SECURITY-AWARE DATA MANAGEMENT AND PERFORMANCE OPTIMIZATION STRATEGIES FOR CLOUD STORAGE SYSTEMS

KANG SEUNGMIN

NATIONAL UNIVERSITY OF SINGAPORE

2016

SECURITY-AWARE DATA MANAGEMENT AND PERFORMANCE OPTIMIZATION STRATEGIES FOR CLOUD STORAGE SYSTEMS

KANG SEUNGMIN

(Msc., Seoul National University)

A THESIS SUBMITTED

FOR THE DEGREE OF Ph.D. OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2016

DECLARATION

I hereby declare that the thesis is my original work
and it has been written by me in its entirety.
I have duly acknowledged all the sources of information,
which have been used in the thesis.

This thesis has also not been submitted for any degree
in any university previously.

Kang Seungmin

03 May 2016

Acknowledgements

First and foremost I would like to thank my supervisor, Professor Bharadwaj Veeravalli, and co-supervisor, Dr. Khin Mi Mi Aung, for their guidance, help and support. It has been an honor to be their Ph.D. student and I appreciate all their contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating.

Professor Bharadwaj who has guided me the research of load scheduling in clouds encouraged me to not only grow as an applied researcher but also as an independent thinker during my graduate studies at NUS. Dr. Mi Mi who has introduced me to the research of security domain made me have an enthusiastic attitude towards this research direction even during tough times in the Ph.D. pursuit. I would also like to thank members of my thesis committee and anonymous reviewers for their insightful comments.

I am especially grateful for Dr. Tram who helped me to pursue my research with useful discussions. My time at NUS and Singapore was made enjoyable in large part thanks to many friends that became a part of my life. I am grateful for my friends who have spent time with me including Sanghoon, Seunghyun, Lisa, Minjeong, Kangmo, Miles, Hanjoon, Wanhee, Yoonjoo, etc.

Lastly, I especially thank my family for all their love, encouragement and support. My parents have sacrificed their lives for my brother and myself with unconditional love. I love them so much, and I would not have made it without them. I also thank my wonderful brother who is the best friend in my life.

Contents

Summary

The emergence of information technology have changed the scale and scope of information systems that generate a huge amount of data everyday, opening the area of big data processing and analytics. Public clouds have become an attractive candidate to meet not only the ever-growing data storage demands but also the heavy and large-scale computation requirements of big data applications. However, moving data to public clouds for storing and processing raises new challenges that cloud users may have not experienced when they manage the data in their own local servers. The performance issue such as the delay in data retrieval or processing is one of the most important issues since it directly affects the users experience on the quality of services offered by cloud providers. The security issue, particularly the data privacy, is also hindering the migration of data to public clouds. However, it is very difficult to achieve the above challenges due to the complexity in architecture of cloud infrastructures, heterogeneity of cloud resources, and the multi-tenant characteristic of cloud environment. Thus, new tools and models are needed to adapt to the diversity of cloud infrastructures and usage.

Focusing on divisible loads, which are widely used in many large-scale and data intensive applications such as monitoring systems, health care systems and smart home, etc., this thesis applies Divisible Load Theory to propose novel data management solutions including scheduling strategies for data processing and data placement strategies, considering the security requirement. Two scheduling strategies: a static scheduling strategy (SSS) and a dynamic scheduling strategy (DSS) have been designed for divisible load scheduling in multi-cloud systems such that the total data processing time is minimized. The proposed strategies take into account the topology and capacity of the system network and the heterogeneity of computing nodes. While SSS considers an ideal scenario where node availability is known prior to the scheduling, DSS relaxes this assumption and predicts node availability based on historical logging information. Furthermore, the thesis proposes a novel data placement algorithm, namely availability and security-aware data placement algorithm for cloud storage systems (A-SEDuLOUS) that minimizes the total data retrieval time and satisfies the security requirement by applying the graph theory. In addition, the thesis also considers other alternative approaches such as encryption techniques to protect data privacy when storing and processing data on public clouds.

The performance studies presented in this thesis were mainly carried out by numerical simulations to demonstrate the effectiveness of the proposed strategies. We additionally consider a real genomic application, which imposes both the performance and security issues, to demonstrate the practicality of the proposed approaches. We designed an entire secure framework for genomic computation on public clouds to exploit the parallel processing on multiple computing nodes so as to improve the performance. We concretized the framework and proposed a 3-encryption-scheme model for genomic sequence mapping (3EGSM) by combining key-hash function, homomorphic encryption and order-preserving encryption. The model not only protects genomic sequences, the intermediate and final computation results but also eliminates as much as possible the heavy computation requirement of fully homomorphic encryption. The simulation and experimental results assess the validity of the proposed strategies against baseline strategies. The results also provide useful insights on their applicability in realistic scenarios.

List of Tables

List of Figures

Chapter 1

Introduction

The advent of cloud computing has changed the way of using computer and satisfying IT demands of both industrial and academic organizations. Cloud computing is capable of elastically scaling services and infrastructures as well as self-adaptively managing the offered services, thus efficient service provisioning can be obtained to improve not only the quality of service for users' applications but also the economic benefit for cloud providers [Moreno-Vozmediano et al., 2013]. Many commercial cloud providers have joined the market and they are offering users cloud resources including computing, storage and network resources under different service models such as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [Sen, 2013] with a pay-per-use model for the resource usage cost. Due the virtualization technologies, multiple users' resource requests can be co-located on the same set of physical resources without any performance interference, making cloud computing be a multi-tenant computing environment. The flexibility in resource management, high quality in service provisioning with performance isolation and low resource usage cost make clouds become an attractive candidate for many large-scale, data-intensive and computing-intensive applications. Many applications, which were run in dedicated in-house servers, are nowadays migrated to public clouds for handling a big amount of data generated everyday such as surveillance systems, health care systems, smart home or environment monitor [Zeng et al., 2016], heralding the area of *big data* processing in the clouds. Throughout this thesis, we use "users" to indicate the actors who rent resources or services from cloud providers, and "providers" to indicate the actors who lease the cloud resources or services.

Considering IaaS clouds where users request resources as a bundle of computing, network and storage resources to deploy their own computing platform, this thesis carried out the studies to provide users efficient methods to optimize the performance of their systems, taking into account the privacy issue of the data stored and processed in their systems. Since users are responsible for managing their own computing platform deployed in the clouds, the performance of the resource manager used in their systems directly affects the overall performance of the systems, e.g., the total processing time of the data, load balancing among computing nodes. Having an efficient resource manager helps users reduce total resource usage time, thereby reducing the resource usage cost paid for providers. However, with complex resource requirements of modern big data applications and the complexity in the architecture of the computing platforms deployed to run such applications, the existing approaches may no longer guarantee the efficiency and may not consider all the features of a cloud computing platform.

In addition to the performance issue, users also concern about the data privacy issue when moving data to the clouds for storage and processing. As mentioned earlier, due to the multi-tenant characteristic of the cloud environment, preventing the data leakage and loss becomes harder and exceeds the capacity of the current protection standards such as data anonymization or differential privacy. Adversaries on the clouds being either honest-but-curious or malicious can collect the leaked data and infer useful information for their own purposes. The resource manager, which assigns data to computing nodes for processing, or the data placement manager, which assigns the data to storage nodes, need to take into account the security requirement to protect the data privacy. This thesis therefore aims at providing cloud users the methods for big data management in cloud storage systems for not only optimizing the overall performance but also considering the data privacy.

1.1 Key Challenges of Big Data Processing in Clouds

In this section, we analyze the key challenges of big data processing in clouds to motivate the research carried out in this thesis. Even though many existing works have addressed these challenges, they did not either provide a complete solution or taking into account sufficient parameters of the systems. Focusing on the performance issue while considering the security requirement, we show that new methods are needed to achieve high performance in data processing in the clouds while protecting the data privacy.

1.1.1 Achieving High Performance for Big Data Processing in Clouds

Due to the flexibility in resource provisioning in clouds, the amount of resources requested by users to deploy their own systems can be elastically adjusted to satisfy the computing requirement and the volume of data needed to be processed, thus obtaining the desired performance and reducing the resource usage cost. Generally, we can refer the amount of work to process data on a computing node as a *load* and hereafter, we use the terms "data" and "loads" interchangeably. Since computing resources are used for processing loads, how to efficiently schedule/assign loads to computing resources is a difficult problem to achieve high performance of the system, e.g., minimize the *total processing time of loads*, which in turn can be obtained only when load balancing among computing nodes is guaranteed [Fang et al., 2010]. Indeed, load balancing among computing nodes will avoid a scenario that one node may be assigned very heavy load while other nodes process smaller loads and stay idle after finishing. However, the load balancing becomes much more complicated since computing nodes are practically heterogeneous in terms of computing capacity, network link capacity and other aspects such as software licence, security limitation. Furthermore, due to the nature of loads submitted to the system, loads may require different amount of computations. For example, one load deals with data compression and the other deals with data encryption using different key lengths, resulting in different amount of running time the computing node spends to accomplish the processing. Throughout this thesis, we refer to this characteristic as *computation requirement* of a load. Combination of all the above characteristics makes the problem of load scheduling become challenging that has not been considered in the literature. We step forward to propose novel scheduling strategies in this thesis.

In general, there are two types of load scheduling approaches: static and dynamic. Static scheduling approach assigns the loads to computing nodes with the assumption that the information of the whole system is available prior to the scheduling process such as computation capacity of the nodes and the time instant that computing nodes are available for processing loads, denoted as *release time* of computing nodes. However, having only prior knowledge of nodes does not reflect dynamic changes of node attributes at run-time, e.g., node crashes and becomes unavailable for processing other loads that have been assigned to it. Furthermore, static approach is not able to adapt to load changes during run-time [Nuaimi et al., 2012]. Therefore, this scheme has a limitation when applying to realistic scenarios. On the other hand, dynamic load scheduling approach considers the dynamic changes in the system such as the number of computing nodes available for processing at the scheduling instant, number of loads remaining to be processed on each computing nodes. Computing nodes may be released in the past but they may not be ready for processing a load since they are busy with others, referred to as *ready time* of computing nodes. Since this approach collects run-time properties of nodes, it can dynamically assign the loads to computing nodes based on up-to-date information of nodes and loads [Nuaimi et al., 2012] and immediately react to the change of the system. For dynamic load scheduling, it is necessary to have a monitoring mechanism in order to observe the node status and load progress. The monitoring mechanism collects information about the state of nodes such as CPU load, running processes, bandwidth consumption, etc. Although it is difficult to implement compared to the static approach, dynamic approach is more accurate and appropriate to apply to large-scale cloud computing infrastructures.

1.1.2 Protection of Data Privacy in Clouds

Although cloud storage systems have become an attractive candidate to meet the ever-growing storage demands from users, ensuring the privacy of data stored in the cloud is major challenge due to the multi-tenant characteristic of the cloud environment [Factor et al., 2013, Huang et al., 2011b, Hao and Han, 2011, Itani et al., 2009]. Without the knowledge of physical location of the data, users concern about the data leakage and loss, which may result in severe consequence for the data owner [IMEX, 2010] for instance users may be denied to insurance services because of leakage of personal medical records. In general, data stored in cloud storage systems such as Amazon Simple Storage Service (Amazon S3) [Amazon S3, 2014] and Google Cloud Storage [Google, 2014] are remained in plaintext by default. However, insecure storage allows malicious users to be able to access and exploit the stored sensitive data [Subashini and Kavitha, 2011]. To protect the data privacy, several methods have been proposed: (i) user-centric authentication, (ii) encryption based approach, and (iii) division of sensitive information. User-centric authentication approaches verify and confirm the identity of users who try to access data and password-based authentication scheme is the most widely used by cloud providers [Moreno-Vozmediano et al., 2013]. However, only using such user-centric authentication service may not be sufficient to solve the problem of data privacy protection due to authentication attacks.

Encryption emerged as one of the most effective means to protect sensitive data by making the original data called plaintext unreadable [Harrin, 2012]. This is done by using an encryption algorithm, which encodes plaintext to generate the encrypted data called *ciphertext* so that it can be read by someone who has legitimate encryption key. With an encryption tool, users can encrypt data on their local machine before uploading the encrypted data to the cloud. However, this approach introduces an additional burden for users who may not be expert to manage the encryption key and operate the encryption tool. Furthermore, users are required to equip local machines, which need to be powerful to be able to handle such a compute-intensive task. These issues make the user-side encryption approach difficult to realize in reality. Thus, a server-side encryption approach is advocated such that cloud providers offer users such encryption service as an added value or free of charge service. However, the server-side approach requires that users trust and delegate the encryption process to providers.

While conventional encryption algorithms such as AES and Blowfish can protect the privacy of the data stored in the clouds, the applications taking this data as an input are not able to perform the computation with the data encrypted by such algorithms. In other words, the conventional encryption algorithms can protect only the data stored in the clouds for backup purpose but not involving in any computation. To protect the privacy of the data involving in computation, *Homomorphic Encryption* (HE) [Gentry, 2009], which has been extensively studied in the past few years, can be applied since it enables complex mathematical operations to be performed on encrypted numeric data without decrypting and any knowledge of the secret decryption key. The intermediate and final results of the computation are also generated under encrypted format so as only the users who have the legitimate secret key can decrypt and obtain the plaintext data. This feature makes homomorphic encryption to be used for protecting the privacy of numeric data that involves in the computation performed in untrusted domains without revealing the original plaintext data. However, most of homomorphic encryption schemes require heavy-weight computation, leading to the performance degradation [Chung et al., 2010]. We therefore study to apply different encryption techniques to respective data depending on the computation characteristic of the data to optimize the overall performance of the system.

In addition to encryption techniques, assuming that sensitive information is spread over the data file, a division approach can be used to divide the data into multiple fragments (chunks), each will be stored or processed on a separate storage or computing node. This will protect the privacy of the data since even a chunk has been attacked, malicious users cannot guess the location of other chunks in a large-scale system. Depending on the *security requirement* of the users, the distance between the chunks of the same data will be defined such as by the number of hops on the network path connecting two nodes. The further the distance is, the higher the security level of the data that can be guaranteed but the lower the performance of the system, e.g., the total retrieval time of the data is longer. Furthermore, determining the size of the chunks stored or processed on each node directly affects the performance since nodes are heterogeneous in terms of capacity. Thus, deciding a data placement solution (a list of storage or computing nodes and the size of chunks assigned to respective nodes) is very challenging to satisfy the security requirement while not sacrificing the overall performance. Motivated by this, we carry out the research in this thesis to apply not only encryption techniques but also the data division technique to protect the data privacy and evaluate the performance of each approach in different scenarios.

1.1.3 Guaranteeing Data Availability in Clouds

Not only being concerned about the privacy issue, users are also concerned about the availability of the data stored in the clouds [Hashem et al., 2015]. While one of the main purposes of using cloud storage systems is to guarantee the data availability, the large-scale characteristic of cloud storage systems leads to frequent hardware failures, making data unavailable. For instance, the T-Mobile/Sidekick incident [Wingfield, 2009] in 2011 experienced temporary lack of availability lasting at least several hours [Helft, 2009, Dignan, 2008] and noticeable loss of personal customer data. To ensure data avail-

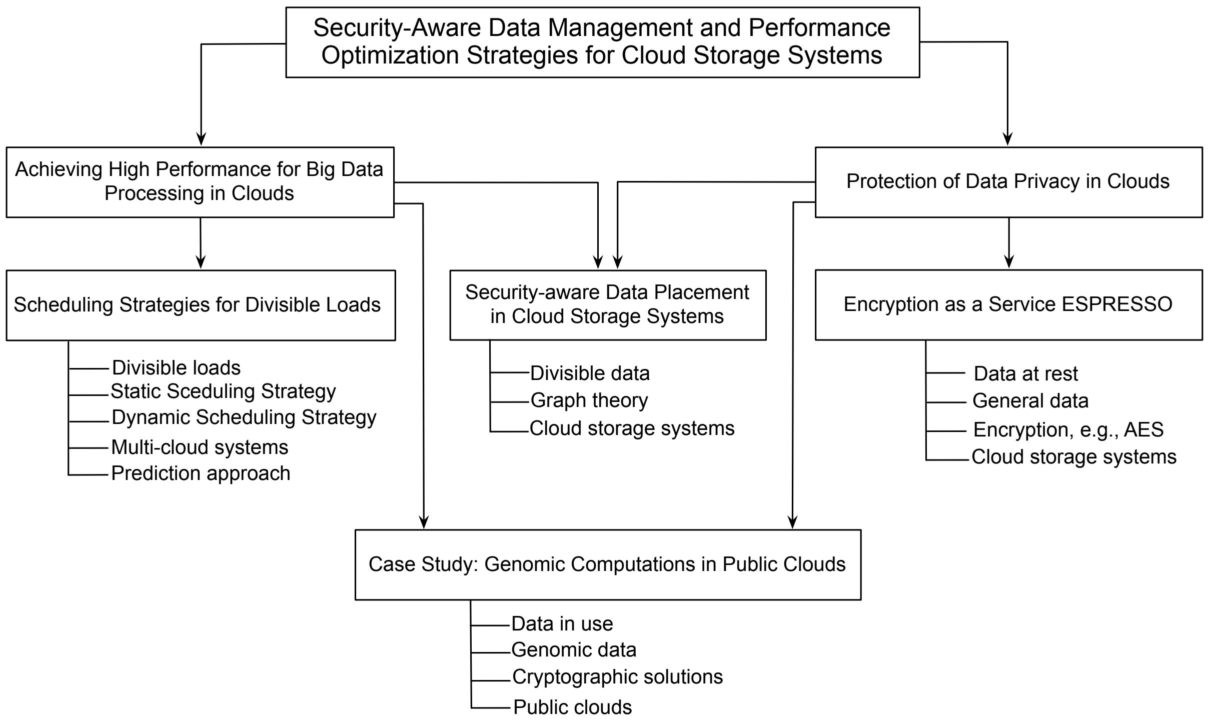

Security-Aware Data Management and Performance Optimization Strategies for Cloud Storage Systems
Achieving High Performance for Big Data Processing in Clouds
Protection of Data Privacy in Clouds
Scheduling Strategies for Divisible Loads
Divisible loads
Static Sceduling Strategy
Dynamic Scheduling Strategy
Multi-cloud systems
Prediction approach
Security-aware Data Placement in Cloud Storage Systems
Divisible data
Graph theory
Cloud storage systems
Encryption as a Service ESPRESSO
Data at rest
General data
Encryption, e.g., AES
Cloud storage systems
Case Study: Genomic Computations in Public Clouds
Data in use
Genomic data
Cryptographic solutions
Public clouds

Figure 1.1: Contributions of thesis and relationship between works.

ability, many cloud providers offer the data replication as a transparent service with multiple data copies that are stored geographically. However, some of them still suffer the system outage such as Google Mail and Hotmail [Albanesius, 2012, Cachin et al., 2009]. This shows that only increasing the number of data copies without considering other aspects such as storage node selections based on network topology will not guarantee the data availability completely when network links are attacked. In this thesis, we first revise the existing approaches and then propose a solid data placement solution to not only guarantee the complete data availability, protect the data privacy but also improve the overall performance of the system by minimizing the total retrieval time of a data.

1.2 Contributions and Organization of the Thesis

1.2.1 Scope and Contributions of the Thesis

This thesis aims at providing efficient methods for the users of IaaS clouds to manage their own computing platform built up by using the resources reserved from IaaS clouds including storage, network and computing resources. The scope of the thesis is limited to divisible loads that are assumed to be divided into multiple chunks with arbitrary size such as text files, video files and genomic sequences. However, we believe that the contributions made in this thesis can apply to a large category of cloud applications that require both high performance for processing large volume of data and the security requirement for protecting the data privacy. In summary, Fig. 1.1 presents the contributions of this thesis and the relationship between the works carried out in this thesis. We focus on the two main research directions that directly affect the user's perception on cloud infrastructures: the overall performance of the systems and the data privacy. We first carry out different research works on each direction separately to provide efficient methods to the users who may have interest in only one of the directions, i.e., the user objective is only achieving high performance of the systems or protecting the data privacy. We then combine both objectives in a joint work and algorithmic contributions and demonstrate the effectiveness of the proposed approach by real experiments of a specific case study.

On the direction of performance optimization, we focus on the scheduling strategies for divisible loads in multi-cloud systems. We propose novel scheduling strategies that take into account the complex requirements of users on computing resources, network topology. They consider the dynamic arrival of loads as well as the availability of computing nodes in a heterogeneous computing environment. The proposed scheduling strategies aim at minimizing the total processing time of loads so as users can reduce the resource usage cost paid for cloud providers. On the direction of data privacy, we design and implement a server-side encryption service for cloud storage systems (ESPRESSO). Though this work applies existing encryption techniques, its flexibility design allows users to enhance the security of their cloud storage systems and thus relieving the concern on data leakage and loss.

In several scenarios, users are requiring to achieve both objectives at the same time. We thus consider the problem of performance optimization while satisfying the security requirement. As a theoretical and algorithmic contribution, we propose a novel approach for data placement in cloud storage systems to meet the increasing demands of nowadays big data applications. The proposed approach applies the divisible load theory to address the performance issue and graph theory for the security issue. As a practical contribution, we design an entire secure framework for genomic data processing on public clouds. We implement the framework for a specific genomic application, genomic sequence mapping, since it is an important step in genomic computation and it handle large amount of data. The propose framework achieves high performance by exploiting parallel computation on different computing nodes and protects the data privacy by using advanced encryption technique, homomorphic encryption.

1.2.2 Organizations of the Thesis

The rest of the thesis is organized as follows.

Chapter 2. Chapter 2 describes the state of the art for load scheduling and the protection of data privacy and availability in clouds. Specifically, we first present divisible load paradigm and existing approaches for scheduling divisible loads. Then we review existing data privacy protection mechanisms not only for securing the data stored for backup purpose but also for securing the data involving in computations in clouds. We also analyze the drawbacks of the existing approaches and show that they are no longer applicable to new context and requirements. Through the analysis of the state of the art, we present our motivation for the study made in this thesis.

Chapter 3. In this chapter, we present our contribution on the load scheduling problem in multi-cloud systems [Kang et al., 2014b, Kang et al., 2016c]. We first design two architectures for a multi-cloud system: a centralized scheduling architecture and distributed scheduling architecture. Based on theses architectures, we then develop two scheduling strategies: a Static Scheduling Strategy (SSS), which assumes that the release time of computing nodes is known prior to the scheduling; and a Dynamic Scheduling Strategy (DSS), which applies a prediction technique to estimate the release time and ready time of computing nodes. Both strategies consider the topology and capacity of the system network, and the heterogeneity of computing nodes. The proposed scheduling strategies were evaluated through comprehensive simulations and the results provide useful insights on the applicability of the proposed architecture and scheduling strategies.

Chapter 4. In this chapter, we present a novel approach for data placement in cloud storage systems considering the performance, data privacy and data availability [Kang et al., 2016b, Kang et al., 2016d]. We first develop an optimization programming formulation for the problem and then an efficient heuristic algorithm, which divides a data into multiple chunks, each will be stored in two nodes as primary and backup nodes. The placement decision is computed in the way that all the primary and backup chunks of a data satisfy the security and availability requirements, and the total retrieval time is minimized. We demonstrate the effectiveness of the proposed algorithm through comprehensive simulations and the results show that the proposed algorithm significantly reduces the retrieval time.

Chapter 5. In this chapter, we design and implement an encryption service namely ESPRESSO (Encryption as a Service for Cloud Storage Systems) to protect the users' data by using advanced encryption algorithms [Kang et al., 2014a]. With the flexible design, cloud service providers can choose the encryption algorithm based on their preference, and users can specify the security level of their data. The data with higher critical level needs to be more securely protected with a longer encryption key.In addition, our design allows cloud service providers to easily integrate ESPRESSO into their systems without heavy modification and implementation of their systems during the integration. The real experiments were conducted and the experimental results assess the performance and effectiveness of ESPRESSO.

Chapter 6. Chapter 6 presents a case study as a proof of concept for all the theoretical contributions made in this thesis [Kang et al., 2016a]. We design an entire secure framework for big data processing on public clouds to achieve data privacy, scalability and performance. We choose a genomic application as a case study since it can be considered as big data application processing huge volume of data and heavy computation. And genomic application also requires a high security level to protect the privacy of the owner of medical data. Based on this framework, we also propose a 3-encryption-scheme model for genomic sequence mapping (3EGSM), which is an initial but important step to process genomic sequences before they are further analyzed in other domain-specific genomic applications . The model protects not only genomic sequences but also the intermediate and final computation results when processing on public clouds. We implement all the steps to interact with clouds and evaluate the proposed framework through intensive experiments using real genomic data. The proposed framework reduces the total processing time by delegating the most compute-intensive tasks to the clouds without any concern about the privacy of input data as well as computation results.

Chapter 2

Literature Background

In this chapter, we introduce existing approaches to address the problems that are in the scope of the thesis to motivate the forthcoming study of this thesis: load scheduling and the protection of data privacy in clouds. For load scheduling, we describe Divisible Load Theory that is mainly used for the efficient load scheduling strategy in this thesis. Then we introduce the literatures for big data scheduling using divisible load theory. For the protection of data privacy, we mainly focus on reviewing the existing approaches for secure data storage and secure computation in clouds. We additionally discuss the works related to genomic computation in clouds that will be used as a case study in the thesis.

2.1 Load Scheduling in Clouds

2.1.1 Divisible Load Theory

Divisible Load Theory (DLT) has been firstly introduced in [Cheng and Robertazzi, 1988]. It assumes that loads (input data) can be perfectly divided into an arbitrary number of chunks with different sizes, namely *divisible loads*. In DLT, each load fraction (chunk) can be independently processed if there is no precedence relations among them. Indeed, many applications that need to process big data satisfy this divisibility property including many streaming data applications such as monitoring systems, continuous write applications as shown in [Zeng et al., 2016].

