015, October). Product change notification - 114074 - 00. `https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf`. Accessed 5/16/2017.

Islam, M. S., M. Kuzu, and M. Kantarcioglu (2012). Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, Volume 20, pp. 12.

JEITA (2002, April). Exchangeable image file format for digital still cameras. `http://www.exif.org/Exif2-2.PDF`.

Kamara, S. and C. Papamanthou (2013). Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pp. 258–274. Springer.

Kamara, S., C. Papamanthou, and T. Roeder (2012). Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, New York, NY, USA, pp. 965–976. ACM.

King, J. and R. Magoulas (2016, September). 2016 data science salary survey. `http://www.oreilly.com/data/free/2016-data-science-salary-survey.csp`.

Klimt, B. and Y. Yang (2004). The enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pp. 217–226. Springer.

Krandle, V., E. Bauman, Z. Lin, and L. Khan (2017, April). Securing system logs with sgx. In *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security*, Abu Dhabi, UAE.

Kuzu, M., M. S. Islam, and M. Kantarcioglu (2012). Efficient similarity search over encrypted data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pp. 1156–1167. IEEE.

Lai, T. L., H. Robbins, and C. Z. 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), 3034.

Leskovec, J., D. Huttenlocher, and J. Kleinberg (2010). Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 1361–1370. ACM.

Leskovec, J., J. Kleinberg, and C. Faloutsos (2007). Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD) 1*(1), 2.

Leskovec, J. and A. Krevl (2014, June). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Leskovec, J., K. J. Lang, A. Dasgupta, and M. W. Mahoney (2009). Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*(1), 29–123.

Lienhart, R. and J. Maydt (2002). An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, Volume 1, pp. I–900. IEEE.

Liu, C., X. S. Wang, K. Nayak, Y. Huang, and E. Shi (2015). Oblivm: A programming framework for secure computation. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 359–376. IEEE.

Lu, W., A. Swaminathan, A. L. Varna, and M. Wu (2009). Enabling search over encrypted multimedia databases. In *IS&T/SPIE Electronic Imaging*, pp. 725418–725418. International Society for Optics and Photonics.

Lux, M. and S. A. Chatzichristofis (2008). Lire: Lucene image retrieval: An extensible java cbir library. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, New York, NY, USA, pp. 1085–1088. ACM.

Matlab (2019). Matlab. `https://www.mathworks.com/products/matlab.html`. Accessed 11/8/2019.

McKeen, F., I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. 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*, pp. 10. ACM.

Meek, C., B. Thiesson, and D. Heckerman (2002). The learning-curve sampling method applied to model-based clustering. *Journal of Machine Learning Research 2*, 397.

Meeker, M. (2014, May). Internet trends 2014. `https://www.kleinerperkins.com/perspectives/2014-internet-trends/`.

Mishra, P., R. Poddar, J. Chen, A. Chiesa, and R. A. Popa (2018). Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 279–296. IEEE.

Naveed, M. (2015). The fallacy of composition of oblivious ram and searchable encryption. Technical report, Cryptology ePrint Archive, Report 2015/668.

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

Naveed, M., M. Prabhakaran, and C. A. Gunter (2014). Dynamic searchable encryption via blind storage. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 639–654. IEEE.

Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman (1996). *Applied linear statistical models*, Volume 4. Irwin Chicago.

Noakes, D. (2019). Metadata extractor, lets you access the metadata in digital images and video via a simple api. `https://drewnoakes.com/code/exif/`.

Novet, J. (2018). Amazons cloud business is competing with its customers. `https://www.cnbc.com/2018/11/30/aws-is-competing-with-its-customers.html`. Accessed 11/18/2019.

NumPy (2019). Numpy, the fundamental package for scientific computing with python. `http://www.numpy.org/`. Accessed 11/8/2019.

Octave (2019). Gnu octave. `https://www.gnu.org/software/octave/`. Accessed 11/8/2019.

Ohrimenko, O., F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa (2016). Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, pp. 619–636. USENIX Association.

OpenStreetMap (2019). Nominatim - is a tool to search osm data by name and address. `https://wiki.openstreetmap.org/wiki/Nominatim`.

Ostrovsky, R. (1990). Efficient computation on oblivious rams. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, New York, NY, USA, pp. 514–523. ACM.

Page, L., S. Brin, R. Motwani, and T. Winograd (1999). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

Pandas (2019). Pandas - python data analysis library. `http://pandas.pydata.org/`. Accessed 11/8/2019.

Pass, R., E. Shi, and F. Tramer (2016). Formal abstractions for attested execution secure processors. Cryptology ePrint Archive, Report 2016/1027. `http://eprint.iacr.org/2016/1027`.

Phillips, P. J., H. Moon, S. Rizvi, P. J. Rauss, et al. (2000). The feret evaluation methodology for face-recognition algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 22*(10), 1090–1104.

Phillips, P. J., H. Wechsler, J. Huang, and P. J. Rauss (1998). The feret database and evaluation procedure for face-recognition algorithms. *Image and vision computing 16*(5), 295–306.

Pinkas, B. and T. Reinman (2010). Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*, pp. 502–519. Springer.

Porter, M. F. (2006). An algorithm for suffix stripping. *Program*.