The advent of clouds as a new model of service provisioning encourages researchers to investigate the use of DLT to design efficient strategies for scheduling loads. Indeed, DLT has contributed to achieve minimizing the processing time of loads compared to other approaches. For example, MapReduce that is a programming model for large-scale data processing splits the data into the identical size of small fractions, i.e., 64 MB [Dean and Ghemawat, 2004]. However, such equal division affects the performance degradation if computing nodes, corresponding to data nodes in MapReduce, have heterogenous computation capacities. On the other hand, DLT enables multiple load chunks of a load, each has different size, to be processed on different nodes by assigning load chunks according to the heterogenous capacities.

Indeed, this achieves reducing the total processing time by assigning different size of load chunks to heterogenous computing nodes. Hence, beyond the existing approaches, which consider the computing capacity of nodes, the size of loads and the computation requirement of loads, it is necessary to apply the DLT. In the following subsection, we will introduce existing algorithms to address the problem of scheduling divisible loads.

2.1.2 Scheduling Strategies for Divisible Loads

Big data processing emerges as an important part of applications and many studies [Zhang et al., 2012, Guo et al., 2012, Mo and Wang, 2012, Jung et al., 2012] are emerging nowadays to explore the possibility of using cloud computing paradigm for big data processing. Those works are driven by a fact that big data processing requires scalable and parallel computing resources rather than using traditional data processing applications [Ji et al., 2012]. Therefore, application of DLT to the problem of scheduling big data on clouds is ideal for concurrent processing with high performance.

To achieve high performance and high resource utilization, there have been many studies for scheduling multiple divisible loads on distributed computing system. In [Drozdowski and Lawenda, 2008], the authors presented a scheduling solution for multiple divisible loads on homogeneous computing nodes, i.e., all the nodes have identical computing capacity, which are connected to the sole load source as a star network. In a star network, there is one node called originator in the center of the star and the communication links only the originator with the remaining node. However, this model as well as homogeneous computing nodes do not reflect a real environment. In [Marchal et al., 2006], the authors presented a model for wide area network that extends over a large geographical distance and proposed a new model for deploying and scheduling multiple divisible load applications on large-scale computing platforms. However, these works considered that there is only one load source, which can distribute loads to all computing nodes. This assumption is not practical in the cloud environment and modern system architectures. In such scenarios, loads may arrive from different sources and computing nodes may not be able to process all types of loads due to the problem of quality of service and software license. Furthermore, to guarantee the quality of service or the deadline constraint, the multi-cloud system must use dedicated links with stable bandwidth. Such links may not be established from the load sources to all computing nodes. Yet, computing nodes may have different release times, i.e., they are available for computation at different moments.

Considering the release times of computing nodes, several previous works have studied this issue. In [Lin et al., 2007], the authors presented a real-time divisible load scheduling with different processors' available times. In [Veeravalli and Wong, 2004], the authors proposed efficient load distribution strategies for scheduling divisible loads to minimize the total processing time of the entire load submitted for processing. However, both works assumed that release times of computing nodes are predetermined and known prior to the start of the scheduling process. This may not be satisfied in a real-life scenario, e.g., nodes are unavailable for computing unexpectedly. To reflect such dynamic situation, we need to check respective node's availability periodically for scheduling divisible loads. Based on this motivation, we advocate for a dynamic scheduling problem in clouds for processing big data with high performance. In addition to the performance issue, protecting big data privacy for storage and computation is also our focus in this thesis as described in Chapter 1. In the following section, we describe existing approaches for both secure data storage and secure data computation in clouds.

2.2 Protection of Data Privacy in Clouds

2.2.1 Secure Data Storage

Most of literatures on data encryption have focused on providing a user-side encryption tool, which allows the owner to share his data with different consumers. [Kang and Zhang, 2010] proposed an identity-based authentication (IBA) scheme by which the owner can share his encrypted data stored in the cloud. This scheme divides the sharing users into the same domain and a user only can share another user's data if and only if they are in the same domain. In [Huang et al., 2011b], YI Cloud, a framework for protecting the data privacy in the cloud, is presented. This system allows the users to encrypt their files in the cloud storage and user's primary encryption key is shared between trusted entities using secret sharing algorithm. Secret sharing [Shamir, 1979] distributes a secret among a group of participants, each of them is allocated a share of the secret. The secret can be reconstructed only when a sufficient number of shares are combined together. The framework includes two components: a client component, which is deployed on the user's machine for encryption and key management, and a server component installed on Sector [Gu and Grossman, 2009] for management of users and storage nodes. However, both [Huang et al., 2011b] and [Kang and Zhang, 2010] did not provide the flexibility for providers and users, i.e., choosing encryption algorithm for providers and specifying the critical level of users' data.

In [Zhao et al., 2010], a progressive encryption system has been proposed based on elliptic curve cryptography (ECC). ECC has been introduced as an alternative mechanism for implementing public-key cryptography. Public-key algorithms create a mechanism for sharing keys among large numbers of participants or entities. The proposed scheme allows data to be encrypted multiple times with different keys and produces a final ciphertext that can be decrypted with a single decryption key. Hence, the system allows the owner to share his encrypted data with other consumers without revealing the plaintext data to untrusted entities. The work did not present any real experiment but we believe that this approach involves an intensive computation, thus introduces high latency.

Furthermore, [Zhao et al., 2010] focused on the encryption algorithm and the sharing mechanism while we aim at providing an entire encryption service, which can be adopted by any existing CSP. Similarly, [Yu et al., 2010] proposed a secure and scalable fine-grained data access control scheme for cloud computing by combining key policy attribute-based encryption (KP-ABE) with techniques of proxy re-encryption and lazy re-encryption. KP-ABE [Goyal et al., 2006] is a public key cryptography primitive for one-to-many communications. In KP-ABE, data are associated with attributes for each of which a public key component is defined. The encryptor associates the set of attributes to the message by encrypting it with the corresponding public key components. Proxy re-encryption (PRE) [Blaze et al., 1998] is a cryptographic primitive in which a semi-trusted proxy can convert a ciphertext encrypted under a user's public key into another ciphertext, which can be opened by different user's private key without seeing the underlying plaintext. Both [Yu et al., 2010] and [Zhao et al., 2010] considered a different threat model where users do not trust any third party such as cloud providers. Hence, users must take full responsibility for the data encryption and key management on their local machines.

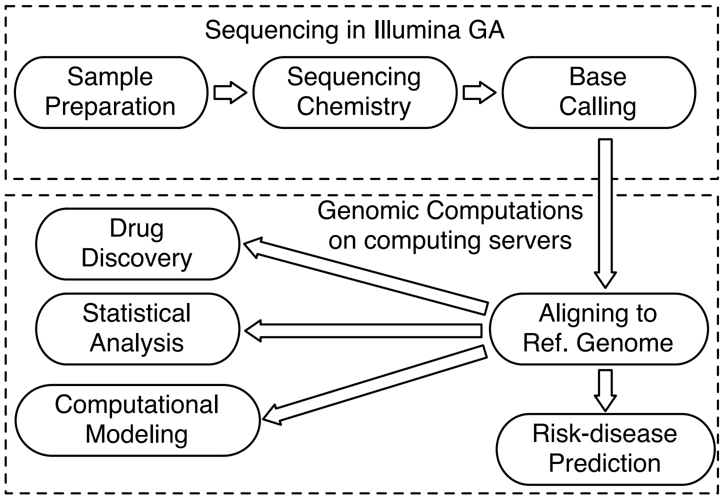


Figure 2.1: Overall genomic computation process.

In [Itani et al., 2009], the authors presented PasS (Privacy as a Service), a set of security protocols for ensuring the privacy of data stored in the cloud. Although a server-side encryption service is presented, this work assumed that the encryption service is maintained by a third party that is trusted by users as well as CSPs. However, the assumption of trusting the third party is not realistic thus we need to propose a security component, which can be integrated in cloud infrastructures. If such component is proposed, cloud providers can increase the reputation and help cloud users alleviate the security concerns with the third party.

2.2.2 Secure Genomic Computation in Clouds

Protection of data privacy in clouds needs to be considered not only data for storage but also data for computation. There are many types of big data processed on clouds but we focus on genomic data in this thesis as a secure data computation. This is because the rapid advances in genomic technologies have changed the scale and scope of genomic data processing. We first present a brief background on genomic computation and then we introduce existing approaches for secure genomic computation in clouds.

2.2.2.1 Genomic Computation

Genomic computation has become a new and active application area of computer science for the past ten years. The explosion of genomic data provides the entire DNA sequences of several organisms including human for life science research. Computer science plays an important role in genomic computation from sequencing and assembling of DNA sequences to analyzing genomes in order to locate SNPs, repeat families, similarities between sequences of different organisms and other applications. We present in Fig. 2.1 the overall genomic computation process with several domain-specific applications, but are not limited to.

In order to make genomic data available for analysis on computer systems, the first step is to sequence, assemble DNA sequences and produce the strands of nucleotides (A, T, C or G). As shown in

Fig. 2.1, with the advent of novel sequencing technologies, this step is now carried out in specific sequencing machines such as Illumina/Solexa [ER, 2008]. The results of this step is an enormous amount of short reads, i.e., millions of short nucleotide sequences, which need to pass by an important stage of aligning to a reference genome, i.e., *read mapping*, before being further analyzed. Read mapping is therefore a fundamental step for most genomic analysis such as SNP discovery, genotyping, and personal genomics [Schuster, 2007]. However, it is also a bottleneck step in genomic processing since it has to process large amount of data with heavy computations, requiring high accuracy. Read mapping involves computing edit distance, which determines the minimum number of edits (substitution, insertion, deletion) to change one sequence into another sequence. The edit distance is one of the most important metrics, which is useful in the genomic research for the diagnosis and treatment of cancer, Alzheimer's disease, Schizophrenia, etc [Network and et al., 2012, Taylor et al., 2001, Waddell et al., 2015].

While the sequencing and assembling stages are handled by particular machines, the development of computer science opens many research directions for the second stage from the aligning step (secondary genomic data analysis) to various genomic analysis applications (tertiary genomic data analysis) as shown in Fig. 2.1. This area includes both application and adoption of older genomic methods to run on different computing infrastructures, and development of novel algorithms for the analysis of genomic sequences. Restrain the problem a bit more to the read mapping problem, we observed that conventional read mapping algorithms have been designed to run sequentially on a single computing unit. This is not scalable when the number of patients increases and clinics are required to finish the process as fast as possible. The algorithms therefore need to be adapted to enable parallel processing on multiple computing units. Furthermore, considering the security issue, particularly the privacy of genomic data, the algorithms also need to be security-aware in the sense that they should not reveal any sensitive information during the execution on untrusted third party. In this thesis, we present our approach to address the above challenges, focusing on the read mapping algorithm.

2.2.2.2 Existing Approaches for Secure Genomic Computation in Clouds

Read mapping is one of the important applications of genomic data processing since it is considered as an initial phase of every genomic application. The well-known read mapping technique is using a hash function to hash genomic sequences of the read, which is defined as the raw genomic data consisting of millions of short nucleotide sequences (25 - 250 bp), and the reference genome. This allows one to perform a fast search and also to reduce the size of input data. Many alignment tools such as BLAST [Altschul et al., 1990], BLAT [Kent, 2002], SOAP [Li et al., 2009], SSAHA [Ning et al., 2001] and RMAP [Smith et al., 2008] are based on this technique. To allow a number of mismatches between the read and the matching segment on the reference genome, these tools have implemented the *seed-and-extend algorithm*, which divides the read and the reference genome into multiple sub-sequences called *seeds* that will be hashed for fast search. Even though these tools use a hash function for hashing genomic sequences, the main purpose of hashing was not to protect the data privacy, but to accelerate the speed of searching and alignment. Indeed, such tools were designed to run on the trusted (private) servers, thus they do not need to consider the security issue. However, the private servers are limited in terms of storage and computational capacities to operate a huge volume of genomic data produced by the next generation sequencing machines nowadays. The existing alignment tools therefore need to be adopted to be able to perform the read mapping using storage and computational resources of public clouds. The adoption is not only in the system design but also in the implementation of mapping algorithm to protect the data privacy for both input data and computed results when moving to the cloud.

To tackle the problem of revealing genomic data processing on public clouds, several studies have proposed to separate the computation of non-sensitive data on public clouds and sensitive data on the private servers [Chen et al., 2012, Wang et al., 2009]. In [Chen et al., 2012], the authors proposed a secure and efficient algorithm to align short sequences to a reference sequence. They implemented the seed-and-extend algorithm and delegated the seeding and extension steps to the public cloud and the private trusted servers, respectively. While this work can protect the privacy of genomic sequences by using a keyed hash function, the matching positions of the seeds on the reference genome are not protected. This allows adversaries on the public cloud to infer the matching position of the read on the reference genome and then infer the study of the clinic. In [Wang et al., 2009], the authors proposed a privacy-protection framework by applying program specialization to distribute a genomic computation between data provider and data consumer based on the sensitivity levels of the genomic data. This approach requires much interactions of the data provider during the processing, which may not be practical in a realistic scenario.

On the other extreme, there also exist several works, which propose to delegate all computations to public clouds. To protect the data privacy, they apply the homomorphic encryption scheme that allows one to perform the arithmetic computation on encrypted data. The trusted servers are responsible for data encryption and keep all secret keys. Several genomic data processing algorithms have been implemented using homomorphic encryption such as edit distance algorithm, Pearson goodness-of-fit test, and Cochran-Armitage test [Lauter et al., 2014]. However, most of homomorphic encryption schemes require heavyweight computation, leading to the performance degradation. As another approach for data privacy protection, secure multi-party computation (SMC) has been used to enable multiple parties to evaluate a function cooperatively without revealing anything beyond the function's output to either party. In [Huang et al., 2011a], the authors presented a generic approach for two-party computation based on the garbled-circuit technique. In order to achieve the security, i.e., not reveal anything except the function's output, SMC requires a complex communication protocol that allows the cooperating parties to exchange data with each other. Compared to homomorphic encryption techniques, the communication overhead of SMC may be worse than the computational overhead of HE, especially when the amount of data needed to be exchanged is large.

When we consider the problem of read mapping in the second part of this thesis, the input data, which is the short read and the reference genome, is provided by the same owner that is the sole party. Public clouds play the role of computing infrastructures but they do not provide any input for the computation. Encryption techniques are therefore an appropriate candidate to protect the data privacy for the read mapping problem when processing on public clouds.

Chapter 3

Scheduling Strategies for Handling Divisible Loads in Multi-Cloud Systems

Focusing on divisible load scheduling in multi-cloud systems, this chapter first proposes two architectures for a multi-cloud system: a centralized scheduling architecture and a distributed scheduling architecture. Both architectures can satisfy complex requirements of users on computing resources, network topology and guaranteed quality of services for their big data applications, and they are used in different realistic scenarios depending on the users' requirements. Based on these architectures, two scheduling strategies are then proposed: (i) A Static Scheduling Strategy (SSS), which assumes that the release time of computing nodes is known prior to the scheduling; and (ii) A Dynamic Scheduling Strategy (DSS) relaxes the assumption of SSS by applying a prediction technique to estimate the release time of computing nodes. Both strategies take into account the topology and capacity of the system network, and the heterogeneity of computing nodes. They were evaluated through comprehensive simulations. The simulation results provide useful insights on the applicability of the proposed architectures and scheduling strategies across a range of realistic scenarios. The study in this chapter is first-of-its-kind in employing divisible load paradigm in multi-cloud scenario and demonstrating its performance.

3.1 Research Motivation and Objectives

Focusing on IaaS clouds where users reserve computing resources represented as virtual machines (VMs), storage space and network bandwidth for building up their own virtual computing infrastructures on public clouds, we study the problem of cloud resource management. On one hand, users have to pay providers for cloud resource usage cost based on the pay-per-use model. On the other hand, they need to efficiently manage such cloud resources to achieve high performance of their applications, improving the utilization of reserved resources, thereby minimizing the usage cost. Reserving large amount of cloud resources may result in good performance of the applications but the resource usage cost will dominate. Thus, intelligent resource management mechanism is needed to efficiently exploit a sufficient amount of cloud resources while satisfying the performance requirement of the applications.

With the emergence of information technology and wireless sensor networks, many complex applications have been migrated to public clouds for running and processing big amount of data. Examples include surveillance systems, health care systems, smart home or environment monitor [Zeng et al., 2016]. Such applications generate a huge volume of data everyday and the data rate is very dynamic over time from different geographically distributed data sources. To handle such high volume complex data, a possible solution for single cloud providers is to scale out their cloud data centers in multiple geographically distributed locations. However, this increases the Total Cost of Ownership (TCO) as well as the maintenance service cost. Thus, the current day trend is towards adopting an infrastructure that spans across multiple clouds and data-centers, referred to as *Cloud-of-Clouds* or *multi-cloud* platforms [Truong-Huu and Tham, 2014]. This allows multiple single cloud providers to cooperate with each other so as to improve the cloud resource utilization and the final revenue. Efficiently managing the resources in such computing platforms to achieve high performance becomes challenging for cloud users due to the dynamic arrival of data from different sources. This chapter aims at providing users a novel approach for resource management for big data processing in multi-cloud systems.

Generally, we can refer to the amount of work to process data on a computing node as a *load*. For any realistic cloud computing platform, the number of loads submitted to the system is usually much larger than the number of available computing nodes. The problem of load assignment to computing nodes is therefore the most important and challenging since it directly affects the system performance. However, designing an efficient assignment strategy, which is known as *scheduling strategy*, faces many challenges that will be listed below. The first one is rather a required performance metric that a scheduling strategy needs to achieve while the others are characteristics of the infrastructure components that a scheduling strategy needs to take into account during the scheduling process.

1. **Load balancing among computing nodes.** An efficient scheduling strategy should guarantee the load balancing among computing nodes. This will avoid the scenario that a node may have to process many loads in a long period while other nodes stay idle. Such scenario prevents users from achieving high resource utilization and it degrades the overall performance. Users may need to pay a higher cost due to longer resource usage time when the running time of their applications is prolonged. The problem will be more difficult when we consider a more complex system architecture that includes multiple load sources, e.g., a monitoring system with many sensors that capture the data and send to different storage servers for processing and storing. To minimize the resource usage cost, in such a system, users may reserve a single pool of computing resources and share among load sources for processing loads, making the problem of load balancing harder.

2. **Availability and heterogeneity of nodes.** Computing nodes are frequently unavailable due to unexpected failures or they are shutdown for maintenance. Knowing the moment when nodes are available, known as *release time*, is important to the scheduling strategy since loads can be assigned only to released nodes. With priori known release time of computing nodes, the scheduling strategy can statically assign loads to computing nodes well ahead before the actual load processing. However, with unknown release times, the scheduling strategy needs to react dynamically to

new released nodes. Furthermore, if it is assumed that nodes are able to process only one load at a time, the scheduling strategy then needs to be aware of the *ready time* of nodes, i.e., the moment when nodes finish the processing of already received loads. Among released nodes, the node with the earliest ready time is then selected for a new arriving load to achieve load balance.

The ready time of computing nodes depends on the processing time of loads, which in turn depends on the capacity of computing nodes and the nature of loads. Since nodes are usually heterogeneous in terms of computing capacity, the processing time of a load may vary on different nodes. Furthermore, due to the nature of loads submitted to the system, loads may require different amount of computations. For instance, one load deals with compression and the other deals with encryption using different key lengths, resulting in different amount of running time. Throughout this chapter, we refer to this characteristic simply as *computation requirement* of loads.

3. **Network topology and link capacities.** Most of existing scheduling strategies do not consider a specific network topology [Braun et al., 1999, Maheswaran et al., 1999]. They assume that the data transmission is performed over the Internet or the network is fully connected. The total processing time of applications including the data transmission time and execution time will be interfered by other users who are using the Internet or sharing the same network. While the interference is tolerable for some applications, i.e., the running time of applications will last longer when bandwidth is low due to bandwidth sharing, many applications are requiring a stable bandwidth on the links they are using, e.g., Content Delivery Networks, to guarantee the quality of service (QoS). Such applications have been deployed on a dedicated platform with specific network topology and link capacities such that the QoS is guaranteed. With the development of virtualization technologies in the cloud environment, such applications are now migrated to public clouds. For instance, Netflix[1], which is a major online video streaming service provider in North America, moved its data storage system, streaming servers, encoding engine, and other major modules to Amazon Web Services (AWS) in 2010 [Ciancutti, 2010, Adhikari et al., 2012]. It is thus necessary to consider these parameters in the scheduling problem to reflect realistic scenarios.

Although many previous studies have considered the scheduling problem in a distributed computing infrastructure [Braun et al., 1999, Maheswaran et al., 1999, Yang et al., 2013], they do not address all the challenges mentioned above. In this chapter, we present our study that bridges the gap between existing scheduling strategies and the new challenges in multi-cloud systems. We propose two scheduling strategies: a Static Scheduling Strategy (SSS) and a Dynamic Scheduling Strategy (DSS). SSS assumes that the release time of computing nodes is known prior to the scheduling. DSS relaxes the assumption of SSS by applying a prediction technique to estimate the release time of computing nodes. Both strategies take into account the topology and capacity of the system network, and the heterogeneity of computing nodes. To achieve a better load balance, beyond the existing approaches, which consider the computing capacity of nodes, the size of loads and the computation requirement of loads, we further apply the *Divisible Load Theory* (DLT), which assumes that loads can be perfectly divided into a number of chunks with different sizes [Cheng and Robertazzi, 1988]. As shown in [Zeng et al., 2016], divisible loads have been broadly used in streaming data applications such as monitoring systems, continuous

[1]Netflix: `www.netflix.com`

write applications. Such applications are nowadays migrated to public clouds to handle large amount of data. Thus, integrating DLT in the scheduling strategies for multi-cloud systems running above applications is naturally appropriate to achieve high performance and load balancing among computing nodes. We implement DLT model by adopting the phase-based scheduling approach [Hu and Veeravalli, 2011] that divides the processing of a load into multiple phases. In each phase, a load chunk will be processed on a node. Multiple load chunks of a load can be processed in parallel on different available nodes.

As mentioned earlier, our study considers multi-cloud systems in which a pool of computing nodes are shared among load sources so as to minimize the resource usage cost that users need to pay for cloud providers. In such environment, since computing nodes may be processing other loads submitted from different gateways, the ready time of computing nodes is therefore unknown to a scheduler that do not have the view of the entire system. To compute the ready time of a computing node when it is processing other loads, we apply existing prediction techniques, which allow the computing node to estimate the processing time of a certain load chunk based on the historical processing information, i.e., we assume that there exists on each computing node a training dataset that will be used for prediction model. In a realistic scenario, the size of this training dataset may be small at the beginning, but it will be enriched over time along with the arrival and processing of loads.

In summary, the main contributions of this chapter are:

- We propose novel architectures of multi-cloud systems that can satisfy the complex requirements of users on computing resources, network topology and guaranteed quality of services.
- Based on these architectures, we propose two scheduling strategies employing DLT paradigm that addresses all challenges and requirements of an efficient scheduling strategy. This is an important contribution to the design of distributed schedulers for multi-cloud systems in handling large volume of data in nowadays large-scale applications.
- We apply prediction techniques on computing nodes to estimate their ready time that is a required information for the scheduling strategy used in the distributed scheduling architecture.
- We evaluate the performance of the proposed scheduling strategies through comprehensive simulations and compare them against baseline strategies to demonstrate the effectiveness of the proposed approach in realistic systems.

3.2 System Architectures

In this section, we present our design for multi-cloud system architectures using two scheduling approaches: a centralized scheduling approach and a distributed scheduling approach.

3.2.1 Centralized Scheduling System Architecture

The architecture of a multi-cloud system using the centralized scheduling approach can be designed in the way shown in Fig. 3.1. Each load source is designed as a gateway in the system architecture. Loads, which are arriving to gateways $1, \ldots, M$, are divided into multiple sub-loads (chunks) and are assigned to computing nodes CN 1,..., CN N. Each gateway may or may not connect to all computing

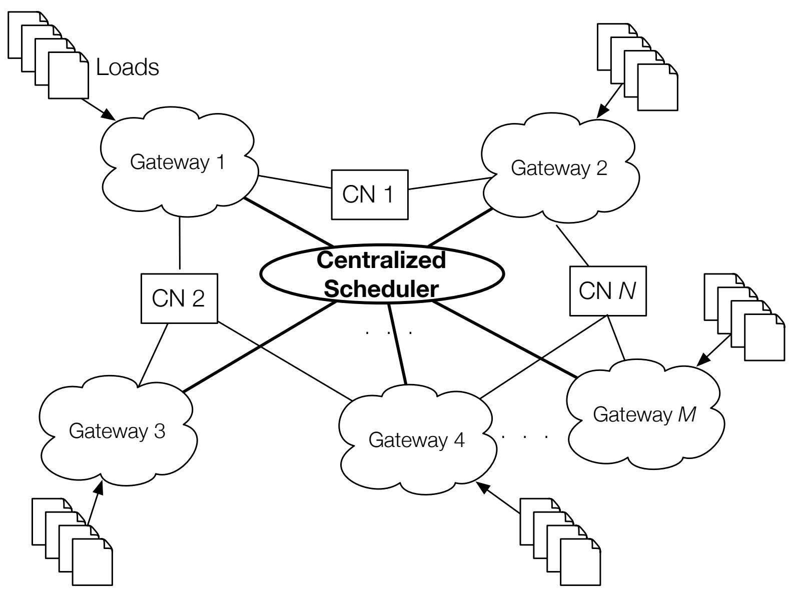


Figure 3.1: Architecture of multi-cloud system with centralized scheduling approach.

nodes by dedicated links with different bandwidth, e.g., Gateway 1 connects to CN 1, CN 2 and CN 3 but not CN N. The centralized scheduler is responsible for scheduling loads on computing nodes. It supposes to have all information about loads, capacities of computing nodes and bandwidth of the links. All these information are used as parameters of the scheduling problem, i.e., *meta-data*. Based on this information, the centralized scheduler will run a scheduling strategy that integrates DLT to (i) select a load, (ii) select a computing node that will process a chunk of the selected load, and (iii) decide the size of load chunk. After receiving the scheduling decision, the gateway performs the load division and transmits the load chunk to the computing node for processing.

We note that the deployment of the proposed architecture is feasible with any public cloud such as Amazon EC2, Google Compute Clusters or Microsoft Azure. Indeed, these cloud providers offer IaaS such that users will rent a pool of virtual machines to serve as computing resources. Leverage on network virtualization, network bandwidth is also reserved to establish dedicated links with guaranteed bandwidth among computing nodes and load sources, i.e., the gateways. With the elasticity of the cloud computing paradigm, users can elastically *adjust* the size of their computing infrastructure based on their monetary budget for resources. Ideally, one may rent many pools of virtual machines, each dedicated to process loads from a particular gateway. Similarly, bandwidth can also be reserved for a fully connected network between computing nodes and gateways. However, the resource usage cost may be significant and resources may be under-utilized due to the dynamic arrival of loads. Thus, the proposed architecture wherein resources are shared among load sources improves the resource utilization and thereby reducing the usage cost that users need to pay for cloud providers.

It is worth mentioning that the centralized scheduling architecture proposed above creates a single point of failure at the scheduler. Since it is responsible for scheduling loads coming from all load sources, it may be overloaded when loads are submitted to the gateways heavily. Furthermore, in case of software crashes or shutdown for maintenance, the entire system will be unavailable for a duration since there is not an alternative scheduler. This motivates us propose a more complex system architecture using distributed scheduling approach. We present this advanced architecture in the next section.

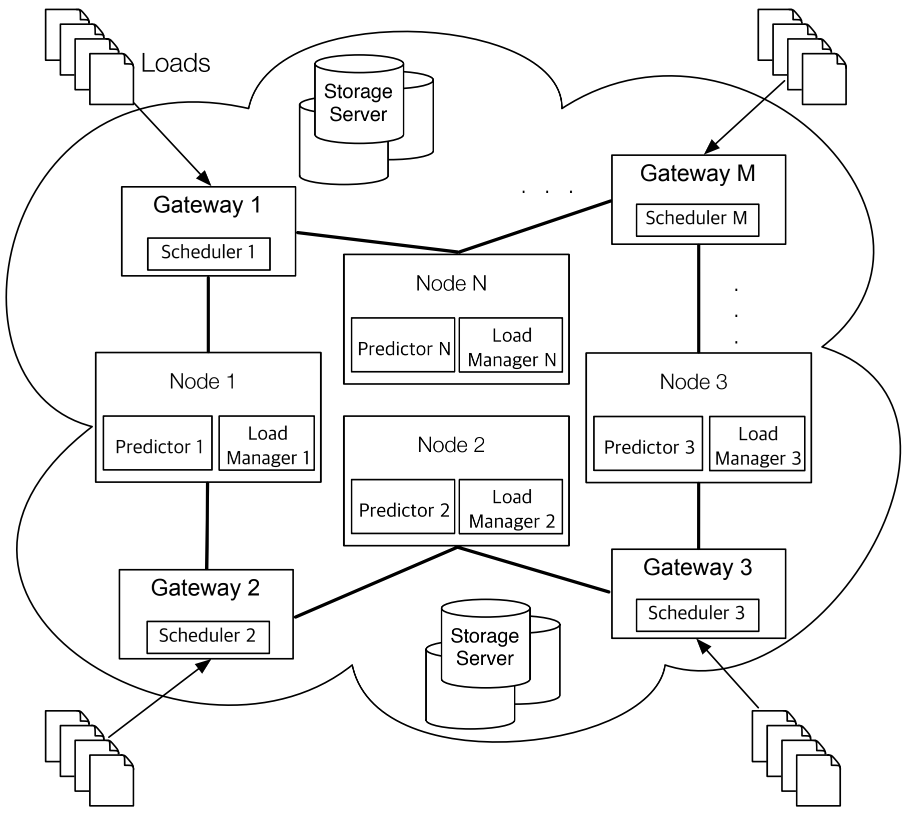


Figure 3.2: Architecture of multi-cloud system with distributed scheduling approach.

3.2.2 Distributed Scheduling System Architecture

The architecture of a multi-cloud system supporting the distributed scheduling approach can be designed in the way as shown in Fig. 3.2. The architecture also comprises multiple gateways that are distributed geographically across public clouds. At each gateway, a dedicated node, that also has a scheduler, is responsible for receiving loads. The scheduler in that node is responsible for scheduling the processing of loads on computing nodes. All the gateways share a common pool of computing nodes that have heterogeneous computing capacities and they are connected to the gateways with heterogeneous link capacities. Each gateway may or may not connect to all nodes by dedicated links with different capacities, e.g., Gateway 1 connects to Node 1 and Node N but not Node 2 and Node 3 as shown in Fig. 3.2. Similar to the case of the centralized scheduling approach, given our assumption that loads are perfectly divisible, and that the distributed scheduler uses the phase-based scheduling approach [Kang et al., 2014b], at each processing phase, the scheduler selects a load, decides the size of a load chunk and a computing node that will process this load chunk. The size of the load chunk is determined based on the computing capacity of the node, the computation requirement of the load and the capacity of the link connecting the computing node and the corresponding gateway.

Unlike the case of the centralized scheduling architecture where all computing nodes are managed by the sole scheduler. Computing nodes in the distributed scheduling architecture receives load chunks from different gateways (schedulers). A scheduler may not know how many load chunk a computing node have received for processing to decide whether or not assign a new load chunk to this node. To serve this purpose, on each computing node, a load manager is also deployed to be responsible for receiving, invoking and logging the processing of load chunks assigned to the corresponding computing node. Following information concerning a load chunk and its processing are logged by the load manager:

- Load type: The application of the load such as encryption, compression, etc. Each type is associated with a value of computation requirement of this load type that is estimated using a prediction technique based on historical processing information;
- Load chunk size: The size of the load chunk determined by the scheduler;
- Start transmission instant of load chunk: The time instant when the scheduler starts sending a load chunk to a computing node. The transmission may not start immediately after assignment since the computing node may be busy for other transmission, assuming that there is only one transmission at a time on each node. Otherwise, load transmissions will interfere with each other;
- Transmission time of load chunk: The duration taken to transmit a load chunk from a gateway to a computing node;
- Start processing instant of load chunk: The time instant when the actual processing of a load chunk starts on a computing node;
- Processing time of load chunk: The duration taken by a computing node to process a load chunk.

When a load chunk is assigned to a computing node, the timing information of its processing, i.e., the last four parameters described above, are not available yet since the computing node might be busy for another load chunk, assuming that computing nodes are able to receive and process one load chunk at a time as mentioned earlier. Thus, the actual transmission and processing of current load chunk will be performed in the future. However, this information is needed for the computing node to estimate its ready time for receiving and processing a new load chunk, i.e., the time instants when the node finishes the transmission and processing of all already received chunks. To address this issue, we equip each node a *predictor* that implements a prediction technique to estimate the required timing parameters.

While the transmission time of a load chunk is estimated based on the load chunk size and link capacities, the processing time of a load chunk depends on not only the load chunk size and computing capacity of the node but also the computation requirement of the load. However, the computation requirement of a load on a particular computing node is unknown a priori. The computing node therefore needs to estimate this parameter using a prediction technique. Furthermore, during the scheduling process, the scheduler also needs to know the computation requirement of a load to determine an appropriate load chunk size for a particular computing node. Thus, when a load is submitted to a gateway, if the respective scheduler does not know the computation requirement of that load, the scheduler will send a request to all computing nodes connected to the gateway. All the requested nodes execute a prediction algorithm and send the results to the scheduler. This process is depicted in Fig. 3.3 from steps 1.1 to 1.4.

When there exists a load to be scheduled for processing at any gateway, the respective scheduler requests all nodes for their ready time for receiving and processing a new load chunk. All requested nodes run a prediction algorithm to estimate their ready time and send the results to the respective scheduler. Based on ready times of all requested nodes, the best computing node is then selected.

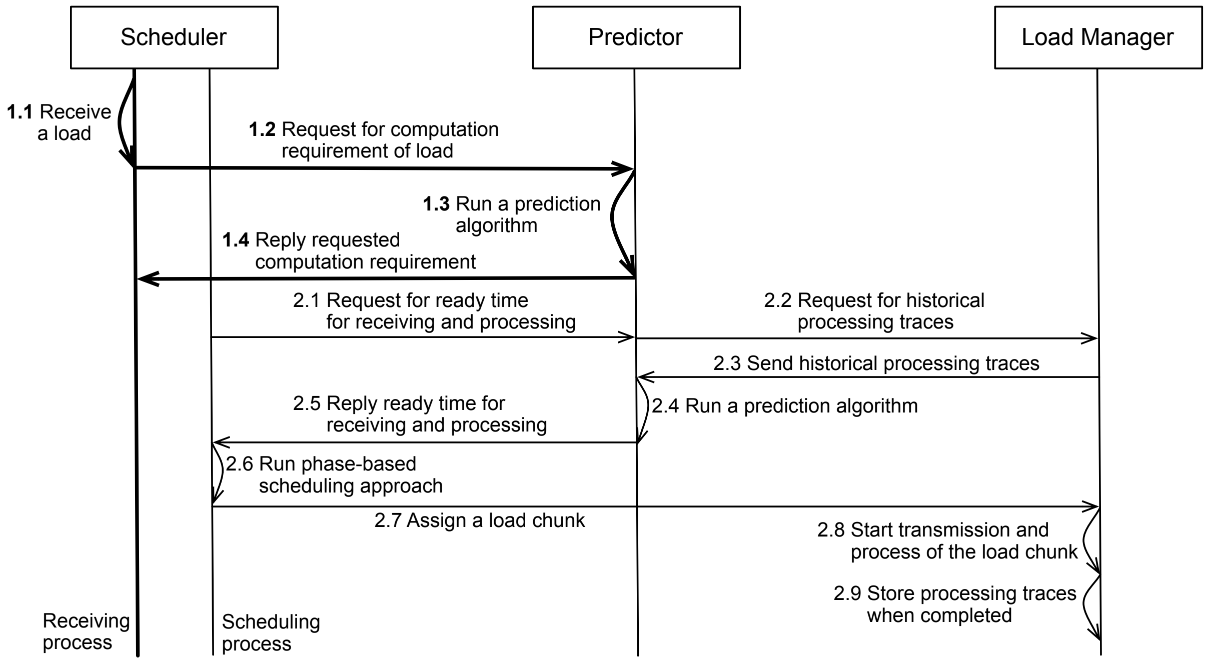

Scheduler
Predictor
Load Manager
1.1 Receive a load
1.2 Request for computation requirement of load
1.3 Run a prediction algorithm
1.4 Reply requested computation requirement
2.1 Request for ready time for receiving and processing
2.2 Request for historical processing traces
2.3 Send historical processing traces
2.4 Run a prediction algorithm
2.5 Reply ready time for receiving and processing
2.6 Run phase-based scheduling approach
2.7 Assign a load chunk
2.8 Start transmission and process of the load chunk
2.9 Store processing traces when completed
Receiving process
Scheduling process

Figure 3.3: Interaction diagram among scheduler, load manager and predictor.

Based on the ready time of the selected node and computation requirement of the load, the scheduler determines an appropriate load chunk size to guarantee the load balancing among computing nodes. Taking the scheduling decision, the scheduler performs the load division and transmits the load chunk to the computing node when the selected node is ready for receiving the load chunk. This scheduling process is also depicted in Fig. 3.3 from steps 2.1 to 2.9.

3.3 Phase-based Scheduling Approach

In this section, we first describe all the mathematical notations used throughout of this chapter. We then present the detailed description of the phase-based scheduling approach that applies the divisible load theory. It is to be noted that similar to the case of the centralized scheduling approach, when applying the distributed scheduling approach, all schedulers deployed on respective gateways will run the same scheduling strategy that will be described hereafter. Thus, regardless of the number of gateways, i.e., the number of schedulers, we omit the index of the scheduler indicated in the following description.

3.3.1 Mathematical Notations and Assumptions

We assume that at the instant when the scheduler runs the scheduling algorithm, there is a total of N computing nodes. With the static scheduling strategy, we assume that release times of computing nodes are known to the scheduler. Among these N nodes, some of them have been released and others will be released in the future. With the dynamic scheduling strategy, we assume that all the N computing nodes have been released and connected to the gateway. The set of released nodes will be dynamically updated during the scheduling process when new nodes are released. It is to be noted that several nodes may also be connected to and receive loads from other gateways. For computing node $i, i = 1, \ldots, N$, let T_i, W_i

Table 3.1: Mathematical notations

Notation	Description
N	Number of computing nodes
T_i	Release time of node i, $i = 1, \dots, N$
W_i	Computation capacity of node i
B_i	Bandwidth of the link between node i and the gateway
S_j	Size of load j submitted to the gateway
S_j^{unit}	Size of a unit load of load j
U_j	Total number of units of load j, $U_j = \lceil S_j / S_j^{\text{unit}} \rceil$
$C_{i,j}^{\text{unit}}$	Computation requirement of load j on computing node i
T^{phase}	Duration of a processing phase
N_j	Number of unallocated units of load j
$R_{i,j}$	Remaining amount of computation of load j on computing node i
T_i^{idle}	Idle time of computing node i
R_i^{pro}	Ready time of node i for processing a new load chunk
T_i^{comp}	Available processing time of node i
R_i^{rec}	Ready time of node i for receiving a new load chunk
$C_{i,j}$	Number of units of load j assigned to computing node i
S_l^{chunk}	Size of load chunk l
T_l^{assign}	Assignment instant of load chunk l
T_l^{strans}	Start transmission instant of load chunk l
T_l^{trans}	Transmission time of load chunk l
T_l^{sproc}	Start processing instant of load chunk l
T_l^{proc}	Processing time of load chunk l

and B_i denote its release time, computing capacity and the bandwidth of the link between computing node i and the gateway, respectively. We also assume that all computing nodes are able to process any load without any technical limitation such as software licenses or libraries.

Assuming that loads are submitted to the gateway with arbitrary arrival rate, load $j, j = 1, 2, \dots,$ has a size of S_j, e.g., in MB. We also assume that load j can be divided with the smallest load unit of size S_j^{unit}, the total number of load units of load j is then defined as $U_j = \lceil S_j / S_j^{\text{unit}} \rceil$. As mentioned previously, loads may be of different types, e.g., one load may deal with encryption and the other deals with compression or encryption with different key lengths. We denote $C_{i,j}^{\text{unit}}$ as the amount of computation required to process a unit of load j on computing node i, which is estimated by the predictor deployed on computing node i. All the mathematical notations are summarized in Table 3.1.

3.3.2 Phase-based Scheduling Approach

The phase-based scheduling approach has been widely used in the literature [Hu and Veeravalli, 2011, Kang et al., 2014b, Lin et al., 2007, Veeravalli and Wong, 2004]. The basic idea of the phase-based scheduling approach is dividing the processing of a load into multiple processing phases. In each phase,

Algorithm 1 Phase-based scheduling approach

```
Input: $N_j$, $C^{\text{unit}}_{i,j}$, $S^{\text{unit}}_j$, $T_i$, $W_i$, $B_i$, $R^{\text{rec}}_i$, $R^{\text{pro}}_i$ and $k$,   $i = 1, \ldots, N, j = 1, 2, \ldots$
Output: $i^*$, $j^*$ and $C_{i^*,j^*}$.
 1: $R_{i,j} \leftarrow N_j C^{\text{unit}}_{i,j}, i = 1, \ldots, N, j = 1, 2, \ldots$;
 2: $R_{i,j^*} \leftarrow \max\{R_{i,j}\}$; /*select a load*/
 3: for $i = 1 \to N$ do
 4:     $T^{\text{idle}}_i \leftarrow (k+1)T^{\text{phase}} - R^{\text{pro}}_i$;
 5: end for
 6: for $i = 1 \to N$ do
 7:     if $\left(T_i < (k+1)T^{\text{phase}}\right) \wedge \left(T^{\text{idle}}_i > 0\right)$ then
 8:         Compute the available processing time of node $i$, $T^{\text{comp}}_i$, as defined in Eq. (3.4);
 9:     end if
10: end for
11: Select the node with the longest available processing time: $i^* \leftarrow \max\{T^{\text{comp}}_i\}, i = 1 \ldots N$;
12: Compute the size of the chunk of load $j^*$ to be assigned to node $i^*$, $C_{i^*,j^*}$, as defined in Eq. (3.9);
```

a load chunk will be processed on a computing node. Depending on the number of available computing nodes, multiple load chunks can be assigned to different nodes to be processed in parallel. We assume that the duration of a processing phase, denoted as T^{phase}, is fixed and pre-defined by the users. In practice, it depends on the nature of each application, users may need to do a benchmark to determine an optimal value for the duration of a processing phase [Kang et al., 2014b]. By applying the phase-based scheduling approach, the scheduler guarantees that load chunks scheduled during phase k, $k = 1, 2, \ldots$, will be completed before processing phase $k + 1$ ends.

The phase-based scheduling approach aims at improving the load balancing among computing nodes and exploiting as much as possible the parallelism for load processing, thereby reducing the total processing time of all loads. The main challenging questions are how to determine an appropriate size for a load chunk that will be assigned to a particular computing node for the corresponding phase; and how to select a load when there exist many loads to be processed. In this section, we present three steps of the phase-based scheduling approach: (i) load selection, (ii) node selection, and (iii) determination of load chunk size. While the load selection method is based on the Most Remaining Load First (MRLF) policy to ensure the balancing among arriving loads at a certain gateway, the node selection method and determination of chunk size reflect the heterogeneity of computing nodes in terms of computing and network capacities. The pseudo code of these steps is briefly presented in Algorithm 1. During each phase, Algorithm 1 can be run multiple times by the scheduling strategy until all available computing nodes are utilized or loads are completely processed.

3.3.2.1 Load selection

Assuming that there are many loads that are waiting for being scheduled at the gateway and given that their computation requirements have been provided by all computing nodes. To achieve balance among loads, we adopt the Most Remaining Load First (MRLF) policy proposed in [Hu and Veeravalli, 2011].

The MRLF policy gives the priority of being processed to the load with the most remaining amounts of required computations. Let $R_{i,j}$ be the total remaining amount of computation required for processing load j on computing node i. Then, $R_{i,j}$ can be defined as follows:

$$R_{i,j} = N_j C^{\text{unit}}_{i,j}, \tag{3.1}$$

where N_j is the number of unallocated units of load j, initialized by U_j when load j is just submitted. Determining the most remaining load is done by simply evaluating $\max\{R_{i,j}\}, i = 1, \ldots, N$ and $j = 1, 2, \ldots$, as shown in line 2 of Algorithm 1. At the end of this step, let load j^* be the selected load to be processed in the corresponding processing phase.

3.3.2.2 Node selection

Given that load j^* is selected, we determine a computing node that will process a chunk of load j^*. Assuming that the index of the current processing phase is k, among the computing nodes that are connected to the gateway of load j^*, node i can be selected if it satisfies the two following conditions:

1. Node i is released before processing phase $k+1$ ends since all computing nodes are scheduled to finish the load chunks when processing phase $k+1$ ends, i.e., $T_i < (k+1)T^{\text{phase}}$; and

2. Node i has idle time before processing phase $k+1$ ends, i.e., the period that node i does not perform any action including load transmission or processing. We denote T^{idle}_i as the idle time of node i before processing phase $k+1$ ends. As shown in line 4 of Algorithm 1, the idle time of node i is defined as $T^{\text{idle}}_i = (k+1)T^{\text{phase}} - R^{\text{pro}}_i$ where R^{pro}_i is the ready time of node i for processing a new load chunk. As mentioned earlier, the scheduler does not know the ready time of node i for processing a new load chunk since it may be assigned load chunks from other schedulers. The scheduler therefore needs to request node i for this parameter before running Algorithm 1.

Among the feasible computing nodes, we select the node that has the longest *available processing time*, i.e., the duration used only for load processing, excluding the time for load transmission. We denote the available time for processing of node i as T^{comp}_i. To obtain T^{comp}_i, four constraints should be considered:

1. Node i can process loads only after it is released, i.e., $T^{\text{comp}}_i \leqslant (k+1)T^{\text{phase}} - T_i$;

2. Node i can process a chunk of load j^* only after it finishes all previously received load chunks. This implicitly means that $T^{\text{comp}}_i \leqslant T^{\text{idle}}_i$;

3. Even if node i has been released and it has idle time, necessary load transmission time should be subtracted from the idle time. This constraint is represented as follows:

$$T^{\text{comp}}_i \leqslant \frac{B_i C^{\text{unit}}_{i,j^*} \left((k+1)T^{\text{phase}} - R^{\text{rec}}_i\right)}{S^{\text{unit}}_{j^*} W_i + B_i C^{\text{unit}}_{i,j^*}}, \tag{3.2}$$

where R^{rec}_i is the ready time of node i for receiving a new load chunk. This parameter is unknown to the scheduler that therefore needs to request computing node i before running Algorithm 1; and

4. The transmission of all load chunks during processing phase k assigned to node i should finish before processing phase $k+1$ starts. This constraint is represented as follows:

$$T_i^{\text{comp}} \leqslant \frac{B_i C_{i,j^*}^{\text{unit}} \left(kT^{\text{phase}} - R_i^{\text{rec}}\right)}{S_{j^*}^{\text{unit}} W_i}. \tag{3.3}$$

Consequently, the available processing time of node i until phase $k+1$ ends is defined as follows:

$$T_i^{\text{comp}} = \min\{V_1, V_2, V_3, V_4\}, \tag{3.4}$$

where

$$V_1 = (k+1)T^{\text{phase}} - T_i, \tag{3.5}$$

$$V_2 = T_i^{\text{idle}} = (k+1)T^{\text{phase}} - R_i^{\text{pro}}, \tag{3.6}$$

$$V_3 = \frac{B_i C_{i,j^*}^{\text{unit}} \left((k+1)T^{\text{phase}} - R_i^{\text{rec}}\right)}{S_{j^*}^{\text{unit}} W_i + B_i C_{i,j^*}^{\text{unit}}}, \tag{3.7}$$

$$V_4 = \frac{B_i C_{i,j^*}^{\text{unit}} \left(kT^{\text{phase}} - R_i^{\text{rec}}\right)}{S_{j^*}^{\text{unit}} W_i}. \tag{3.8}$$

After having all T_i^{comp}'s values, $i = 1, \ldots, N$, determining the best computing node is done by simply evaluating $\max\{T_i^{\text{comp}}\}, i = 1, \ldots, N$. We denote i^* as the index of the selected node that has the longest available processing time to process a chunk of load j^*.

It is worth mentioning that the formulas above take into account the heterogeneity of computing nodes in a cloud computing infrastructure. A computing node is selected because not only it necessarily has a dedicated link to the gateway for data transmission but also it has the highest link capacity, computing capacity, leading to the longest available time for load processing. As mentioned earlier, in an elastic computing environment like clouds, users may rent as many as possible virtual machines to serve as computing nodes. They can also reserve as much as possible network bandwidth to create dedicated links among computing nodes and load sources. However, reserving full computing and network capacities costs a lot and resources may be under-utilized when computing demand is low during off-peak hours. Thus, the proposed architecture and methods for load scheduling presented above provide users means for improving resource utilization and reducing the usage cost of cloud resources.

3.3.2.3 Determination of chunk size

Given the results of the previous steps that a chunk of load j^* will be assigned to node i^*, we now describe the formula to compute the size of the chunk. Based on the available processing time of node i^*, $T_{i^*}^{\text{comp}}$, we can obtain the chunk size, denoted as C_{i^*,j^*}, which is the number of units of load j^*:

$$C_{i^*,j^*} = \lfloor \frac{W_{i^*} T_{i^*}^{\text{comp}}}{C_{i^*,j^*}^{\text{unit}}} \rfloor. \tag{3.9}$$

If the number of unallocated units of load j^*, N_{j^*}, is smaller than C_{i^*,j^*}, then C_{i^*,j^*} is accordingly reduced to N_{j^*}. This means that all unallocated units of load j^* are assigned to node i^*.

3.4 Load Management on Computing Nodes and Prediction Techniques for Distributed Scheduling Architecture

As discussed in the section of system architectures, with the distributed scheduling architecture, each computing node is equipped a load manager for receiving, invoking and logging the processing of load chunks. This section gives the detailed description of the components running on computing nodes and mathematical models used in prediction techniques. Since these models will be executed on all computing nodes, we thus generally present the description for a particular computing node, say node i.

3.4.1 Load Management on Computing Nodes

We assume that there exists a number of load chunks that are waiting for being processed on computing node i and load chunk l is the latest assigned one. As mentioned in Section 3.2.2, to estimate the ready time for receiving and processing of node i, the load manager deployed on node i stores several information of load chunk l such as load type, load chunk size, start transmission instant, transmission time, start processing instant and processing time. While load type and load chunk size are known when load chunk l is assigned to node i, the timing information are unknown until load chunk l is completely processed. The estimated values of the timing parameters are however needed to compute the ready time of nodes for receiving and processing a new load chunk, whereas the actual values obtained after load chunk is completely processed can be used to enrich the training dataset of the prediction techniques.

Since the capacity of the link connecting node i and the gateway is B_i, the transmission time of a load chunk assigned to node i is easily computed by dividing the size of the load chunk and the link capacity. Thus, the transmission time of load chunk l, denoted as T_l^{trans}, is as follows:

$$T_l^{\text{trans}} = \frac{S_l^{\text{chunk}}}{B_i}, \tag{3.10}$$

where S_l^{chunk} is the size of load chunk l. Given the estimated transmission time of all previously assigned chunks to node i, we can infer the start transmission instant of load chunk l, denoted as T_l^{strans}, as follows:

$$T_l^{\text{strans}} = \max\{T_l^{\text{assign}}, T_{l-1}^{\text{strans}} + T_{l-1}^{\text{trans}}\}, \tag{3.11}$$

where T_l^{assign} is the assignment instant of load chunk l. This means that at the assignment instant of load chunk l, if computing node i does not perform any transmission, then it can immediately start receiving load chunk l. Otherwise, load chunk l should wait until the previously assigned load chunk, load chunk $l-1$, is completely transmitted to start the its transmission.