Qin, Z., J. Yan, K. Ren, C. W. Chen, and C. Wang (2014). Towards efficient privacy-preserving image feature extraction in cloud computing. In *Proceedings of the ACM International Conference on Multimedia*, pp. 497–506. ACM.

R (2019). R: The r project for statistical computing. `https://www.r-project.org/`. Accessed 11/8/2019.

Rane, A., C. Lin, and M. Tiwari (2015). Raccoon: closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pp. 431–446.

Raval, N., M. R. Pillutla, P. Bansal, K. Srinathan, and C. Jawahar. Efficient content similarity search on encrypted data using hierarchical index structures.

Schuster, F., M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich (2015). Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 38–54. IEEE.

Seber, G. A. and A. J. Lee (2012). *Linear regression analysis*, Volume 936. John Wiley & Sons.

Shih, M.-W., S. Lee, T. Kim, and M. 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*.

Spark, A. (2019). Apache spark - lightning-fast cluster computing. `http://spark.apache.org/`. Accessed 11/8/2019.

SpiderOak (2019). Spider oak - securing the world's data. `https://spideroak.com/`. Accessed 10/16/2019.

Stadmeyer, K. (2014, June). Google drive update to protect to shared links. `https://security.googleblog.com/2014/06/google-drive-update-to-protect-to.html`.

Stefanov, E. and E. Shi (2013). Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 253–267. IEEE.

Stefanov, E., M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas (2013). Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 299–310. ACM.

Stefanov, E., M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas (2013). Path ORAM: An extremely simple oblivious ram protocol. In *CCS*, pp. 299–310.

Sun, W., R. Zhang, W. Lou, and Y. T. Hou (2018). Rearguard: Secure keyword search using trusted hardware. *IEEE INFORM*.

Thomee, B., D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li (2015). The new data and new challenges in multimedia research. *arXiv preprint arXiv:1503.01817*.

Tu, S., M. F. Kaashoek, S. Madden, and N. Zeldovich (2013). Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, Volume 6, pp. 289–300. VLDB Endowment.

Turk, M. and A. Pentland (1991, Jan). Eigenfaces for recognition. *Cognitive Neuroscience, Journal of 3*(1), 71–86.

van Liesdonk, P., S. Sedghi, J. Doumen, P. Hartel, and W. Jonker (2010). Computationally efficient searchable symmetric encryption. In W. Jonker and M. Petkovi (Eds.), *Secure Data Management*, Volume 6358 of *Lecture Notes in Computer Science*, pp. 87–100. Springer Berlin Heidelberg.

Viola, P. and M. Jones (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, Volume 1, pp. I–511. IEEE.

Weber, M. Frontal face dataset. `http://www.vision.caltech.edu/html-files/archive.html`.

Wikipedia contributors (2019). Bitonic sorter — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=908641033`. [Online; accessed 28-October-2019].

Xia, Z., Y. Zhu, X. Sun, and J. Wang (2013). A similarity search scheme over encrypted cloud images based on secure transformation. *International Journal of Future Generation Communication and Networking 6*(6), 71–80.

Xu, Y., W. Cui, and M. 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, Washington, DC, USA, pp. 640–656. IEEE Computer Society.

Yang, Z., S. Kamata, and A. Ahrary (2009, Nov). Nir: Content based image retrieval on cloud computing. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, Volume 3, pp. 556–559.

Zheng, W., A. Dave, J. Beekman, R. A. Popa, J. Gonzalez, and I. Stoica (2017). Opaque: A data analytics platform with strong security. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA. USENIX Association.

Zookeeper, A. Apache zookeeper. `https://zookeeper.apache.org/`. Accessed 5/16/2017.

BIOGRAPHICAL SKETCH

Fahad Shaon received his Bachelor of Computer Science and Engineering from Bangladesh University of Engineering and Technology in 2011. He joined the doctoral program in Computer Science at The University of Texas at Dallas in 2012. His research interests are secure cloud computing and privacy-preserving data mining. Since 2017, he has been working as lead software developer in Data Security Technologies, LLC, where he develops cutting edge big data security products. In addition, he also serves as the principal investigator of the National Science Foundation (NSF) Small Business Innovation Research (SBIR) grants for the company.

CURRICULUM VITAE

Fahad Shaon

October 19, 2019

Educational History:

B.S.Engg., Computer Science and Engineering, Bangladesh University of Engineering and Technologies, 2011
M.S., Computer Science, The University of Texas at Dallas, 2019

Employment History:

Lead Software Developer, Data Security Technologies LLC, Richardson, TX-75083 January 2017 – present
Software Development Engineer Inter, AWS, Amazon Corporate LLC, Seattle, WA, June 2016 – August 2016
Research Assistant, The University of Texas at Dallas, Computer Science Department, Richardson, TX-75080 June 2013 – December 2016
Teaching Assistant, The University of Texas at Dallas, Computer Science Department, Richardson, TX-75080 August 2012 – May 2013

Grants:

NSB Award #1647681, SBIR Phase I, PrivateMR: A Security, Privacy and Governance Policy Enforcement Framework for Big Data
NSF Award #1758628, SBIR Phase II, Secure DL: A Security, Privacy and Governance Policy Enforcement Framework for Big Data