The processing time and the start processing time instant of load chunk l can be estimated similarly. However, since the processing time of load chunk l depends on the computation requirement of the load that is unknown to computing node i, we apply prediction techniques to estimate such parameter. The prediction models will be presented in the next section. Given the estimated processing time of all previously assigned load chunks on computing node i, the start processing time instant of load chunk l, denoted as T_l^{sproc}, is defined as follows:

$$T_l^{\text{sproc}} = \max\{T_l^{\text{strans}} + T_l^{\text{trans}}, T_{l-1}^{\text{sproc}} + T_{l-1}^{\text{proc}}\}, \tag{3.12}$$

where T_l^{strans} and T_l^{trans} are the start transmission instant and the transmission time of load chunk l, respectively. T_{l-1}^{sproc} and T_{l-1}^{proc} are the start processing instant and the processing time of the previously assigned load chunk, load chunk $l-1$. This also means that load chunk l can be processed immediately after finishing its transmission if node i is not busy. Otherwise, load chunk l should wait until the previously assigned load chunk, load chunk $l-1$, is completely processed.

3.4.2 Prediction Techniques

In this section, we will present the prediction techniques used by the predictor deployed on computing node i to estimate the computation requirement of a load and processing time of load chunks assigned to computing node i. It is to be noted that designing a novel prediction technique is out of scope of our study since there have been lots of fundamental prediction algorithms proposed in the literature [Zhang et al., 2011, Jiang et al., 2011, Verma et al., 2013, Islam et al., 2012]. Among them, we adopt the linear regression technique [Islam et al., 2012] because of its simplicity.

3.4.2.1 Computational requirement of load

Assuming that a scheduler requests node i for the computational requirement of load j, the predictor at node i needs to estimate the computation requirement of load j based on historical processing information that is used as the training dataset of the prediction model. We also assume that in the training data set stored on node i, there are M load chunks, which have the same load type with load j, have been received from other gateways and completely processed by node i. The computation requirement of load j on node i, denoted as $C_{i,j}^{\text{unit}}$, is estimated as follows:

$$C_{i,j}^{\text{unit}} = \frac{\sum_{l=1}^{M} \frac{W_i T_l^{\text{proc}} S_j^{\text{unit}}}{S_l^{\text{chunk}}}}{M}, \tag{3.13}$$

where W_i is the computing capacity of node i, T_l^{proc} is the processing time of load chunk l, S_j^{unit} is the size of a unit of load j, S_l^{chunk} is the size of load chunk l.

3.4.2.2 Processing time of load chunk

Given that load chunk l with size of S_l^{chunk} is assigned to node i, the processing time of load chunk l on node i is estimated by the linear regression technique as follows:

$$T_l^{\text{proc}} = \gamma_0 + \gamma_1 S_l^{\text{chunk}}, \tag{3.14}$$

where γ_0 and γ_1 are the linear regression parameters, which are determined by executing the gradient descent on the training dataset obtained from the historical processing traces. Since the gradient descent is simple, we omit the description of the method in this chapter.

3.4.2.3 Ready time of node for receiving a new load chunk

If computing node i has never been assigned a load chunk after it is released, it is then ready for receiving load immediately. However, if node i has been assigned several load chunks, the ready time of node i

for receiving a new load chunk is the moment when it finishes the transmission of the latest assigned load chunk. Assuming that load chunk l is the latest assigned load chunk, the ready time of computing node i for receiving a new load chunk is defined as follows:

$$R_i^{\text{rec}} = \begin{cases} T_i & \text{if node } i \text{ has never been assigned a load chunk;} \\ T_l^{\text{strans}} + T_l^{\text{trans}} & \text{otherwise.} \end{cases} \tag{3.15}$$

3.4.2.4 Ready time of node for processing a new load chunk

Similar to the ready time for receiving, if node i has never been assigned a load chunk after it is released, it is then ready for processing a new load chunk. However, if node i has been assigned several load chunks, the ready time of node i for processing a new load chunk is the moment when it finishes the processing of the latest assigned load chunk. Let load chunk l be the latest assigned load chunk, the ready time of node i for processing a new load chunk is then defined as follows:

$$R_i^{\text{proc}} = \begin{cases} T_i & \text{if node } i \text{ has never been assigned a load chunk;} \\ T_l^{\text{sproc}} + T_l^{\text{proc}} & \text{otherwise.} \end{cases} \tag{3.16}$$

3.5 Static Scheduling Strategy

By applying the phase-based scheduling approach, we now present the Static Scheduling Strategy (SSS) with the assumption that the release times of computing nodes are known before the scheduling starts. This assumption still reflects the realistic scenario where computing nodes are shut down for maintenance or any scheduled plan. Then, they will be turned on and ready for processing at the scheduled time. In addition, we restrain the application of SSS to the centralized scheduling architecture where the scheduler also knows the ready time of computing nodes for processing a new load chunk since all the load chunks assigned to a computing nodes are managed by the same scheduler.

As shown in Algorithm 2, the input of SSS is the release time of computing nodes, network topology and links' bandwidths, computation capacity of computing nodes, and size of loads. It is to be noted that since a computing node may be connected to different gateways with different link bandwidth. We thus denote the bandwidth of the link connecting node i and gateway g as $B_{i,g}$. The output of SSS is the entire scheduling solution of all loads submitted to all the gateways, including scheduling solutions for all phases from the moment when the system starts until when all loads are completely processed. It is also worth mentioning that since the scheduler knows the release time of computing nodes, Algorithm 2 is run only once and the scheduling decisions are stored as a lookup table for the use during each processing phase. The load chunk transmission can start immediately even a load chunk may not be processed right after it finishes the transmission.

The algorithm starts by initializing required parameters such as the number of units of each load, the time instant when CN i, $i = 1, \ldots, N$ can start receiving the first load chunk, $R_i^{\text{rec}}, \forall i$, and the time instant when node i can start processing the fist load chunk, R_i^{proc}. Both R_i^{rec} and R_i^{proc} are initialized by the release time of computing nodes (see lines 2–4 in Algorithm 2). The main part of SSS is the *while* loop. In each iteration, the algorithm produces the scheduling decisions for a processing phase. This

Algorithm 2 Static Scheduling Strategy

```
Input: $T_i$, $B_{i,g}$, $W_i$ and $S_{j,g}$, $\forall j, \forall g$ and $\forall i$.
Output: $\{j^*\}$, $\{i^*\}$, $\{k\}$, and $\{C_{j^*,i^*,k}\}$.
 1: $k \leftarrow 0$; /*index of processing phase*/
 2: $U_j \leftarrow \lceil S_j / S_j^{\text{unit}} \rceil, \forall j$;
 3: $N_j \leftarrow U_j, \forall j$; $R_i^{\text{rec}} \leftarrow T_i, \forall i$;
 4: $R_i^{\text{proc}} \leftarrow T_i, \forall i$;
 5: while $\exists N_j \neq 0, \forall j$ do
 6:     $k \leftarrow k + 1$;
 7:     $stop \leftarrow 0$; /*stopping flag for a processing phase*/
 8:     while $stop \neq 0$ do
 9:         $\{j^*, i^*, C_{j^*,i^*,k}\} \leftarrow$ Run Algorithm 1 with $N_j$, $T_i$, $R_i^{\text{rec}}$, $R_i^{\text{proc}}$, $B_{i,g}$, $W_i$, $\forall j, \forall g, \forall i$;
10:         if $\{j^*, i^*, C_{j^*,i^*,k}\} \neq \emptyset$ then
11:             $N_{j^*} \leftarrow N_{j^*} - C_{j^*,i^*,k}$;
12:             $R_{i^*}^{\text{rec}} \leftarrow R_{i^*}^{\text{rec}} + C_{j^*,i^*,k} S_{j^*}^{\text{unit}} / B_{i^*,g}$;
13:             Update $R_{i^*}^{\text{proc}}$ as defined in Eq. (3.17);
14:         else
15:             $stop \leftarrow 1$;
16:         end if
17:     end while
18: end while
```

outer *while* loop stops only when all loads have been scheduled to be processed on computing nodes. To compute the scheduling decisions for a processing phase, Algorithm 1 is repeatedly executed as shown in lines 8–17. In each iteration of this inter *while* loop, Algorithm 1 selects one load, one computing node and computes the size of a chunk of the selected load to be processed. Let load j^* be selected and computing node i^* process a chunk of size $C_{j^*,i^*,k}$. The algorithm then updates the number of remaining units of the selected load j^*, the new ready time of node i^* for receiving a new load chunk and the new ready time of node i^* for processing a new load chunk (see lines 11–13 of Algorithm 2). It is to be noted that updating $R_{i^*}^{\text{rec}}$ is simply adding up an amount of time as follows:

$$R_{i^*}^{\text{proc}} \leftarrow R_{i^*}^{\text{proc}} + T^{\text{add}}, \tag{3.17}$$

where T^{add} is defined as follows:

$$T^{\text{add}} = \begin{cases} \dfrac{C_{j^*,i^*,k} S_{j^*}^{\text{unit}}}{B_{i^*,g}} + \dfrac{C_{j^*,i^*,k} C_{i^*,j^*}^{\text{unit}}}{W_{i^*}} & \text{if node } i^* \text{ has never been assigned a load chunk;} \\ \dfrac{C_{j^*,i^*,k} C_{i^*,j^*}^{\text{unit}}}{W_{i^*}} & \text{otherwise.} \end{cases} \tag{3.18}$$

If Algorithm 1 results in an empty solution, it means that no more loads need to be processed or no more computing nodes are available for processing loads during phase k. The algorithm exits the inter *while* loop and starts a new iteration of the outer *while* loop for new processing phase $k + 1$ or finishes the scheduling, i.e., complete Algorithm 2, if no more loads are remaining to be processed.

3.6 Dynamic Scheduling Strategy

3.6.1 Overview of Dynamic Scheduling Strategy

In this section, we present a Dynamic Scheduling Strategy (DSS), which also uses the phase-based scheduling approach presented in Section 3.3 to schedule loads on computing nodes. It is an iterative strategy in which Algorithm 1 is repeatedly run until all loads are completely processed. In practice, it is obvious that DSS continues to run forever since loads are submitted to the gateway at an arbitrary arrival rate and time. Thus, the scheduler may stay idle when there is no load for scheduling and becomes active again when new loads arrive. The dynamic characteristic of the proposed scheduling strategy is twofold.

First, DSS reacts immediately to new released nodes. Knowing that the scheduler is not aware of the release times of computing nodes, the scheduling decision is dynamically generated and applied to respective processing phases. On one hand, generating the scheduling decision well ahead of the actual transmission and processing as carried out in Static Scheduling Strategy [Kang et al., 2014b], results in poor resource utilization since the computing nodes, which are released after the decision instant, can be seldom used. On the other hand, since it has been shown in our previous work [Kang et al., 2014b], late scheduling leads to waste of computing nodes that have been already released and may stay idle. To address this issue, at the beginning of each processing phase, DSS repeatedly runs Algorithm 1 to generate the scheduling decision for all computing nodes that have been released before. Furthermore, during the runtime of the corresponding processing phase, if a new node is released, DSS will immediately run Algorithm 1 again to assign an appropriate load chunk to this new node to utilize the available time of that node until the respective processing phase ends. It is worth mentioning that, to achieve this feature, we assume that there exists a mechanism that allows computing nodes to inform the scheduler their ready status when they are released, e.g., by sending a "hello" message to the scheduler.

Secondly, DSS dynamically takes into account the ready times of computing nodes for receiving and processing a new load chunk. As previously mentioned, the distributed scheduling approach enables the parallel execution of the schedulers deployed on different gateways. This leads to the case that a scheduler does not know the ready times for receiving and processing of the computing nodes that are shared with other schedulers. Requesting the ready times of released nodes whenever running Algorithm 1 allows the scheduler to be aware of their available processing times to assign appropriate chunks. This results in a better load balancing among computing nodes. It is worth mentioning that while SSS is limited by the centralized scheduling architecture, DSS can be used in both architectures. In the centralized scheduling architecture, DSS reacts dynamically to the release of computing nodes whereas in the distributed scheduling architecture, DSS additionally applies prediction techniques to estimate the ready time of computing nodes for receiving and processing a new load chunk.

3.6.2 Detailed Description of DSS

The pseudo code of DSS is presented in Algorithm 3. Since computing nodes are released arbitrarily during the scheduling process as well as loads are submitted at an arbitrary arrival rate, we omit the presentation of the input parameters of DSS in Algorithm 3. Instead, the set of released nodes and existing loads are updated gradually during the scheduling and processing of loads. Furthermore, since

Algorithm 3 Dynamic Scheduling Strategy

```
1:  $k \leftarrow 1$; /*index of processing phase*/
2:  $stop \leftarrow 0$; /*stopping flag*/
3:  $start \leftarrow$ GetSystemTime();
4:  while $stop \neq 1$ do
5:      Update set of released nodes, $\mathbb{N}$;
6:      Update set of existing loads, $\mathbb{L}$;
7:      for $j = 1 \to |\mathbb{L}|$ do
8:          $N_j \leftarrow \lceil S_j / S_j^{\text{unit}} \rceil$; /*number of load units*/
9:      end for
10:     for $i = 1 \to |\mathbb{N}|$ do
11:         Request node $i$ for ready times, $R_i^{\text{rec}}$ and $R_i^{\text{pro}}$;
12:         if $C_{i,j}^{\text{unit}}, j = 1, \ldots, |\mathbb{L}|$ is not available then
13:             Request node $i$ for $C_{i,j}^{\text{unit}}, j = 1, \ldots, |\mathbb{L}|$;
14:         end if
15:     end for
16:     $i^*$, $j^*$, $C_{i^*,j^*} \leftarrow$ Run Algorithm 1 with $N_j$, $C_{i,j}^{\text{unit}}$, $S_j^{\text{unit}}$, $T_i$, $W_i$, $B_i$, $R_i^{\text{rec}}$, $R_i^{\text{pro}}$ and $k$, $i =$
        $1, \ldots, |\mathbb{N}|, j = 1, \ldots, |\mathbb{L}|$;
17:     if $(i^* \neq 0) \wedge (j^* \neq 0)$ then
18:         Subtract load $j^*$ with a chunk of size $C_{i^*,j^*}$ and assign to node $i^*$;
19:         $S_{j^*} \leftarrow S_{j^*} - S_{j^*}^{\text{unit}} C_{i^*,j^*}$;
20:     else
21:         if $j^* = 0$ then
22:             $stop \leftarrow 1$;
23:         else
24:             Stay idle in $\tau$ units of time;
25:             if GetSystemTime() - $start \geqslant kT^{\text{phase}}$ then
26:                 $k \leftarrow k + 1$;
27:             end if
28:         end if
29:     end if
30: end while
```

the output of DSS, i.e., the scheduling decision, is realized immediately during each processing phase, we therefore do not need to store all scheduling decisions until DSS finishes.

Assuming that DSS starts with the first processing phase, i.e., $k = 1$, the scheduling process is enclosed in a *while* loop in which k will be incremented gradually. The *while* loop is terminated only when there are no more loads to be processed, i.e., the *stop* variable is set to 1. At the beginning of the *while* loop, DSS updates the set of released nodes and the set of existing loads including any new submitted loads and the loads that have been partially processed (lines 5 and 6 in Algorithm 3). Let $\mathbb{N}$ and $\mathbb{L}$ denote the set of released nodes and existing loads, respectively. Given the set of existing loads, the number of remaining units is then computed for all loads (line 8). For each released node, the scheduler requests for its ready times for receiving and processing a new load chunk. Furthermore, if the scheduler does not know the computation requirement of any existing load on this computing node, it then also requests for this parameter. This is shown from line 10 to line 15. Next, in line 16 of Algorithm 3, DSS runs Algorithm 1 to obtain a scheduling decision: determining a computing node and a load of which a chunk will be processed. If DSS successfully obtains node i^*, load j^* and chunk size C_{i^*,j^*}, this load chunk will be then assigned to node i^* immediately. The size of the selected load, i.e., load j^*, is also updated to a new smaller size (see line 19 of Algorithm 3).

If Algorithm 1 cannot obtain a solution, DSS will check different conditions to perform following further actions. If there is still available node, i.e., $i^* \neq 0$, however, there is no more loads to be processed, i.e., $j^* = 0$, DSS then exits the *while* loop by setting $stop = 1$. Conversely, if there is no more available node, i.e., $i^* = 0$, DSS stays idle for τ units of time to wait for new released nodes that will inform the scheduler their ready status in a parallel process. We can set the value of τ in the order of several seconds to be able to react when new nodes are released. Furthermore, the scheduler should check the time for initializing a new processing phase. To do so, we assume that there exists a function called **GetSystemTime()** that returns the current time of the system. Before starting the *while* loop, we initialize a $start$ variable as the system current time as shown in line 3. Thus, checking to initialize new processing phase is just to evaluate the elapsed time from the beginning of the scheduling process to the current instant as shown in line 25 of Algorithm 3.

3.7 Performance Study

3.7.1 Performance of Algorithms in Centralized Scheduling Architecture

3.7.1.1 Simulation Setup

In all simulations, we consider a multi-cloud system with 5 gateways. For all divisible loads arriving to storage nodes, we uniformly generate their sizes in the large range of $[500, 3000]$ which can be considered as the size of files (in MB) stored in a cloud storage system. The per unit required computation of loads is also uniformly generated in the range of $[1, 5]$. Both parameters assure that loads are heterogeneous in terms of size and computation requirement. We set the per unit load size to $S_j^{\text{unit}} = 1, \forall j$ (MB). Without explicitly indicating, we set the number of computing nodes to $N = 100$ for all simulations. The network topology is randomly generated. Each computing node can be connected to one or several CSPs. The bandwidth of dedicated links connecting computing nodes and CSPs is set to 16 or 32 Mbps. The computing capacity of computing nodes is randomly generated in the range of $[10, 50]$ which also assures the heterogeneity of computing nodes. All computing nodes are released randomly in the time interval from 0 to 1000. The phase length is set to $T^{\text{phase}} = 200$s, except explicitly indicated.

To evaluate the performance of the proposed scheduling algorithms, four simulations were examined. We measure the total processing time of the system to completely process all loads with respect to:

- The number of computing nodes;
- The number of arriving loads;

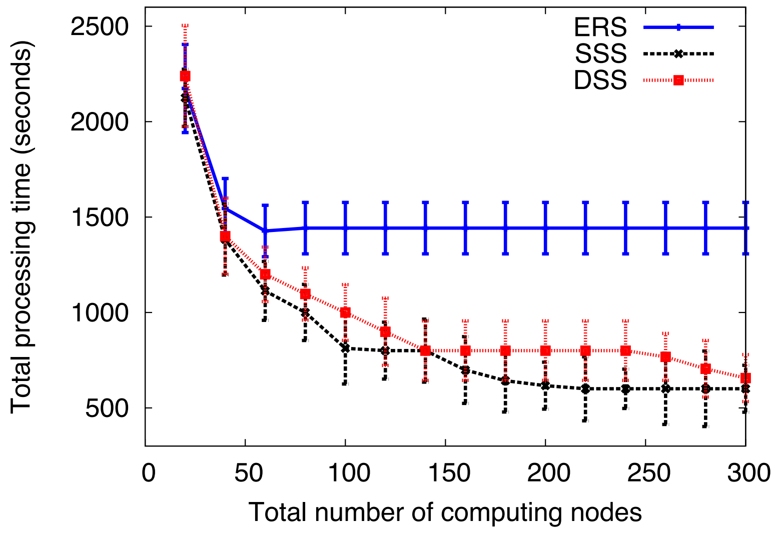


Figure 3.4: Impact of number of computing nodes.

- The phase length; and
- The network topology.

For all simulations, we compare the performance of the proposed algorithms, SSS and DSS, with a baseline strategy namely *Earliest Release Strategy* (ERS). ERS assigns the biggest load to the earliest available computing node. Once decision is made, the whole selected load will be processed by the selected computing node. If a computing node is selected, its earliest available time for the next load is the time instant when it finishes the processing of the current load. It is to be noted that ERS has been widely used in realistic systems such as Amazon EC2 or Google Cluster. This algorithm is simple and does not need heavy computation before assigning the load to computing nodes. It is considered as the First Come First Served algorithm. We believe that using a naive algorithm will be better to show the effectiveness of the proposed algorithm.

3.7.1.2 Analysis of Results

Impact of number of computing nodes: To observe the impact of the number of computing nodes, we perform the simulation with a total of 100 loads distributed to 5 gateways. We vary the number of computing nodes from 20 to 300. The additional computing nodes are added to the existing system when we increase the number of computing nodes. It is to be noted that the release time of new computing node should be at least the moment they are connected to the system or later. The load sizes and required computation are kept the same for all runs of the simulation when changing the value of N. Fig. 3.4 shows the total processing time with respect to the number of computing nodes for all scheduling strategies. It is expected that with the same number of loads, the larger number of computing nodes, the shorter the total processing time. However, when the number of computing nodes is more than enough, the total processing time does not decrease anymore. For instance, with a total of 100 loads, ERS needs 100 computing nodes. Thus, adding more computing nodes which are released later than the first 100 computing nodes does not result in any improvement of the total processing time.

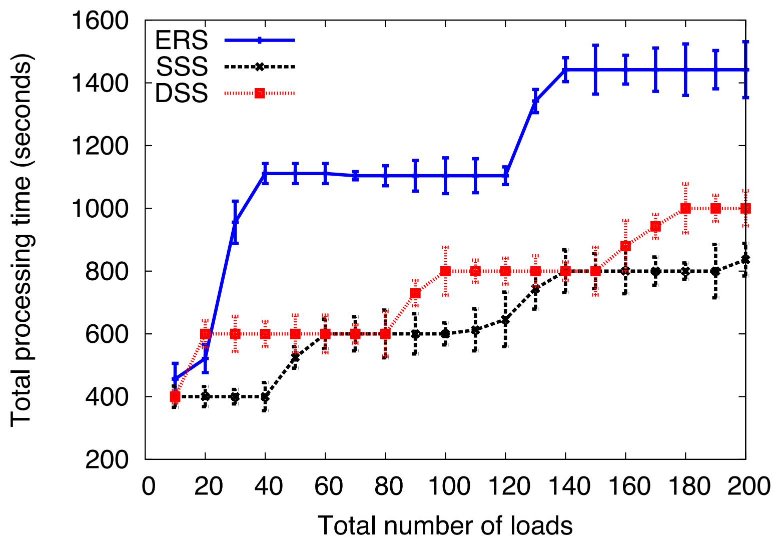


Figure 3.5: Performance of algorithms with respect to number of loads.

Comparing the performance among strategies, we see that SSS and DSS have better performance than that of ERS. Due to the use of the phase-base scheduling approach, all computing nodes are utilized and they all finish processing loads in the same period. ERS has the worst performance since it does not guarantee the load balance among computing nodes. One node may be heavily utilized while others process small loads and stay idle until all loads are processed. It is also clear that SSS has better performance than DSS since it knows the release time of computing nodes prior to the start of the scheduling process. However, this assumption may not be realistic and thus we have to use DSS for scheduling loads. In the ideal case, DSS may have the same performance as SSS, e.g., with 140 computing nodes. In the worst case, the total processing time with DSS increases 33.33%.

Impact of total number of arriving loads: In this simulation, we evaluate the impact of the number of loads arriving to the system. We perform the simulation with a total of 100 computing nodes. The total number of loads arriving to the system is increased from 20 to 200 loads, which are evenly distributed among gateways. In Fig. 3.5, we present the total processing time of all scheduling strategies with respect to the total number of loads submitted to the system. It is expected that with a fixed number of computing nodes, the total processing time increases when there are more loads submitted to the system. However, the increase is not gradual since new arriving loads can be accommodated within the idle time of computing nodes after finishing the processing of earlier arriving loads. For instance, the total processing time keeps unchanged when the number of loads increases from 20 to 80 with DSS. In any case, SSS always has the best performance and ERS has the worst performance.

Impact of the duration of a processing phase: An important parameter affecting the performance of SSS and DSS is the length of a processing phase. Choosing a sub-optimal phase length may create a poor performance of the scheduling strategies. Indeed, we perform the simulation by varying the length of a processing phase. In the first case, we assume that there is no transition overhead between processing phases, e.g., additional time to divide loads to multiple chunks, or the transition when computing nodes change from one load to another. In Fig. 3.6, we present the total processing time with respect to the

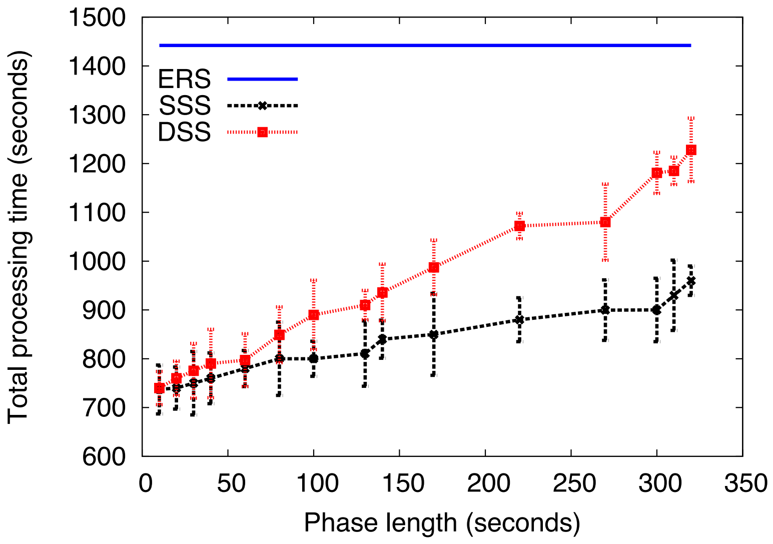


Figure 3.6: Impact of phase length without transition overhead.

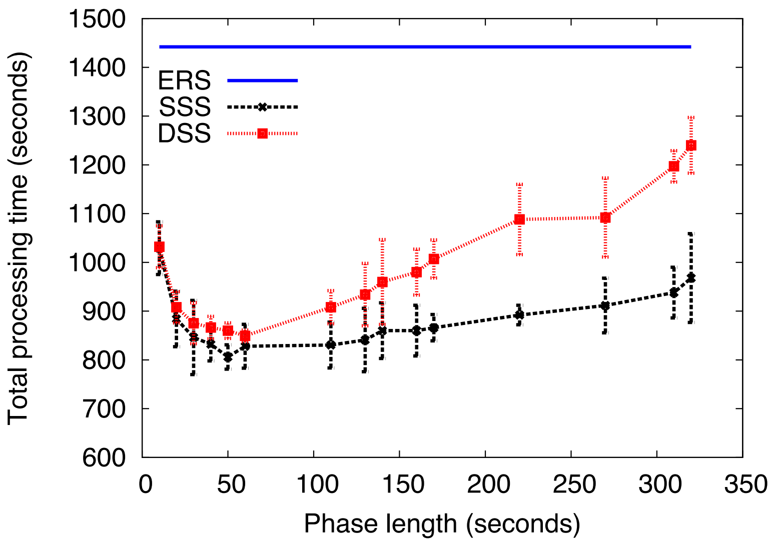


Figure 3.7: Impact of phase length with transition overhead.

length of a processing phase. It is observed that when the phase length is increased, the total processing time also increases. There are two reasons behind this behavior. First, the time taken in waiting for communication of load chunks assigned in the first scheduling becomes considerable when there is only a few processing phases. Second, it is due to the load unbalance at the last processing phase. The first available computing node may be assigned a larger load chunk to process until the end of the phase while other computing nodes, being available later, process the remaining small load chunks. Compared to ERS, we see that ERS performance can be considered as the upper bound of SSS and DSS since ERS does not use phase-based approach, i.e., the phase length in ERS is $T^{\text{phase}} = \infty$.

In the second case, we assume that there is a transition overhead between phases. We define a function computing the transition overhead with respect to the number of processing phases and total number of loads as $T^{\text{overhead}} = \alpha k J$ where α is a weighting coefficient, k is the number of processing phases and J is the total number of loads. In Fig. 3.7, we present the change of the processing time with respect to the processing phase length. We can observe that when the phase length is too small, the number

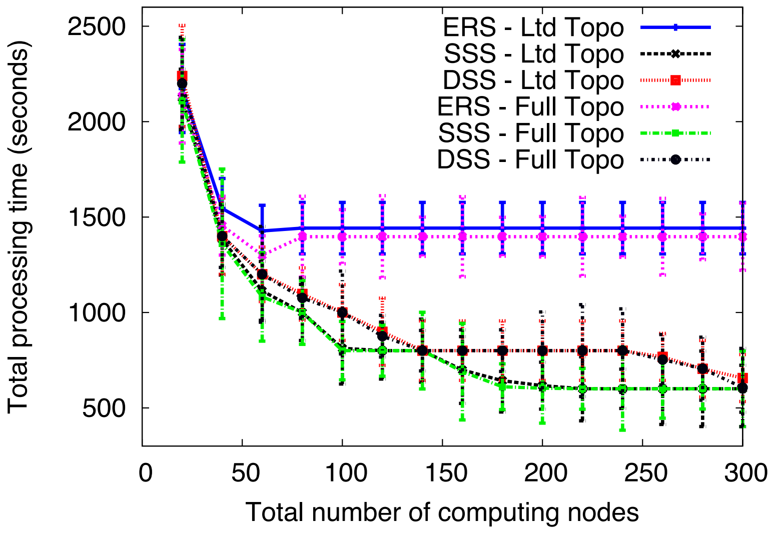


Figure 3.8: Impact of the network topology.

of processing phases is high. Thus, the total processing time dominates due to the transition overhead between phases. When we increase the phase length, the number of processing phases decreases thereby decreasing the total processing time. The total processing time is minimized at an optimal value of the phase length, a value in the interval $[50, 80]$, before increasing again when we increase the phase length. When implementing SSS and DSS for real-world applications, simulations and/or real experiments can help users to determine an optimal value of phase length depending on the given parameters such as size, per unit size and computation requirement of loads.

Impact of the network topology: In the last simulation, we evaluate the impact of the network topology: a limited topology (Ltd Topo) where each computing node may not be connected to all gateways by the dedicated links of 16 or 32 Mbps, and a full topology (Full Topo) where each computing node is connected to all gateways with the bandwidth of 32 Mbps. This situation reflects the similar case in real-world that a computing node may or may not able to process all types of loads due to the availability of software or license. As shown in Fig. 3.8, for all scheduling strategies, the performance with the full topology is slightly better than that with the limited topology. This is due to the fact that with the full topology, the scheduler has more choices to select a computing node for a load of a certain gateway. However, even using the highest bandwidth, the improvement is not significant since most of load transmissions are overlapped with load processing.

3.7.2 Performance of DSS in Distributed Scheduling Architecture

3.7.2.1 Simulation Setting

To evaluate the performance of DSS in the distributed scheduling architecture, our testbed also comprises a multi-cloud system with 5 gateways hosting 5 respective schedulers, which operate in parallel to receive and schedule loads. Without loss of generality, we assume that the schedulers are connected with 100 computing nodes with two kinds of link bandwidths: 16 Mbps and 32 Mbps, respectively. As described in earlier sections, as per our distributed design, it is to be noted that many computing nodes

Table 3.2: Number of Computing Node That a Particular Scheduler can Access

Scheduler	**Number of Computing Nodes**
S_1	59
S_2	72
S_3	73
S_4	67
S_5	64

are shared among schedulers. In Table 3.2, we present the number of computing nodes that a particular scheduler can access. For all the simulations, we vary the arrival rate of loads, i.e., the number of loads that are submitted to each scheduler per minute, and we measure the total processing time of all loads after 5 minutes of receiving loads. It is also to be noted that while we assume the availability of computation requirement of loads in the evaluation of SSS, we consider the realistic scenario in this simulation to evaluate the performance of DSS such that the computation requirement of loads is also unknown to schedulers and computing nodes, which therefore need to use the prediction technique presented in Section 3.4 to estimate this parameter when required.

3.7.2.2 Load Setup and Training Dataset for Prediction Algorithm

To the best of our knowledge, there is not any publicly available processing workload traces since such information is often regarded by providers as being strictly confidential. Recently, Google has published a dataset pertaining to workloads on Google Compute Clusters [Reiss et al., 2011], which includes the resource requirements of tasks submitted to a cluster of $12,000$ physical machines over a time period of 29 days. While this workload trace provides information on the timestamp of workloads processed, it does not provide enough information on the size of tasks, i.e., the size of input data and application type of each task. This prevents us from adopting this workload trace directly in our work. Fortunately, we can use the data produced by our previous work [Kang et al., 2014a]. In this work, we designed and implemented an encryption service for cloud storage system to protect the data privacy. The experimental validation was performed on real servers and data files that are archives of `Wikipedia`. The processing traces keep the information of all files sent to the storage server including the file size, the key length used for encryption algorithm and the time overlap for the encryption service. Note that, in this application, i.e., encryption service, the application requires different amount of computation for the same file when the key lengths are different. The longer key requires longer time to complete the encryption of the same file. The processing traces were generated with three key lengths: 256, 192 and 128 bits. In Fig. 3.9, we present the encryption time with 256-bit keys with respect to the file size of input data on a physical machine with PowerEdge C6220 with Intel(R) Xeon(R) Processor E5-2640 2.50GHz, 24GB RAM. The file size varies between 41MB and 4.1GB.

3.7.2.3 Analysis of Results

Overall Performance of Proposed System: In this section, we evaluate the overall performance of the proposed system and algorithms by comparing its performance with that of a baseline scheme, ERS. It

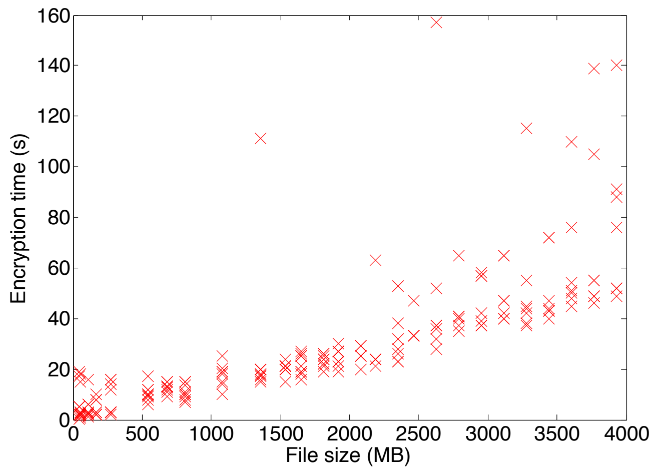


Figure 3.9: Training data for the prediction algorithm, i.e., encryption time with respect to the file size with the key length of 256 bits.

is worth mentioning that ERS assigns the biggest load in terms of size to the earliest released computing node. For each arrival rate, DSS and ERS were executed more than 100 times to obtain the average total processing time. In Fig. 3.10, we present the total processing time of DSS and ERS with respect to the arrival rate of loads. Generally, the results show that DSS outperforms ERS and DSS can reduce the total processing time up to 44.60% compared to ERS. The improvement is more significant when arrival rate of loads is low, e.g., cases of 2, 4 and 6 loads per minute, since many computing nodes will be wasted with ERS. These results lead to two following affirmations. First, the use of the phase-based scheduling approach definitely improves the resource utilization in a computing infrastructure. By dividing loads into multiple smaller chunks and assigning to different nodes in parallel, it significantly reduces the total processing time of loads and also efficiently exploits the capacities of computing nodes. Secondly, without the knowledge on the computation requirement of loads, the use of prediction models effectively helps the scheduler to assign appropriate chunks to computing nodes depending on their capacities and the computation requirement of loads. This is one of the important contributions of this chapter.

The above affirmations are further vouched by Fig. 3.11 wherein we plot the scheduling solutions of DSS and ERS with the arrival rate of loads at 14 loads per minute. The horizontal axis represents the time and the vertical axis represents computing nodes, i.e., labeled by "CNs". For each computing node, two horizontal lines are depicted: one for transmission of load chunks and the other for processing of load chunks. The color art is just to make difference between the transmission and processing of two consecutive load chunks assigned to the same node: grey and blue are for transmission and light blue and red are for processing. It is observed that DSS generates better scheduling solution, shown in Fig. 3.11a, where all the computing nodes finish processing load chunks almost at the same time. Considering the case of ERS shown in Fig. 3.11b, the scheduling solution does not result in a good load balance among computing nodes. While several computing nodes finish the processing of assigned loads early and stay idle in a long period, other computing nodes suffer heavy loads and prolong the total processing time. It is to be noted that the use of the phase-based scheduling approach that divides loads into multiple smaller load chunks, needs an important parameter, namely the computation requirement of loads. As previously mentioned, this parameter is unknown to schedulers and computing nodes and it is estimated by using

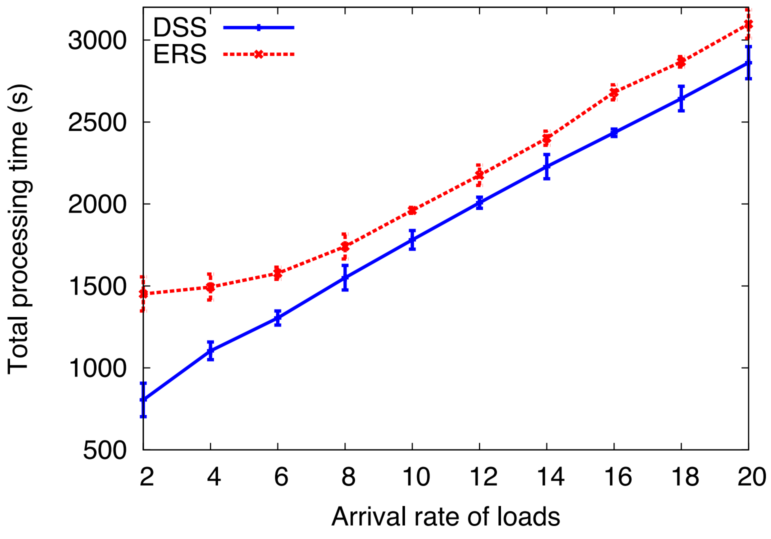


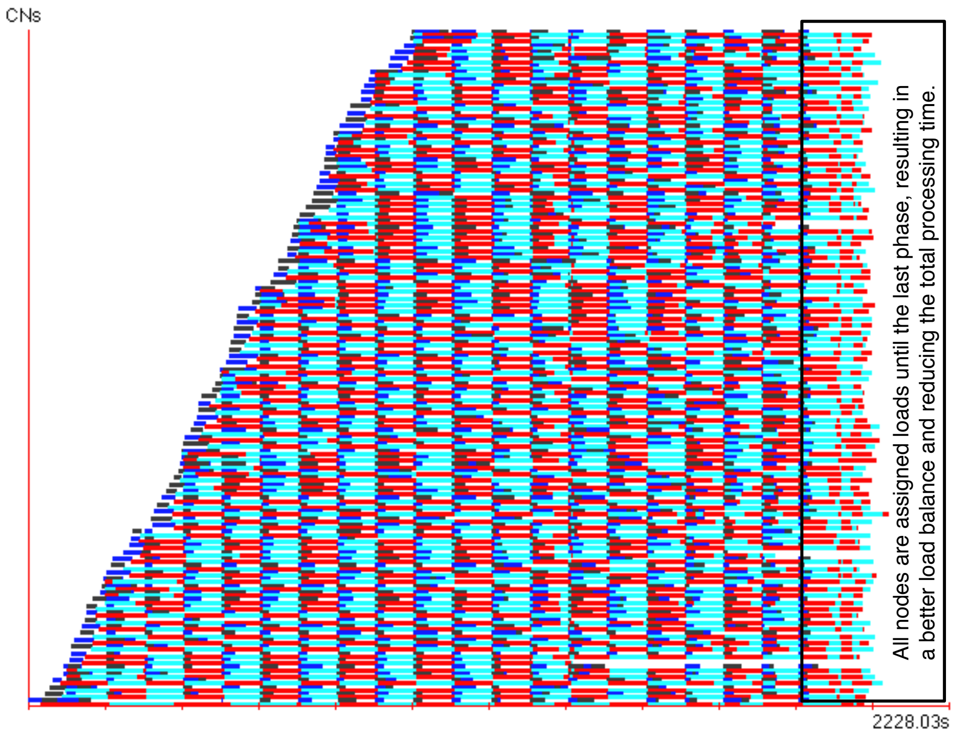
Figure 3.10: Total processing time of the proposed system using DSS compared to baseline scheme, ERS, with respect to arrival rate of loads.

a prediction technique. The simulation results further demonstrate the effectiveness of the prediction technique used in the proposed system. Due to the accuracy of the prediction technique, the size of load chunks is appropriately determined based on the computation requirement of loads and capacities of computing nodes. Computing nodes therefore are well balanced and they can finish processing load chunks almost at the same time at the last phase.

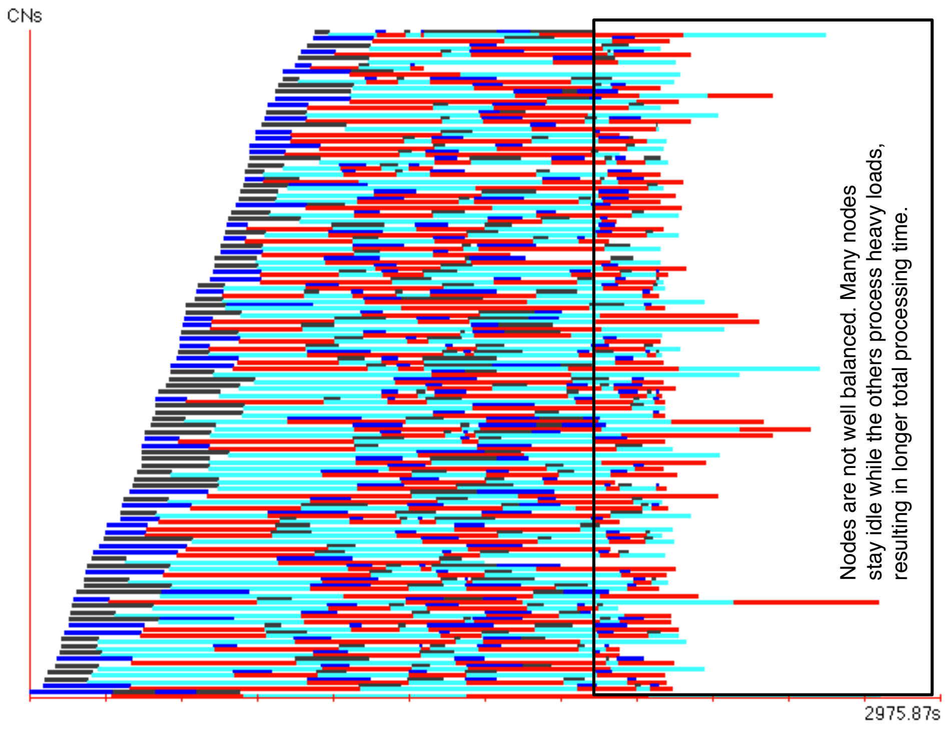
It is to be noted that we do not have a specific balancing index in this work although the load balance index of the proposed algorithms can be observed from Fig. 3.11. As shown in Fig. 3.11b, nodes are not well balanced when we apply the baseline algorithm, whereas Fig. 3.11a shows the better load balance by using the proposed algorithms since all nodes are assigned loads until the last phase evenly.

Effective Utilization of New Released Nodes: In this section, we evaluate the improvement in DSS compared to the version presented in [Kang et al., 2014b]. As mentioned in Section 3.6.1, at the beginning of each phase, Algorithm 1 is repeatedly executed until no more computing nodes, which are available to process load chunks in the corresponding phase. With the version presented in [Kang et al., 2014b], DSS waits until when the next processing phase starts to run Algorithm 1 again. It is to be noted that the running time of Algorithm 1 is negligible compared to the duration of a processing phase. Waiting until next processing phase makes the computing nodes, which are released after the scheduling of the corresponding phase finishes, be wasted until the next phase. We improve the utilization of computing nodes by starting assigning loads to nodes immediately after they are released. In Fig. 3.12, we present the total processing time with the original version and the improved version with respect to the arrival rate of loads. It is observed that the improved version of DSS has a better performance. In the case of the simulation shown in Fig. 3.12 where the duration of processing phase is 100s, the improved DSS can reduce the total processing time by up to 7.14%.

The improvement will be more significant when the duration of processing phase is large. In Fig. 3.13, we present the total processing time of DSS with and without the improvement when the duration of processing phase is 200s. The improved DSS can reduce the total processing time by up



(a) Scheduling solution of DSS.



(b) Scheduling solution of ERS.

Figure 3.11: Scheduling solutions of DSS and ERS with arrival rate of loads at 14 loads per minute.

to 30.24%. For better understanding, we show in Fig. 3.14 the scheduling solutions of DSS with and without the improvement during the period where computing nodes are released. Fig. 3.14a shows the scheduling solutions of the improved DSS where all the computing nodes are utilized immediately after

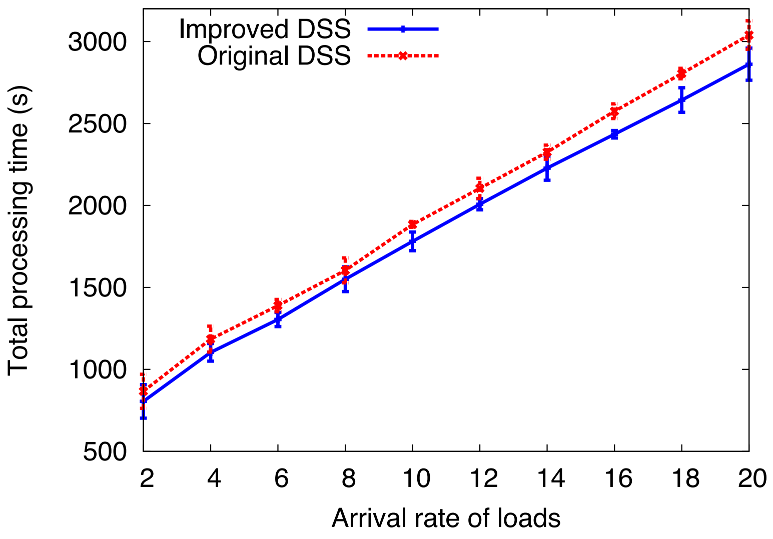


Figure 3.12: Total processing time of proposed system using DSS with and without improvement. Duration of processing phase is 100s.

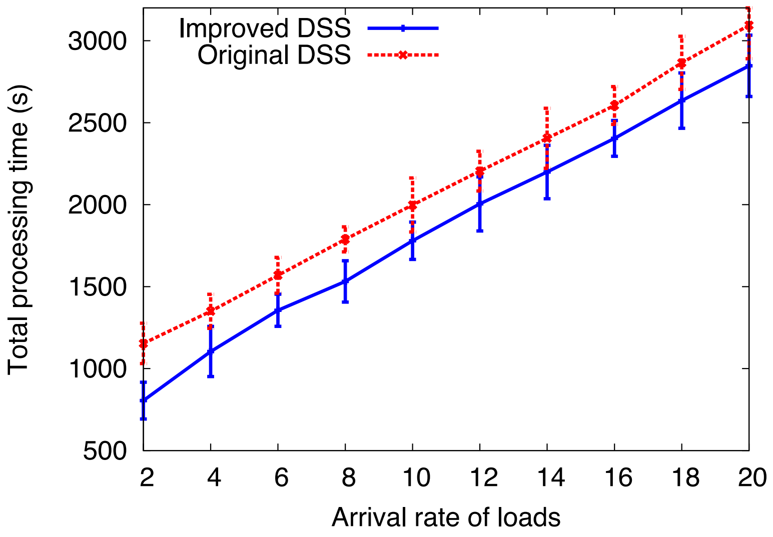
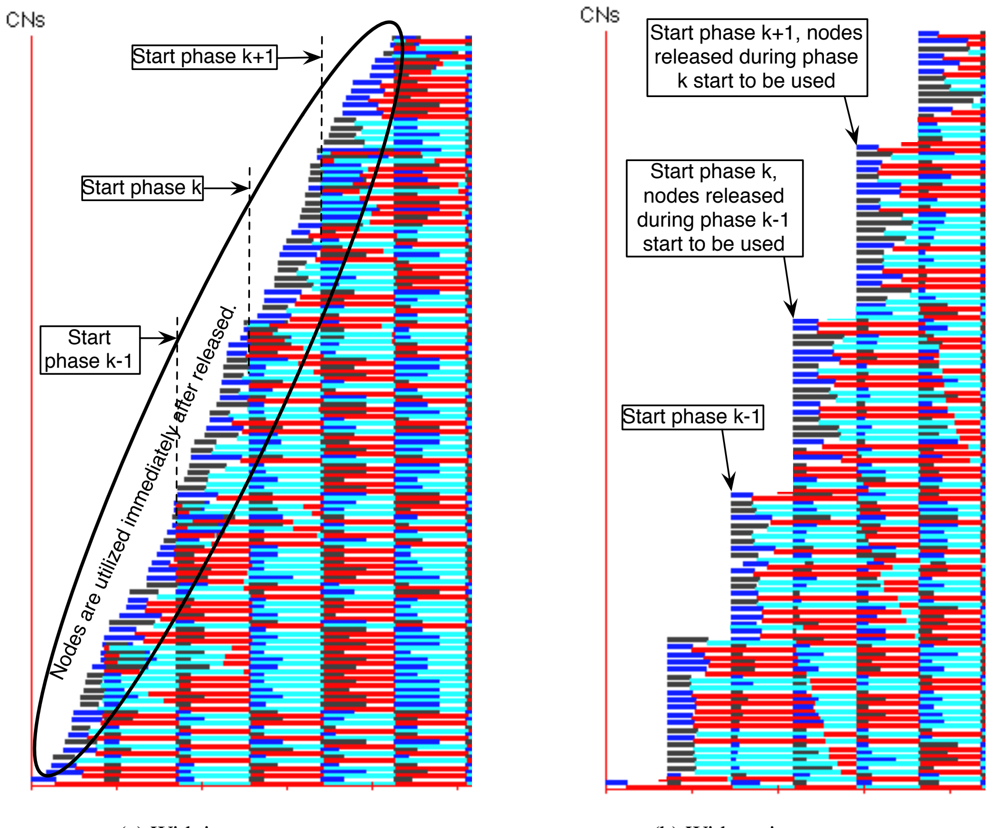


Figure 3.13: Total processing time of proposed system using DSS with and without improvement. Duration of processing phase is 200s.



(a) With improvement. (b) Without improvement.

Figure 3.14: Scheduling solutions of DSS with and without improvement: processing phase is 200s.

they are released whereas Fig. 3.14b shows the scheduling solutions of DSS without the improvement where computing nodes are utilized only from the next phase. The results show the importance of this gain in the scheduling algorithm. It eliminates the dependence of the resource utilization on system parameters, i.e., duration of processing phase that usually varies depending on application types.

Centralized Scheduler vs. Distributed Schedulers: In this section, as a matter of natural curiosity, we compare the performance of the proposed algorithm, DSS, in the centralized scheduling architecture and the distributed scheduling architecture. It is worth mentioning that, in the case of centralized scheduler, when loads arrive to the gateways, their information (application type, size, unit size) are immediately sent to the centralized scheduler. In Fig. 3.15, we present the total processing time of the proposed system using these two scheduling approaches. It is expected that the centralized scheduling approach has a better performance than that of the distributed scheduling approach. Since the centralized scheduler holds the information of the entire system, especially the availability of all computing nodes, no synchronization is needed during the scheduling process. With the distributed scheduling approach, a node, that belongs to more than one gateway, has to communicate with all corresponding schedulers. Therefore, the synchronization is needed to avoid conflict when several schedulers request for ready time of the node at the same time. This synchronization incurs a delay for load processing. The results show that the centralized scheduling approach can reduce the total processing time by up to 6.5%.

While the gain in the total processing time is only in the order of 6.5%, the scheduling load posed on

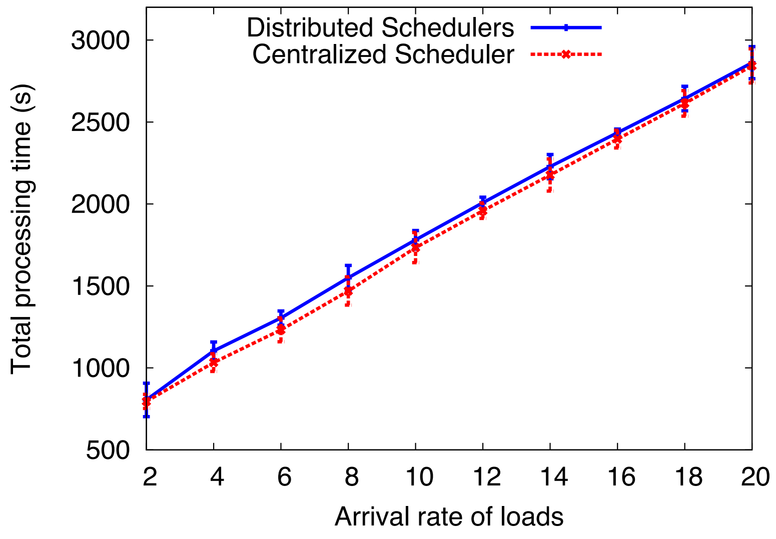


Figure 3.15: Total processing time of DSS using centralized scheduler or distributed schedulers.

the centralized scheduler increases with the factor of the number of gateways and arrival rate of loads. Unfortunately, simulating the load at schedulers to measure the scheduling delay is beyond the scope of the available simulation testbed. We believe that the distributed scheduling approach will dominate due to its effectiveness in fault-tolerance and load balancing. It will be scalable when realizing the proposed approach in a realistic system with many gateways and high arrival rate, e.g., a multi-cloud system with more than 10 gateways and 1000 loads arriving per unit of time.

3.8 Chapter Summary

In this chapter, we presented next generation novel multi-cloud architectures to address heavy cloud resource requirements from users who want to use cloud infrastructure for their modern and complex applications. Based on these architectures, two scheduling strategies, which leverage on the divisible load theory and the phase-base scheduling approach, are then proposed: A Static Scheduling Strategy (SSS), which assumes that the release time of computing nodes is known prior to the scheduling and A Dynamic Scheduling Strategy (DSS) relaxes the assumption of SSS by applying a prediction technique to estimate the release time of computing nodes [Kang et al., 2014b, Kang et al., 2016c]. Both strategies take into account the topology and capacity of the system network, and the heterogeneity of computing nodes. This is an important contribution to the design of multi-cloud system schedulers that are capable of handling multiple large-scale loads. Nevertheless, this chapter addresses only the performance issue of multi-cloud systems, particularly the problem of scheduling loads to achieve load balancing and to minimize the total processing time. Moving data to public clouds for storing and processing leads users to other challenging issues. Among them, the security issue (especially the data privacy and availability) is the most important since it is hindering the migration of data and its computations to public clouds. This motivates us to carry out the research on the problem of guaranteeing the data availability and protecting the data privacy in public clouds. We present our study on these issues in the next chapter.

Chapter 4

A Security-aware Data Placement Algorithm for Data Privacy and Availability in Cloud Storage Systems

In the previous chapter, we focused only on the problem of performance optimization for multi-cloud systems without considering the security requirement to protect the data privacy. In this chapter, we move forward to address both issues in a joint problem. We consider the problem of data placement in cloud storage systems where the performance of the system represented by the total retrieval time of a data should be minimized and the data privacy should be protected. We first develop an optimization programming formulation for the problem whose objective is minimizing the total retrieval time of a data and the constraints are the security and availability requirements. We then develop an efficient heuristic algorithm, namely Availability and Security-awarE Data placement algorithm for cLOUd storage Systems (A-SEDuLOUS) to solve the problem. A-SEDuLOUS divides a data into multiple chunks, each will be stored in two nodes as primary and backup nodes. The placement decision is computed in the way that all the primary and backup chunks of a data satisfy the security and availability requirements, and the total retrieval time is minimized. We evaluate the proposed algorithm through comprehensive simulations on random-network-topology systems and the Internet2-topology system.

4.1 Research Motivation and Objectives

To meet the ever-growing data storage demands from users, cloud storage systems have become an attractive candidate that provides online services for data storage with immense capacity and high quality of service but at a low cost [Zeng et al., 2009]. Many commercial cloud storage providers, which include Amazon Simple Storage Service (S3), Google Cloud Platform, Dropbox, etc., have joined the cloud market and they provide a simple Web interface that can be used to store and retrieve any amount of data at any time and from anywhere on Earth [Wang et al., 2010]. From a user's point of view, delay in data retrieval and data availability are the two important criterion that evaluate the quality of service or performance of cloud storage systems. However, providing a fast data retrieval storage service is a challenge for any cloud storage provider. The problem is more complex when considering a large-scale cloud storage system where storage nodes are geographically distributed with heterogeneous storage capacity of the nodes and bandwidth of the links connecting the nodes. On the data availability, many cloud providers offer the data replication with multiple data copies that are stored geographically. However, some of them still suffer the system outage such as Google Mail and Hotmail [Albanesius, 2012, Cachin et al., 2009]. While the replication can address only the attacks on storage nodes, it may not be able to solve the problem of attacks on the links connecting the storage nodes to the end-node of users making the data unavailable if there is no alternate network path for data transmission. The problem of placing the primary copies and backup copies of a data therefore needs to consider many other parameters to obtain an optimal placement solution that minimizes the retrieval time and guarantees the data availability regardless of attacks on nodes or links.

In addition to the quality of service issue, moving data from local servers to public cloud storage systems poses new challenges in security and data privacy. Without the knowledge of physical location of data and the control over the storage servers, users are concerned about the leakage of sensitive data and malicious behaviors of other users sharing the same cloud service. The consequence of revealing sensitive data is very severe in several scenarios such as revealing personal health information may lead to denied access to insurance services of the data owner. To relieve users' concerns, several cloud providers such as Amazon S3 and Google Cloud Platform offer the server-side encryption service, which encrypts the users' data according to the security level represented by the length of encryption key [Kang et al., 2014c]. However, this approach incurs a significant delay due to the encryption process and requires an additional effort from users to manage the encryption keys. A novel approach is therefore needed to guarantee the data security while minimizing the overhead of the security service.

Combination of the performance issue and the security issue makes the problem of data placement in cloud storage systems become a challenging problem, which has not been considered in the literature (to the best of our knowledge). We take the first step towards addressing the problem with this work. Given that an entire data file is divisible and sensitive information is spread over the fragments of the file, we propose to divide the data into multiple fragments (chunks), each will be stored in a separate node. The chunks of a particular data are then spread over storage nodes with a certain distance between any pair of chunks. This ensures that even in case of a successful attack to a particular chunk, no meaningful information of the entire data file is revealed and malicious users cannot guess the location of other data chunks. To address the requirement of data availability, we propose to replicate an additional backup chunk for each primary chunk. The placement of backup chunks has to satisfy not only the security requirement but also the availability requirement such that the paths from the nodes storing primary chunk and backup chunk to the access point of end-users must be link-node-disjoint. The entire data placement problem is then addressed by solving three intertwined sub-problems: decision of number of chunks, decision of size for each chunk and selection of storage nodes.

We address these three sub-problems by first formulating an optimization programming model that minimizes the total retrieval time of a data file while satisfying the security and availability requirements.

Since solving such an optimization problem is computationally hard [Baev et al., 2008], we then develop an algorithm, namely Availability and Security-awarE Data placement algorithm for cLOUd storage Systems (A-SEDuLOUS) to solve the problem. It uses a greedy approach that gives the priority for the nodes, which can faster transfer data to the access point of users. To address the security challenge, we propose to apply the graph T-coloring approach [Hale, 1980], which ensures that two adjacent nodes are not assigned the same color. This implicitly means that they should not simultaneously store the chunks of a data. For each data, A-SEDuLOUS determines the primary placement decision by computing the number of primary chunks, chunk sizes and selecting primary nodes. Based on the placement decision of primary chunks, it determines a list of backup nodes and the assignment of chunks to backup nodes. While the placement of primary chunks needs to satisfy only the security requirement, the placement of the backup chunks needs to satisfy both security and data availability requirements. We evaluate the proposed algorithm through extensive simulations on different cloud storage systems using random network topologies and the realistic Internet2 topology. We compare A-SEDuLOUS against baseline algorithms to demonstrate its effectiveness.

4.2 System Model

We consider a distributed cloud storage system that consists of M storage nodes. Node i has a total storage capacity of C_i units. The connections among nodes are represented by symmetric matrix $B(M \times M)$ where $B_{i,j} = B_{j,i}$ indicates the bandwidth of the bi-directional link between node i and node j. Thus, the case of $B_{i,j} = B_{j,i} = 0$ means that node i and node j are not connected. The topology of the system is represented as a graph $G(V, E)$ where V is the set of vertices, i.e., storage nodes, and E is the set of edges among vertices, i.e., the links that connect the storage nodes. We assume that any storage node can be considered as an access point of the system where users can submit storage requests and retrieve their data. This assumption reflects the practical situation of a distributed storage system such as Amazon S3, in which storage nodes are geographically distributed over the world and some of them are used as access points of the system at the respective region.

Given a user request that requires to store a volume of data sized D, we assume that the data is divisible into multiple chunks with arbitrary size, each chunk contains partial important and sensitive information of the entire data. Thus, revealing the information of a single chunk does not make sense to malicious users. To address the data availability issue, we also assume that each chunk will be replicated with an additional copy. The challenging issue is how to store all data chunks including primary chunks and backup chunks in the storage system to minimize the total retrieval time while satisfying the security and availability requirements. By satisfying the security requirement, the system guarantees that even if a successful intrusion to a chunk happens, the attacker cannot guess or infer the locations of other chunks. By satisfying the availability requirement, the system ensures that in case of attack happening on any storage node or network link, which connects the storage node and the access point of the user, the backup chunk of the affected primary chunk is always accessible for data retrieval. It is worth mentioning that we consider a single attack or failure scenario in our work. This means that at a given time instant, there is maximum one storage node or network link that are unavailable due to attacks. This prevents the situation that both primary and backup chunk of a data are unavailable simultaneously due to two nodes or two links are attacked at the same time.

Since the security requirement of different users may be different, we define the security level as the minimum distance between the two nodes storing any pair of chunks of the data. The non-security level means the whole data file can be stored in the same node. The lowest security level means two different chunks should be stored on different nodes, which may be two adjacent nodes. We can also define a default security level such that two chunks should not be stored in two adjacent nodes. We assume that users are responsible for specifying the security level for their data, depending on the sensitivity of the data. For instance, medical and financial record data should be stored with higher security level than normal data such as movies.

4.2.1 Retrieval Time Function

Let $(S_1, S_2, \ldots, S_M)$ denote a feasible placement solution of the primary chunks of a data where $S_i \neq 0$ implies that node i is selected to store a data chunk of size S_i and node i satisfies the security requirement whereas $S_i = 0$ indicates that node i is not selected for storing a data chunk. Obviously, $\sum_{i=1}^{M} S_i = D$. We now formulate the function, which computes the retrieval time of the original data upon a read request. Let node p denote the access point where the user submits his read request. We assume that a user always use the same access point for storing and retrieving data. Data chunk S_i stored on node i is then transferred to the access point, node p, on a path that connects node i and node p and may traverse multiple intermediate nodes. We assume that the transmission path of a data chunk is the shortest path from the storage node to the access point. The shortest path is computed based on the data transmission time. It is to be noted that there exist in the literature many algorithms, which compute the shortest path between two nodes in a graph [Goldberg and Harrelson, 2005]. We thus omit the presentation of such algorithms and assume that $P_{i,p}$ is the shortest path connecting node i and node p for data transmission. The total transmission time of a data chunk sized S_i from node i to node p is computed as follows:

$$T_{S_i} = \sum_{l \in P_{i,p}} \frac{S_i}{B_{l_s,l_e}}, \tag{4.1}$$

where l is a link belonging to path $P_{i,p}$, B_{l_s,l_e} is the bandwidth of link l, l_s and l_e are the source and the sink of link l, respectively. Assuming that node p can perform only one data transmission at a time, the total transmission time of a data sized D, which is divided into multiple chunks and stored on different nodes, is defined as follows:

$$T_D = \sum_{i=1}^{M} \sum_{l \in P_{i,p}} \frac{S_i}{B_{l_s,l_e}}. \tag{4.2}$$

We assume that the read/write time from/to the storage devices are negligible. We also assume that network links are bi-directional. Thus, the data transmission time from node i to node p for a read request will be the same as the time needed for transferring the data from node p to node i for a write request. Therefore, if we can minimize the total retrieval time of a data, then its total uploading time will be also minimized.

4.2.2 Problem Statement for Primary Chunk Placement

The formal statement of the primary chunk placement problem is given as follows. "*Given a user request that requires to store a volume of data sized D submitted to node i of a distributed cloud storage system,* *the system is responsible for determining: (i) A list of nodes, each will store a data chunk, satisfying the required security level of the user; (ii) The size of the data chunk, S_i, which will be stored on node i, such that the total data retrieval time is minimized."* The optimization programming formulation of this problem is mathematically presented as follows:

$$\text{Minimize:} \quad T_D = \sum_{i=1}^{M} \sum_{l \in P_{i,p}} \frac{S_i}{B_{l_s,l_e}} \tag{4.3}$$

$$\text{subject to:} \quad \sum_{i=1}^{M} S_i = D, \tag{4.4}$$

$$S_i \geqslant 0, \qquad i = 1, \dots, M \tag{4.5}$$

$$S_i \leqslant S^{\max}, \qquad i = 1, \dots, M \tag{4.6}$$

$$S_i \leqslant C_i, \qquad i = 1, \dots, M \tag{4.7}$$

$$f(S_i, S_j, i, j) \geqslant K, \qquad i, j = 1, \dots, M. \tag{4.8}$$

The constraint in (4.4) ensures that the sum of all chunk sizes is equal to the original size of the data. The constraints in (4.5) and (4.6) ensures that the size of every chunk should not be negative and not exceed the limit that is defined by the user, denoted as $S^{\max}$. Obviously, $S^{\max} \leqslant D$. We argue that the owner of the data is the best candidate to specify the chunk size threshold to not reveal much sensitive information if a chunk is leaked. The constraint in (4.7) ensures that a chunk will be stored in a node, which has sufficient storage capacity. The constraint in (4.8) ensures that the placement solution guarantees the required security level, i.e., the minimum distance between the nodes that store two chunks of the same data, denoted as K. For instance, $K = 0$ represents the non-security level, $K = 1$ represents the lowest security level and $K = 2$ represents the default security level. We then define function $f(S_i, S_j, i, j)$ to compute such a distance. The function is defined as follows:

$$f(S_i, S_j, i, j) = \begin{cases} \mathcal{D}(i, j) & \text{if } S_i \neq 0 \text{ and } S_j \neq 0; \\ MaxInt & \text{if } S_i = 0 \text{ or } S_j = 0. \end{cases} \tag{4.9}$$

When $S_i \neq 0$ and $S_j \neq 0$, the function takes the value, $\mathcal{D}(i, j)$, which is computed as the distance, i.e., the number of hops, between the two nodes based on the shortest path in the network topology of the system. In case if $S_i = 0$ or $S_j = 0$, meaning that either node i or node j is not selected to store any data chunk, the result of the function is set to a value larger than K, e.g., maximum integer value, $MaxInt$.

4.2.3 Problem Statement for Backup Chunk Placement

Given the placement solution of primary chunks obtained by solving the optimization problem presented above, denoted as $(S_1^*, S_2^*, \dots, S_M^*)$, we now present the problem formulation of the backup chunk placement such that the total retrieval time of backup chunks will be minimized. Let $(I_1, I_2, \dots, I_H)$ denote the indices of the nodes that store a primary chunk of the data, i.e., $I_h \in [1, \dots, M]$ and $S_{I_h}^* \neq 0$. In other words, the data is divided into H chunks with size $S_{I_h}^*, h = 1, \dots, H$. We then need to determine H different storage nodes that will store all H backup chunks such that the total retrieval time of backup chunks is minimized, and the security as well as availability constraints are satisfied. Let $(I'_1, I'_2, \ldots, I'_H)$ denote the indices of the backup nodes. The problem is formulated as follows:

$$\text{Minimize: } T'_D = \sum_{h=1}^{H} \sum_{l \in P_{I'_h,p}} \frac{S^*_{I_h}}{B_{l_s,l_e}} \tag{4.10}$$

subject to:

$$I_h \neq I'_{h'}, h, h' = 1, \ldots, H \tag{4.11}$$

$$S^*_{I_h} \leqslant C_{I'_h}, h = 1, \ldots, H \tag{4.12}$$

$$\mathcal{D}(I_h, I'_{h'}) \geqslant K, \quad h \neq h', h, h' = 1, \ldots, H \tag{4.13}$$

$$\mathcal{D}(I'_h, I'_{h'}) \geqslant K, \quad h \neq h', h, h' = 1, \ldots, H \tag{4.14}$$

$$P_{I_h,p} \cap P_{I'_h,p} = \emptyset, h = 1, \ldots, H. \tag{4.15}$$

The constraint in (4.11) ensures that backup nodes are different from primary nodes. The constraint in (4.12) ensures that backup node I'_h has sufficient storage capacity to store chunk of size $S^*_{I_h}$. The constraint in (4.13) guarantees that the distance between a backup node and any primary node satisfies the security requirement where $\mathcal{D}(i, j)$ is computed as the distance, i.e., the number of hops between node i and node j based on the shortest path between them. Similarly, the constraint in (4.14) guarantees that the distance among any pair of backup chunks satisfies the security requirement. Finally, the constraint in (4.15) guarantees the data availability such that the paths from the nodes storing the primary chunk and the backup chunk to the access point are link-node-disjoint paths where nodes I_h and I'_h are the two nodes storing the primary and its backup chunk, respectively.

4.2.4 Discussion

It is worth mentioning that we separate the problem formulation of primary chunk placement and that of backup chunk placement to give the storage priority for the primary chunks. We argue that upon a read request, the access point will first request the storage nodes storing the primary chunks. Thus, the data retrieval time will be minimized if all primary chunks are available. In case of attacks on any node or network link, the access point will then request the node storing the backup chunk. Since the selected backup nodes minimize the total retrieval time among backup node candidates, the retrieval time of a particular backup chunk will also be minimized in the event of an attack.

Solving the above problems is computationally hard since it is $\mathcal{NP}$-complete in nature to determine whether there exists a valid placement solution even for a simple and small storage system [Baev et al., 2008]. The optimization programming models formulated above give us better description of the data placement problem and better understanding of the problem complexity. It shows that it is essential for us to devise efficient heuristic algorithms to solve the problem. However, developing novel and efficient heuristic algorithms is not a straightforward task. In the next section, we present A-SEDuLOUS, an Availability and Security-awarE Data placement algorithm for cLOUd storage Systems, which is a greedy algorithm that satisfies the security and availability requirements of users while minimizing the data retrieval time.

4.3 A-SEDuLOUS

4.3.1 T-coloring Problem

Given a graph $G = (V, E)$ that represents the network topology of the storage system, and a set $\mathbb{T}$ that contains non-negative integers including 0, the T-coloring problem is defined as function $f : V \rightarrow \mathbb{C}$, a mapping from set of vertices V to a set of non-negative integers, $\mathbb{C}$, which represent the values used to color the vertices, such that $|f(i) - f(j)| \notin \mathbb{T}$ where $i, j \in V$. In other words, the T-coloring problem assigns a color to a vertex such that the distance between the colors of the adjacent vertices must not belong to $\mathbb{T}$. When $\mathbb{T} = \{0\}$, the T-coloring problem reduces to a common vertex coloring problem, in which two adjacent vertices cannot be assigned the same color.

The T-coloring problem when $\mathbb{T} = \{0\}$ can naturally apply to the data placement problem to ensure complete security of data. Given that a data can be divided into an arbitrary number of chunks, with the default security level, i.e., $K = 2$, two different chunks of the data cannot be stored in two adjacent nodes. In other words, given a coloring solution of a storage node graph, all the chunks of the same data can be stored on the storage nodes, which are assigned the same color, since they are not neighboring nodes in the graph. It is worth mentioning that the number of nodes in a cloud storage system with the same color is likely much larger than the number of chunks of a particular data. The security of data is therefore guaranteed since malicious users cannot guess the locations of other data chunks even if a successful intrusion happened on any node.

It is also worth mentioning that since data is divided into multiple chunks, there should be a node that keeps track the list of nodes, which store the chunks of a particular data, for data retrieval upon request. Obviously, the access point where users submit their storage and retrieval requests can maintain such database. However, since we consider the realistic scenario where every storage node in the system can be the access point, they also suffer the risk of attacks and the meta-data of a particular data may be revealed. To overcome this issue, cloud providers can deploy an additional server that is totally isolated from storage nodes and it is accessible only by network administrator and administration software. This server maintains the database of meta-data of all data stored in the system such that upon a retrieval request, the corresponding access point will contact the server via a secure protocol for requesting the list of storage nodes of the requested data.

The problem is more complex when the required security level is higher than the default level, i.e., the distance between two different chunks of a data is required to be larger than 2. To solve this problem, we propose an algorithm to convert the problem with $K > 2$ to the problem with $K = 2$. Given a value of K and based on the original network topology, the algorithm connects all the pair of nodes whose distance is less than K, resulting in a virtual network topology. Fig. 4.1 shows an example of the conversion process when K is set to 3. Fig. 4.1a shows the original graph of the system. Fig. 4.1b shows the result of the graph conversion algorithm where the added edges are marked as dashed lines. Fig. 4.1c shows a coloring solution obtained on the generated graph. Given the coloring solution, we obtain a set of feasible storage node solutions. Based on the example shown in Fig. 4.1c, we can see that a data can be divided into 2 chunks and stored on feasible pair of nodes such as node 1 and node 6 (red color), node 2 and node 7 (blue color), node 3 and node 4 (green color) or node 5 and node 9 (purple color). The pseudo code of the graph conversion is presented in Algorithm 4. Since there exists in the literature

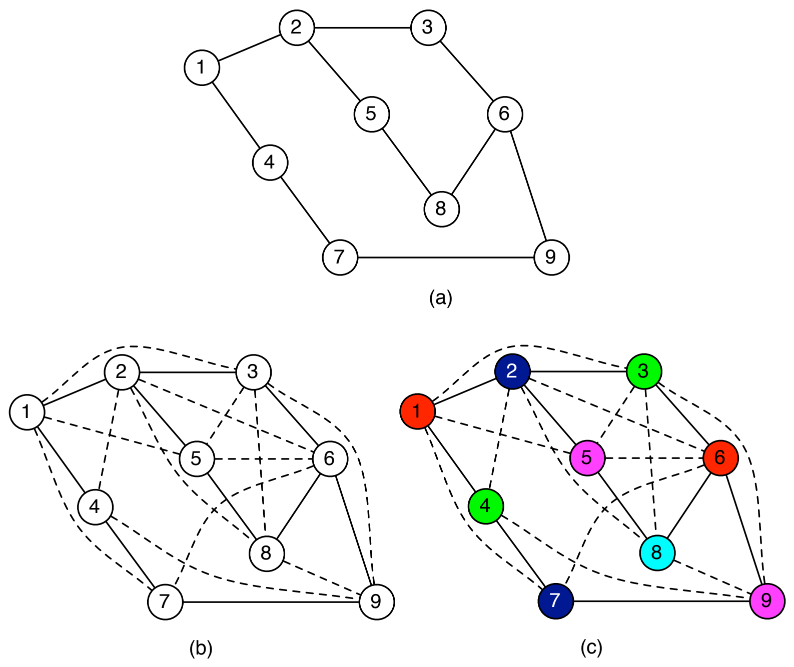


Figure 4.1: Conversion example with the required security level, $K = 3$: (a) Original graph. (b) Result of the conversion where added edges are the dashed lines. (c) A possible coloring solution.

the algorithm for coloring a graph, we omit the presentation of such an algorithm in this paper. We refer the readers to [Pardalos et al., 1999] for further details of the coloring algorithms. In Section 4.3.3, we integrate the conversion algorithm into A-SEDuLOUS as an initial step before determining the best placement solution that minimizes the data retrieval time.

4.3.2 Link-node-disjoint Path for Data Availability

We now present our approach to guarantee the data availability in the event of attacks that happen on any node or network link in the system. Many works in the literature proposed to replicate multiple copies of a data such that if one copy of the data is not available, the user can request from the backup copy. We argue that replication without considering the network topology may not guarantee the data availability due to attacks on nodes and network links in the system. Fig. 4.2 illustrates the replication scenarios where the data availability may or may not be guaranteed. The replication shown in Fig. 4.2a can guarantee the data availability regardless of the security attacks that happen on any node or link of the system assuming that there is a single attack at a time. The replication shown in Fig. 4.2b cannot guarantee the data availability if attacks happen on node 6 or the link connecting node 6 and node 9.

At a very fine-grained level, one may suggest to apply an advanced network recovery mechanism at the network layer to address the attacks on the nodes and links on the path to the access point such as the local rerouting approach [Murali Mohan et al., 2015]. Such an advanced mechanism allows the nodes on the path to determine an alternate path to forward the data to the access point if the next-hop node or link is attacked. However, this mechanism requires that between any pair of storage nodes there must exist at least two paths connecting them. The network state of the entire system must also be available at every node. Furthermore, it also requires the computing capacity of storage nodes to be able to compute

Algorithm 4 Graph Conversion Procedure

```
Input: $G = (V, E)$, $K$.
Output: $G' = (V, E')$ /*Graph with new added edges*/
1: for $i \in V$ do
2:     for $j \in V$ do
3:         if $(i \neq j) \wedge (\mathcal{D}(i, j) < K)$ then
4:             Add an edge to $E'$ to connect $i$ and $j$;
5:         end if
6:     end for
7: end for
8: Color graph $G'$ to obtain set of feasible nodes;
9: return $G'$;
```

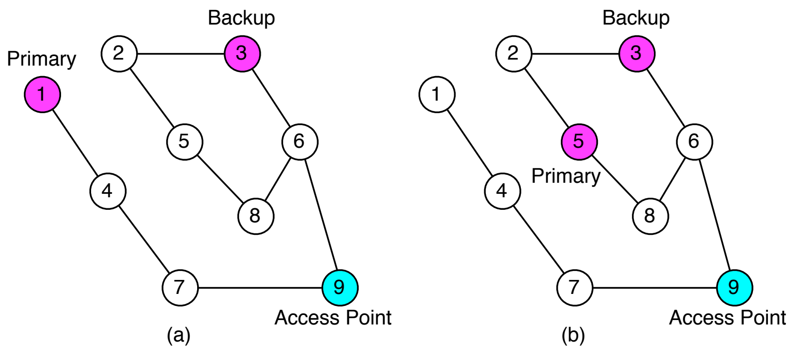


Figure 4.2: Replication example: (a) Replication guarantees the data availability. (b) Replication fails to guarantee the data availability due to security attacks.

the alternate path to the access point. All these assumptions may not be realistic and the approach adds an additional overhead, i.e., path computation delay, on the storage nodes.

Our proposed approach is much simpler and does not incur any additional overhead at the network layer to recover from the failures caused by attacks. For every data chunk, a pair of primary and backup nodes are determined such that the paths from the primary node and backup node to the access point must be link-node-disjoint. For instance, in Fig. 4.2a, the path from primary node, node 1, to the access point is $1 \to 4 \to 7 \to 9$. This path is link-node-disjoint with the path from backup node, node 3, to the access point that is $3 \to 6 \to 9$. Upon a retrieval request, the access point first requests the primary node to retrieve the chunk. In the event of attack on the primary node, or any node or link on the path from the primary node to the access point, the access point will switch to request the backup node after a timeout without the response from the primary node. It is to be noted that while the path from the primary node to the access point should be the shortest path, to ensure the link-node disjointness of paths, the path from the backup node to the access point may not necessarily be the shortest one. Thus, given a backup node that satisfies the security requirement, our approach determines the k-th shortest path that satisfies the disjointness constraint. If there does not exist any path from the backup node to the access point such that the path is link-node-disjoint with the primary path, the backup node is then eliminated. In the next section, we present the integration of the link-node-disjoint path determination into A-SEDuLOUS for the backup node selection.

Algorithm 5 A-SEDuLOUS

```
Input: $G = (V, E), K, p, B_{l_s,l_e}, D, S^{\max}$, and $C_i, i = 1 \dots M$.
Output: $\mathbb{S}^* = (S_1^*, S_2^*, \dots, S_M^*)$ and $I' = (I_1', I_2', \dots, I_H')$.
 1: Run Algorithm 4 for graph conversion and coloring;
 2: Get the set of colors, denoted as $\mathbb{C}$;
 3: Run Algorithm 6 to select primary nodes, sizes of primary chunks and the color of the selected primary nodes;
 4: if $\mathbb{S}^* \neq \emptyset$ then
 5:     Run Algorithm 7 to select backup nodes;
 6: end if
 7: if $\mathbb{S}^* \neq \emptyset \wedge I' \neq \emptyset$ then
 8:     return $\mathbb{S}^*$ and $I'$;
 9: else
10:     return $\emptyset$ and $\emptyset$; /*the request is rejected*/
11: end if
```

4.3.3 A-SEDuLOUS

In this section, we present the entire algorithm of A-SEDuLOUS to obtain the data placement solution for a storage request such that the security and availability requirements are satisfied. The pseudo code of the algorithm is presented in Algorithm 5, which consists of three parts. In the first part, it runs Algorithm 4 to realize the graph conversion and coloring. Based on the coloring solution, the second part realizes a greedy algorithm to select a list of primary nodes and to determine the size of data chunks stored on the selected nodes. If there exists a solution for storing primary chunks of the data, A-SEDuLOUS runs the third part to determine the list of backup nodes. If there is no either the solution for the primary or backup nodes, the request is then rejected. We present below the detailed description of the algorithms for the second and the third part of A-SEDuLOUS.

4.3.3.1 Primary Chunk Determination and Placement

Given a coloring solution, the algorithm, presented in Algorithm 6, starts by counting the number of colors used to color the graph. The set of colors used for coloring the graph, denoted as $\mathbb{C}$, is then used in the outer *for* loop (see lines 3–22) to determine the best placement solution. Precisely, for every color c in set $\mathbb{C}$, the algorithm first sorts all the nodes, which are colored by c, in the ascending order of the transmission time of a unit data to the access point. Given the ordered set of storage nodes, denoted as $V_c^{'}$, the algorithm then assigns a data chunk from the first node with the smallest transmission time. For every node v in the ordered set, $V_c^{'}$, the size of the stored data chunk is determined by $\min\{S^{\max}, C_v, D^{\text{temp}}\}$. This means that if the size of the remaining data is large, only a chunk with a maximum size $S^{\max}$ can be assigned. However, if the available storage space of storage node v is not sufficient, only a smaller chunk can occupy the remaining available storage space. This step is represented in the inner *for* loop (see lines 7–13). When the entire data has been accommodated, the algorithm breaks out the *for* loop even though there still exist feasible storage nodes since these nodes are slower in data transmission.

Algorithm 6 Primary Node Selection and Size Determination of the Primary Chunks

Input: $G = (V, E), K, p, B_{l_s,l_e}, D, S^{\max}, C_i, i = 1 \ldots M$, and $\mathbb{C}$.

Output: $\mathbb{S}^* = (S_1^*, S_2^*, \ldots, S_M^*)$ and c^*, color of primary nodes.

```
1: $T^* \leftarrow \infty$;
2: $c^* \leftarrow \infty$;
3: for $c \in \mathbb{C}$ do
4:     Pick all the nodes that have color $c$ and put them to set $V_c$;
5:     Sort the nodes in $V_c$ in the ascending order of retrieval time to the access point. The result is
       stored in set $V_c^{'}$;
6:     $D^{\text{temp}} \leftarrow D$;
7:     for $v \in V_c^{'}$ do
8:         Assign a chunk, $S_v = \min\{S^{\max}, C_v, D^{\text{temp}}\}$;
9:         $D^{\text{temp}} \leftarrow D^{\text{temp}} - \min\{S^{\max}, C_v, D^{\text{temp}}\}$;
10:        if $D^{\text{temp}} = 0$ then
11:            break;
12:        end if
13:    end for
14:    if $D^{\text{temp}} = 0$ then
15:        Compute the retrieval time, $T^{\text{temp}}$, as defined in Eq. (4.2);
16:        if $T^{\text{temp}} < T^*$ then
17:            $T^* \leftarrow T^{\text{temp}}$;
18:            $\mathbb{S}^* \leftarrow \{S_v, v \in V_c^{'}\}$;
19:            $c^* \leftarrow c$;
20:        end if
21:    end if
22: end for
23: if $T^* \neq \infty$ then
24:    return $\mathbb{S}^*$ and $c^*$;
25: else
26:    return $\emptyset$ and $\infty$; /*the request is rejected*/
27: end if
```

It may also happen that there are not sufficient nodes to store a data, i.e., the *for* loop already finishes all feasible nodes with the same color but the data still remains. This assignment solution is infeasible since the data cannot be accommodated in the system. Thus, only when the entire data is accommodated, the algorithm then computes the total retrieval time of the placement solution as shown in line 15. If the total retrieval time of the current solution is smaller than the previous best solution, the algorithm then updates the best solution for future comparison, using the nodes with different color.

The algorithm stops when it has completely verified all the feasible solutions represented by the set of colors used for graph coloring. The final solution is the best data placement solution, which not only satisfies the security requirement but also minimizes the total retrieval time of the data. It may also happen that there does not exist a feasible solution if the size of data is too large while all the storage nodes have been saturated or the number of nodes with the same color is too small. In such a case, the storage request is then rejected, i.e., an empty set is returned as shown in line 26. Practically, if the rejection ratio is too high, providers may scale out the system by increasing the number of nodes and their capacities. On the user side, they can also increase the maximum size of a chunk to reduce the number of storage nodes required for storing a data while still satisfying the security requirement.

4.3.3.2 Backup Node Selection for Data Replication

In this section, we present the algorithm for backup node determination. The pseudo code of the algorithm is shown in Algorithm 7. Given the decision of node selection and chunk size determination of primary chunks, obtained by Algorithm 6, we denote $I = (I_h | h = 1, \ldots, H, S^*_{I_h} \neq 0)$ as the vector of indices of the primary nodes. As shown in the main *while* loop of Algorithm 7, it will repeat the process for determining a backup node for each primary chunk.

For each iteration, the algorithm first selects the largest primary chunk that has not been replicated (see line 3). We give the priority for the largest chunk to avoid unnecessary runs for smaller chunks before rejecting the request due to the shortage of storage space or the availability requirement. Given that $S^*_{I_h}$ is the size of the primary chunk stored in primary node I_h, the algorithm then goes through all feasible nodes that have sufficient capacity, satisfy the security constraint, and they do not store any chunk of the data. The list of feasible nodes is determined by line 4. For each feasible node v, the algorithm determines a k-th shortest path that is link-node-disjoint with the primary path from node I_h to the access point p as shown in line 7. If such path exists, then node v is considered as a candidate for replicating chunk $S^*_{I_h}$. The algorithm computes the retrieval time of chunk $S^*_{I_h}$ from node v to the access point to compare against that of the ever-best node that has been checked. After exiting the *for* loop (lines 6–14), if node v^* is the best node in terms of retrieval time, chunk $S^*_{I_h}$ is assigned to node v^* for being replicated. If there does not exist any candidate for chunk $S^*_{I_h}$, the algorithm will stop the *while* loop and the request will be rejected since the system cannot satisfy the availability requirement. If all the primary chunks have been replicated, the algorithm will stop and return the list of backup nodes.

4.3.4 Complexity of A-SEDuLOUS

The graph conversion algorithm consists of two *for* loops. Thus, it has a complexity of $\mathcal{O}(|V|^2)$ where $|V|$ is the number of nodes in the graph. However, for every pair of nodes, the procedure invokes an algorithm for determining the shortest path between them. The well-known algorithm for finding the shortest path between two nodes is the Dijkstra algorithm, which has a complexity of $\mathcal{O}(|E| + |V| \log |V|)$ where $|E|$ is the number of edges in the graph. It is noted that the graph conversion also performs the graph coloring. Since we do not focus on finding the minimum number of colors used for the graph, a greedy algorithm with a complexity of $\mathcal{O}(|V| + |E|)$ can produce a coloring solution, which is sufficient to guarantee the security level in the data placement problem. Thus, the overall complexity of the graph conversion procedure is $\mathcal{O}(|V|^2(|E| + |V| \log |V|))$.

To determine the primary nodes and size of primary chunks, Algorithm 6 will run two nested *for* loops as shown in line 3 and line 7. To complete these *for* loops, the worst case will have a complexity

Algorithm 7 Backup Node Selection for Data Replication

```
Input: $G = (V, E), p, B_{l_s,l_e}, (C_1, \ldots, C_M), \mathbb{S}^* = (S^*_1, \ldots, S^*_M)$, and $c^*$, the color of primary nodes.
Output: $I' = (I'_1, I'_2, \ldots, I'_H)$. /*index of backup nodes*/
 1: $I \leftarrow (I_h|h = 1, \ldots, H, S^*_{I_h} \neq 0)$; $stop \leftarrow 0$; $success \leftarrow 1$;
 2: while $stop = 0$ do
 3:     $\{S^*_{I_h}, I_h\} \leftarrow \max\{S^*_{I_h}|S^*_{I_h}$ is not replicated$\}$;
 4:     Pick all the nodes that have color $c^*$ and sufficient storage capacity, but do not store any primary
        or backup chunk of the data. Add them to set $V_{c^*}$;
 5:     $T^{\text{bak}} \leftarrow \infty$; $v^* \leftarrow \infty$;
 6:     for $v \in V_{c^*}$ do
 7:         Get $P_{v,p}$ the $k$-th shortest path that is link-node-disjoint with the primary path $P_{I_h,p}$;
 8:         if $P_{v,p}$ exists then
 9:             Compute the retrieval time, $T^{\text{temp}}$, as defined in Eq. (4.1);
10:             if $T^{\text{temp}} < T^{\text{bak}}$ then
11:                 $T^{\text{bak}} \leftarrow T^{\text{temp}}$; $v^* \leftarrow v$;
12:             end if
13:         end if
14:     end for
15:     if $v^* \neq \infty$ then
16:         Assign chunk $S^*_{I_h}$ to node $v^*$ for replication;
17:         Add node $v^*$ to $I'$;
18:         Mark chunk $S^*_{I_h}$ as replicated;
19:     else
20:         $stop \leftarrow 1$; $success \leftarrow 0$;
21:     end if
22:     if All primary chunks have been replicated then
23:         $stop \leftarrow 1$;
24:     end if
25: end while
26: if $success = 1$ then
27:     return $I'$; /*the list of backup nodes*/
28: else
29:     return $\emptyset$; /*the request is rejected*/
30: end if
```

of $\mathcal{O}(|V|^2)$, i.e., every node of the graph has a different color value. In Algorithm 7, the worst case has the complexity of $\mathcal{O}(|V|^2(|E|+|V|\log|V|))$ since the algorithm has to check every pair of primary and backup nodes and determine the shortest path from the backup node to the access point. As mentioned earlier, determining the shortest path has the complexity of $\mathcal{O}(|E| + |V|\log|V|)$. Since A-SEDuLOUS runs Algorithm 4, Algorithm 6 and Algorithm 7 sequentially, the complexity of A-SEDuLOUS is therefore $\mathcal{O}(|V|^2(|E|+|V|\log|V|))$. We believe that this polynomial complexity is affordable for both cloud providers and users with respect to the gain that users can obtain in data availability and security. Indeed, even for a large system, such as the Internet2 topology that spreads over the US, the total number of nodes in the system is in the order of 50 nodes. This ensures that the algorithm will produce the data placement solutions in the order of a second. The low complexity of the proposed algorithm will also guarantee that it will finish the running in a polynomial time with the size of the system is extremely large with more than 1000 nodes.

4.4 Performance Study

In this section, we present the performance study to demonstrate the effectiveness of the proposed data placement algorithm. We first describe the simulation setting and then present the analysis of results.

4.4.1 Simulation Setting

The proposed algorithm was tested on the cloud storage systems with random network topologies and the Internet2 topology [Internet2, 2016]. We run the simulations with different scenarios in which the security requirement has to be strictly satisfied whereas the availability requirement is optional. We use the three following performance metrics to evaluate the proposed algorithm:

- Average retrieval time: the total transmission time of all the chunks of a data to the access point, taking average value of all the accepted requests;
- Number of rejections among storage requests; and
- Average distance, i.e., the number of hops, from the nodes storing data chunks to the access point.

We compare the performance of A-SEDuLOUS against that of the two following baseline algorithms:

- Random Selection of Nodes (RSN) randomly selects a storage node among the nodes that satisfy the security and availability requirements to store a data chunk whose the size is decided based on the available storage space of the node, maximum chunk size and the remaining size of the data. The procedure repeats until the entire data is accommodated or no more storage nodes are available. Otherwise, the request is rejected;
- Furthest Node First (FNF) selects a node that has the furthest distance to the nodes that have been selected to store the chunks of the same data. It is also an iterative algorithm that has the same stopping condition as that of RSN algorithm.

The storage requests are generated such that the size of data in each request is randomly chosen from the range of $[500, 1500]$ MB. We vary the number of requests from 1000 to 10000 to evaluate the performance of the proposed algorithm with different level of loads. The access point of each request is also chosen randomly among storage nodes.

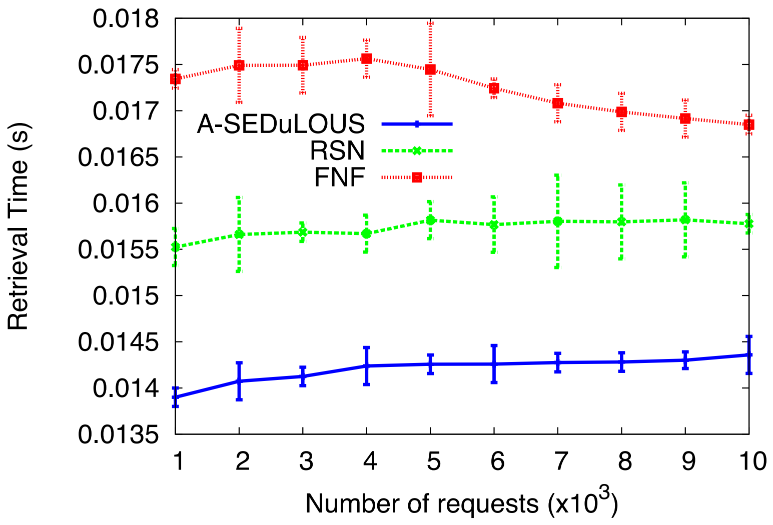


Figure 4.3: Performance of algorithms.

4.4.2 Performance without Availability Requirement

In this section, we analyze the performance of the proposed algorithm without requirement of data availability, i.e., only primary chunks are stored. We run the simulations on the systems with random network topologies and the Internet2 topology.

4.4.2.1 Performance with Random Network Topologies

We randomly generate network topologies with minimum degree of nodes is set to 2. The capacity of storage nodes is chosen randomly from $[500, 1000]$ GB. The bandwidth of the links that connect the storage nodes is chosen randomly from $[100, 500]$ Mbps. Without explicitly indicating, the security level is set to 3, i.e., the distance between the nodes storing any pair of chunks of the same data regardless of primary or backup chunks should be at least 3 hops.

Overall Performance of A-SEDuLOUS: In this simulation, the system consists of 30 storage nodes. In Fig. 4.3, we present the retrieval time resulted by the algorithms with respect to the number of requests. The results show that A-SEDuLOUS has the best performance compared to other algorithms. In the best case, A-SEDuLOUS reduces the retrieval time by up to 20%. The results also show that when the number of requests increases, the retrieval time with A-SEDuLOUS slightly increases. It is due to the fact that A-SEDuLOUS exploits the nearest nodes that satisfy the security constraint when the number of arriving requests is low. At higher number of requests, all the nearest storage nodes are saturated, further nodes are then utilized, thereby increasing the retrieval time.

It is also observed that not only having the best performance in terms of retrieval time, A-SEDuLOUS also has the best performance in terms of total volume of data admitted for storage. We present the total volume of data admitted for storage in Fig. 4.4 and the number of rejections in Fig. 4.5, respectively. This implies that A-SEDuLOUS does not sacrifice other performance metrics to achieve the data security whereas other algorithms incur a trade-off between retrieval time and security.

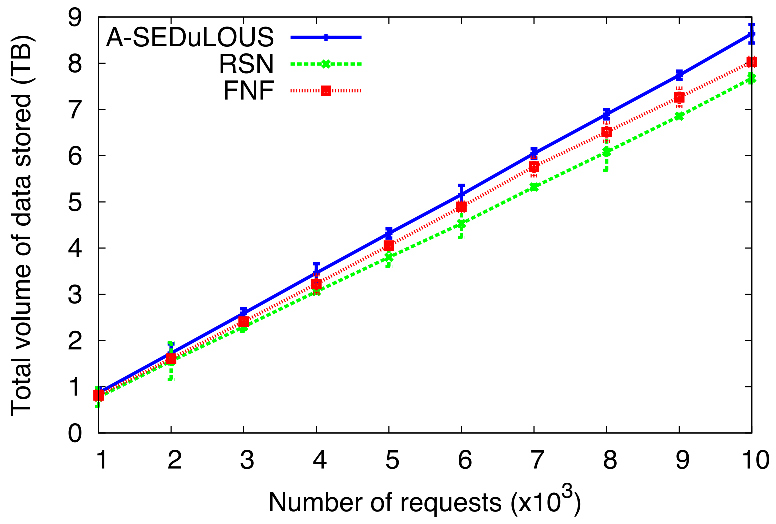


Figure 4.4: Total data volume admitted to be stored.

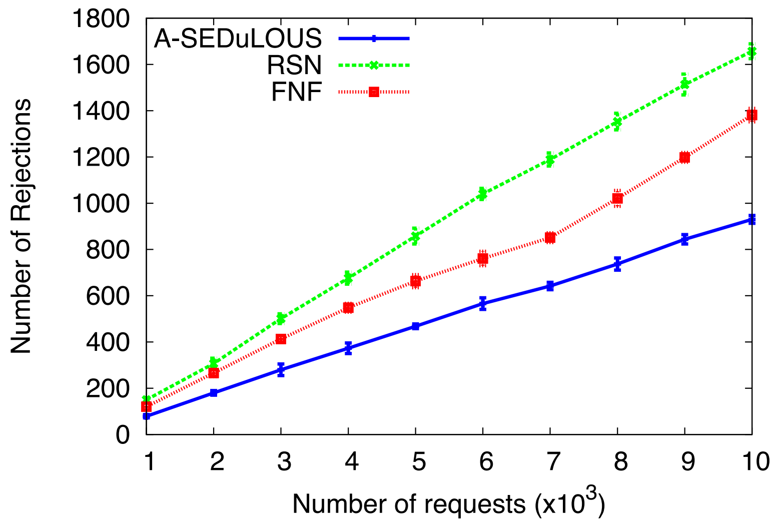


Figure 4.5: Number of rejections.

Impact of the number of storage nodes: We evaluate the performance of the proposed algorithm with different network topologies by varying the number of storage nodes from 25 to 50. Fig. 4.6 presents the retrieval time of the algorithms with respect to the number of storage nodes. The results show that A-SEDuLOUS always has the best performance. The fluctuation in retrieval time resulted by each algorithm is because of the heterogeneity of the link capacity. In Fig. 4.7, we present the average distance from the nodes storing data chunks to the access point. The results show that this distance gradually increases with the increase in the number of storage nodes. FNF has the highest increase since it always chooses the furthest node for storing data chunks. Nevertheless, A-SEDuLOUS always outperforms other algorithms while guaranteeing the data security.

Impact of security level: In this simulation, we evaluate the impact of the security level, i.e., the minimum distance between two nodes that store two different chunks of the same data. We vary the security level from 2 to 5 and evaluate the performance of the algorithms. In Fig. 4.8, we present the

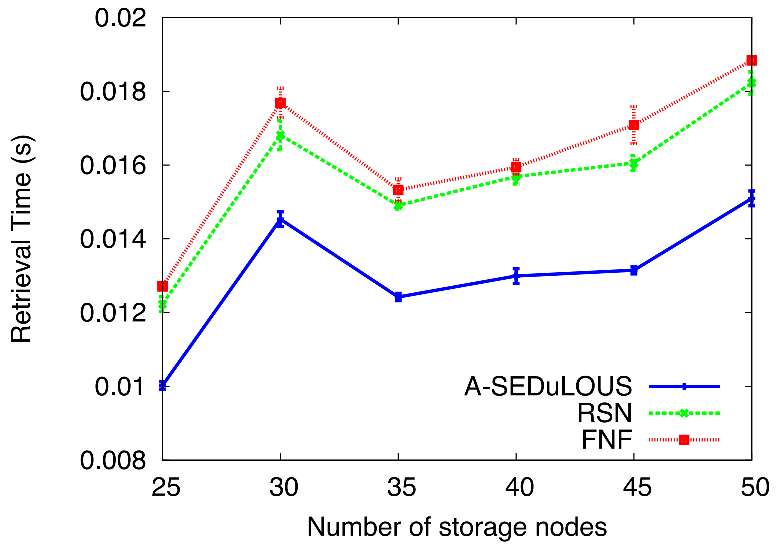


Figure 4.6: Performance of algorithms with respect to number of storage nodes.

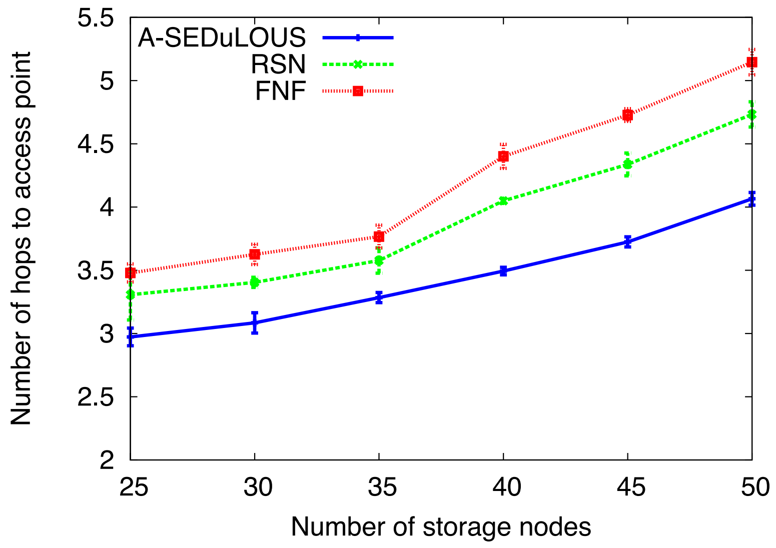


Figure 4.7: Number of hops to access point from chunks.

retrieval time with respect to the security level. The results show that the retrieval time increases when the security level increases. It is because the storage nodes, which are close to the access point, violate the security constraint. This is shown clearer in Fig. 4.9 with the average number of hops from the nodes storing data chunks to the access point. This distance increases with the increase in the security level. In all scenarios, A-SEDuLOUS always has the best performance compared to other algorithms. It minimizes the retrieval time while satisfying the security requirement. We also observe an interesting behavior that the performance of A-SEDuLOUS is significantly impacted by the security level whereas FNF and RSN are not. It is because FNF always chooses the furthest nodes for storing data while RSN randomly selects the feasible nodes.

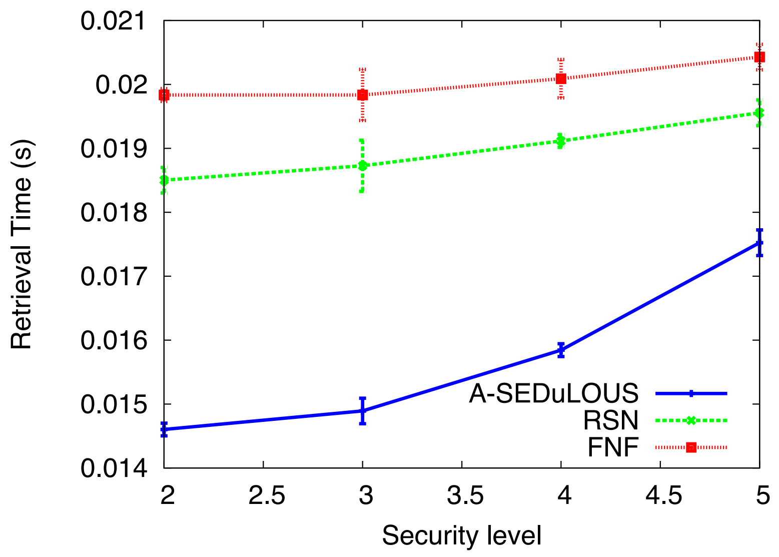


Figure 4.8: Retrieval time with respect to security level.

4.4.2.2 Performance with Internet2 Topology

We also evaluate the performance of the proposed algorithm with a realistic network topology, the Internet2 topology with 36 nodes in US. We set the capacity of all the network links connecting the storage nodes to 1 Gbps. The capacity of storage nodes is chosen randomly from $[500, 1000]$ GB. We obtain the same performance behavior as in the case with random topologies. In Fig. 4.10, we present the retrieval time of the algorithms with respect to the number of requests. The results show that the retrieval time is quite stable regardless of the number of requests is small (1000 requests) or high (10000 requests). This is because the storage nodes in the Internet2 topology are well distributed with small node degree that takes a value among 2 and 3. The results also show that A-SEDuLOUS has the best performance by reducing the retrieval time by up to 19% and 14% compared to FNF and RSN, respectively.

With the Internet2 topology, A-SEDuLOUS also shows the best performance in terms of the number of rejections. As shown in Fig. 4.11, even at high number of requests, A-SEDuLOUS does not reject any request while FNF and RSN reject 400 and 250 requests, respectively. We also evaluate the impact of the security level with the Internet2 topology. The results in Fig. 4.12 show the same behavior as that with random topologies. The retrieval time increases with respect to the security level. In all scenarios, A-SEDuLOUS always shows its effectiveness with a performance gain by up to 19% in reduction of retrieval time compared to other algorithms.

4.4.3 Performance with Availability Requirement

In this section, we analyze the performance of the proposed algorithm with the availability requirement. We use random network graphs with 70 nodes to ensure that there are sufficient nodes for both primary and backup nodes. Every data chunk is replicated on a backup node, guaranteeing that all primary and backup chunks satisfy not only the security requirement but also the availability requirement. We consider the two following scenarios:

- The first scenario assumes that no attack happens in the system. Thus, the data retrieval time is

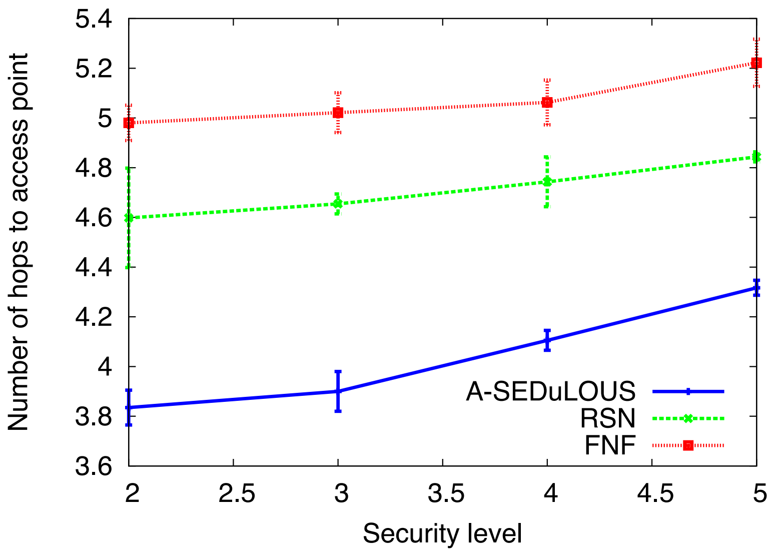


Figure 4.9: Number of hops to access point from chunks.

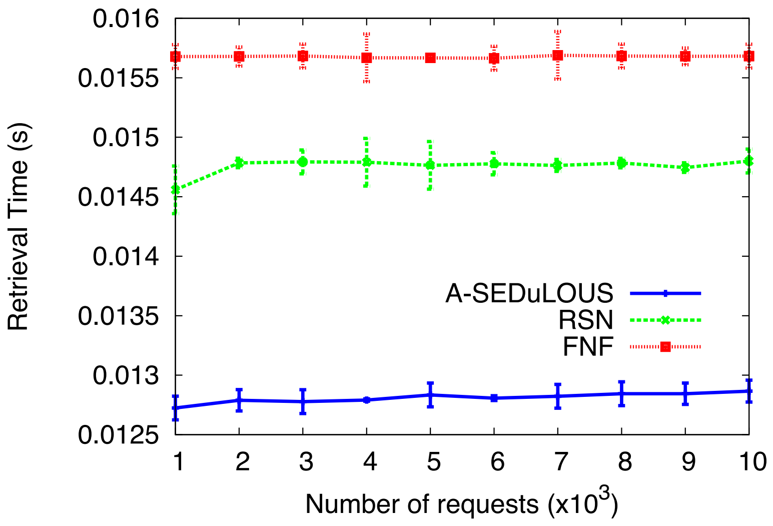


Figure 4.10: Performance of A-SEDuLOUS and other algorithms with Internet2 topology.

computed by using primary chunks; and

- The second scenario assumes that there is an attack that happens on a random node or link in the system. All the primary chunks stored on the nodes, which use the attacked link or node in the primary path, will not be available. Thus, the data retrieval time is computed by using the backup chunks to substitute the unavailable primary chunks.

4.4.3.1 Performance with No Attack

In Fig. 4.13, we present the average retrieval time with respect to the number of requests arriving to the system. We obtain the same behavior as that in the case without the data availability requirement. A-SEDuLOUS always has the best performance compared to other baseline algorithms by reducing the retrieval time by up to 20% compared to RSN and 34% compared to FNF, respectively. This improve-

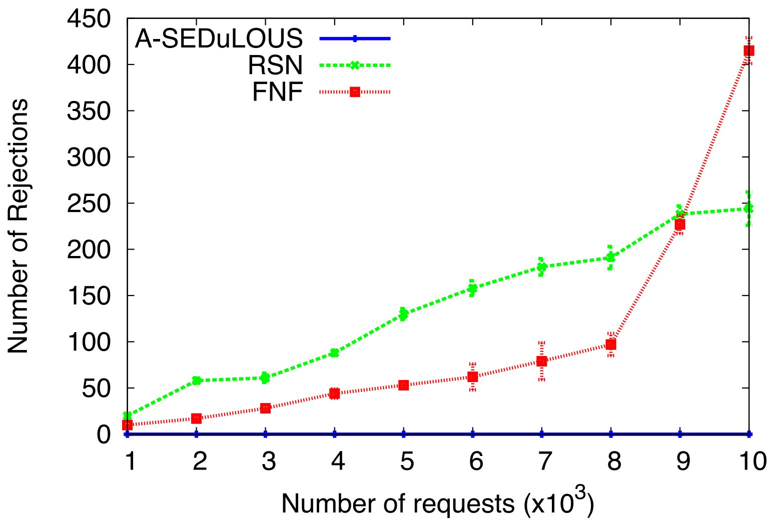


Figure 4.11: Number of rejections with Internet2 topology.

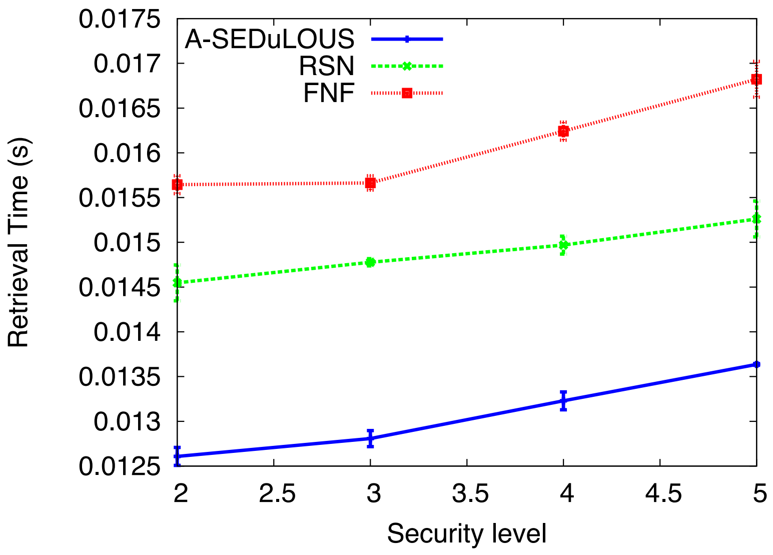


Figure 4.12: Retrieval time with respect to the security level.

ment is much better than that of A-SEDuLOUS in the case without the availability requirement. This shows that with more constraints in the selection of nodes, A-SEDuLOUS has better improvement. A-SEDuLOUS also has the best resource efficiency by rejecting the least number of requests as shown in Fig. 4.14, leading to higher benefit for cloud providers when they charge users for resource usage. It will also improve the cloud resource utilization since more requests are accommodated.

We present the performance of A-SEDuLOUS with and without the availability requirement in Fig. 4.15, denoted as A-SEDuOUS-WAR and A-SEDuLOUS-WoAR. It is to be noted that though we consider the availability requirement, we assume that there is no attack in this simulation. Thus, the retrieval time of a data is computed as the total retrieval time of the primary chunks even though backup chunks are also available. It is observed that when considering the data availability requirement, the average retrieval time is longer and it increases with respect to the increase in the number of arriving requests. Indeed, without considering the availability requirement, the node with the shortest retrieval

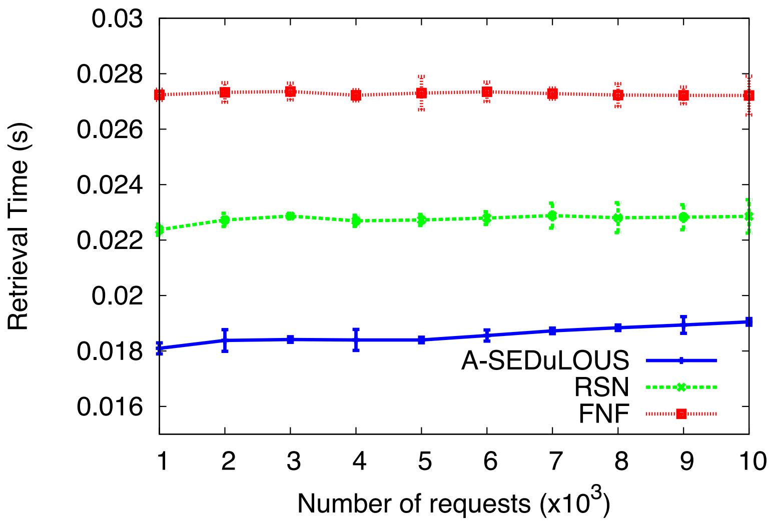


Figure 4.13: Performance of A-SEDuLOUS (vs) other algorithms with data availability requirement.

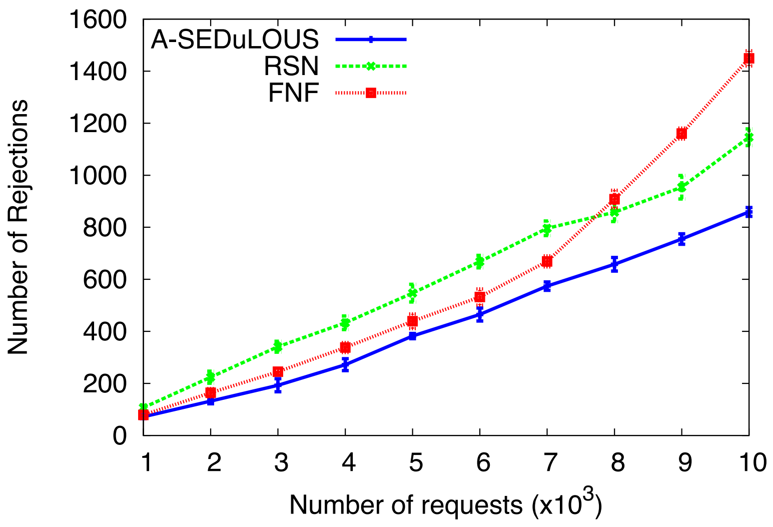


Figure 4.14: Number of rejections with data availability requirement.

time will be selected for storing a chunk. However, when considering the data availability requirement, this node may not be selected as a primary node because there does not exist another node, which will be used as backup node that satisfies both security and availability requirements, i.e., a node that is far away at least K hops according to the security requirement from all the primary nodes as well as other backup nodes, and it has a link-node-disjoint path to the access point. While the data retrieval time increases by up to 7%, we believe that the gain of data availability in case of attacks is worth to compensate such longer retrieval time.

4.4.3.2 Performance with Attack

In this simulation, we assume that there is an attack happening on a node or link that is used in the paths from primary nodes to the access point. We compute the total retrieval time of a data using the backup chunks for the unavailable primary chunks as mentioned earlier. Since we consider a single

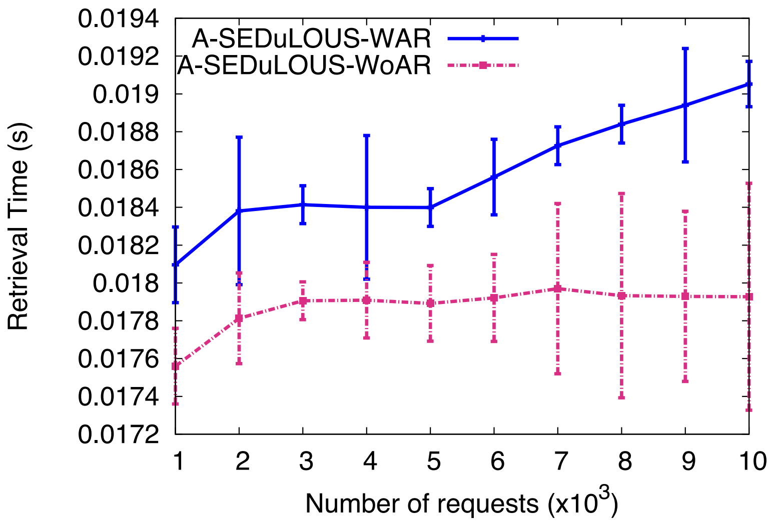


Figure 4.15: Performance of A-SEDuLOUS with and without availability requirement and no attacks.

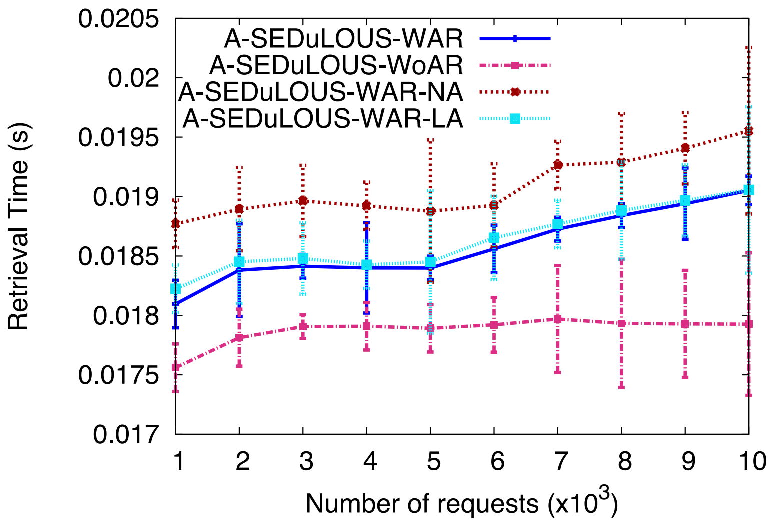


Figure 4.16: Performance of A-SEDuLOUS with and without availability requirement in case of one attack that happens on a random node or link.

attack scenario, we run two simulations. The first one is for the link attack scenario, making the attacked link will be disconnected. The second one is node attack scenario where the attacked node is unavailable for data retrieval or forwarding in case it is an intermediate node on the path.

In Fig. 4.16, we present the average retrieval time of A-SEDuLOUS considering different scenario of attacks. We denote A-SEDuLOUS-WAR-NA and A-SEDuLOUS-WAR-LA for the performance of A-SEDuLOUS with the availability requirement in case of node attack and link attack, respectively. It is obvious that the retrieval time increases when an attack happens on any node or link used in the primary paths, since the data retrieval has to use backup path than is longer that the shortest path used for the primary chunks. However, while the impact of link attack is not very significant, i.e., increasing the data retrieval time by up to 1% compared to the case without attacks, the impact of node attack is much significant with an increase in the retrieval time by up to 3%. Compared to the performance of A-SEDuLOUS in case without the availability requirement, the retrieval time when considering node attack increases by up to 9%. From a user's point of view, we believe that the increase in retrieval time is completely affordable when considering the gain of data availability such that users can always access their data regardless of any attack in the system. This simulation results also show that a node attack is more severe than a link attack since it is used not only for storing primary chunks but also for being an intermediate node on a primary path.

4.5 Chapter Summary

In this chapter, we addressed the data placement problem in cloud storage systems considering the security and availability requirements. We formulated the problem as an optimization programming model that minimizes the total retrieval time of a data, which is divisible into multiple chunks with arbitrary size. Given a primary placement solution, we developed a backup placement model such that every primary chunk is replicated and stored in a backup node, which satisfies the requirement that even if an attack happens on any link or node in the system, the data is always accessible. Since the optimization problem is computationally prohibitive, we developed an efficient heuristic algorithm namely Availability and Security-awarE Data placement algorithm for cLOUd storage Systems (A-SEDuLOUS) to solve the data placement problem [Kang et al., 2016b, Kang et al., 2016d]. We integrated the T-coloring approach into A-SEDuLOUS to solve the security issue. The validation was performed through extensive simulations on the cloud storage systems with random network topologies as well as the realistic Internet2 topology. The simulation results demonstrate the effectiveness of the proposed algorithm by reducing the retrieval time by up to 34% for random topologies and 19% for the Internet2 topology. The simulation results also show that while achieving the best performance and guaranteeing the data privacy and availability, the proposed algorithm does not sacrifice other performance metrics such as the rejection ratio and retrieval time, thus leading to a higher economic profit for commercial cloud providers. While this work applies graph theory for addressing the data privacy and availability issues, in the next chapter, we explore another approach for protecting the data privacy. We study to use the cryptography approach to encrypt the data before being stored in public clouds.

Chapter 5

ESPRESSO: An Encryption as a Service for Cloud Storage Systems

In this chapter, we continue studying the problem of protecting data privacy in cloud storage systems. While the previous chapter applies graph theory to protect data privacy, this chapter investigates the traditional method based on encryption techniques. We design and implement an encryption service namely ESPRESSO (Encryption as a Service for Cloud Storage Systems). The flexible design and the standalone property of the proposed encryption service allow cloud providers to easily integrate it into their infrastructures without heavy modification and implementation. We integrated ESPRESSO into two open-source cloud storage platforms: OpenStack/Swift and Nimbus/Cumulus and carry out the experiments to evaluate the performance and demonstrate the effectiveness of the proposed approach.

5.1 Research Motivation and Objectives

Many cloud storage systems are providing cloud users high data availability and the flexibility in data management, and they become the primary storage space for users' data. Thus, instead of storing and managing data in local servers, most of users nowadays are moving their data into the cloud and paying for storage and management service by a pay-per-use model. In this sense, cloud users are sharing a common storage space offered by Cloud Service Providers (CSPs). This characteristic raises several challenges, which are hindering the migration of users' softwares and data into the cloud [IMEX, 2010]. Among them, the security and data privacy are the most important challenges needed to be solved to relieve the users' concerns [Tian et al., 2010]. While consumers have been willing to trade privacy for the convenience of cloud storage services, this is not the case for enterprises and government organizations. This reluctance can be attributed to several factors that range from a desire to protect mission-critical data to regulatory obligations to preserve the confidentiality and integrity of data. The latter can occur when the customer is responsible for keeping personally identifiable information (PII), or financial and medical records [Kamara and Lauter, 2010]. Driven by the need to secure growing cloud data storage systems as well as high profile security breaches, data protection in cloud storage systems has become a hot topic in both academia and industry [Factor et al., 2013, Huang et al., 2011b, Hao and Han, 2011, Itani et al., 2009]. While the current approaches rely on a user-centric authentication service such as login/password or Two Factor Authentication (2FA), which can be broken by authentication attacks, encryption emerged as one of the most effective means to protect sensitive data no matter where it lives [Harrin, 2012].

With an encryption tool, users can encrypt data on their local machine before uploading the encrypted data to a cloud. However, this approach introduces an additional burden for users to manage the encryption key and operate the encryption tool. Furthermore, users are required to equip local machines which are able to handle such a compute-intensive task that incurs a delay and complicates the data management on the user side. These issues make the user-side encryption approach difficult to realize, especially when users are using scarce resource devices such as smartphones or mobile devices. A server-side encryption approach is therefore needed. On one hand, CSPs can provide the encryption to users as an added value service with minimum additional cost. On the other hand, this encryption can be offered as a free charge service. It then becomes a competitive advantage of a CSP against other CSPs to attract users and increase the CSP's reputation.

Among existing CSPs, only two commercial CSPs: Google Cloud Storage [Google, 2014] and Amazon S3 [Amazon S3, 2014] offer a server-side encryption service. However, the encryption services developed by Google and Amazon cannot be adopted by many other CSPs, which want to offer the server-side encryption to users such as Microsoft Azure [Microsoft Azure, 2014], GoGrid [GoGrid, 2014], RackSpace [RackSpace, 2014]. This observation inspires us to design and implement a standalone encryption service, ESPRESSO, for such CSPs to integrate into their infrastructures without heavy modification and implementation. Furthermore, we aim at providing a configurable and flexible encryption service for both CSPs who can choose the encryption algorithm based on their preference, and users who can specify the critical level of their data. The data with higher critical level needs to be more securely protected. Last but not least, we aim at providing ESPRESSO as a transparent encryption service, which makes users perceive no difference between with and without the encryption service in terms of latency and complexity of data management operations.

5.2 Problem Statement

5.2.1 The System and Threat Model

In this chapter, we consider the CSPs that provide a data storage service. To protect the data privacy, an encryption service is used to encrypt the data before being stored in the cloud, and decrypt the data whenever users need to retrieve the data. Depending on the deployment location of the encryption service, different threat models are introduced and analyzed in the following:

1. The first model applies the user-side encryption approach. Users deploy the encryption software on their local machine and flexibly operate the service without needing to trust any third party. However, users are generally not expert in the security domain. The user's machine therefore suf-

fers the security risks such as key exposure attacks or attacks from malicious programs. Moreover, it is not an easy task for non-expert users to take full responsibility of encryption key management such as key generation, key storage and keeping those keys always safe. Yet, if users are using scarce resource devices such as mobile devices, performing the data encryption on such devices may not be possible since the encryption is considered as a compute-intensive task.

2. Users rely on a third party who offers the encryption service. The third party takes full responsibility for managing the data encryption, protecting the encryption server and preventing the exposure of the users' encryption keys. Assuming that the encryption service is resistant to the security risks, users still have the sole concern on the adversarial behavior of the third party. With the curiosity and economic purpose, the third party might collude with malicious users to harvest data contents when it is highly beneficial [Yu et al., 2010]. Moreover, this model requires further effort from users to retrieve the encrypted data from the third party to their local machine before uploading again to cloud storage servers.

3. CSPs play the role of the third party presented in the second model. CSPs deploy the encryption software on a server in its trusted domain as one of its components. Users therefore benefit all advantages but also suffer the security risks as mentioned above. The operation overhead might be lesser since users do not need to manage the encrypted data. Instead, users upload the plaintext data to the CSP who will forward the data to the encryption server to encrypt before storing the encrypted data in storage servers.

As described, each model has advantages and disadvantages. Assuming that users trust the third party in the second model and the CSP in the third model at the same level, we believe that the third model brings users the most advantages. Depending on the model, the encryption service is designed and implemented differently to assure that it efficiently operates at high performance. We present in the next section the design goals of ESPRESSO, the encryption service for CSPs as we advocate the third model presented above.

5.2.2 Design Goals

Several design requirements should be carefully considered since the design directly affects the overall performance of the system.

Architectural requirements: The encryption service should include two main components. The first component is the encryption key management. To increase isolation among users, a CSP may use different keys to encrypt different users' data and a user may have multiple keys for different data. To prevent leaking one's key to another, the encryption key must also be encrypted. Additionally, since the data availability is an important requirement of a cloud storage system, keys need to be replicated to be available when requested. The second component is the data encryption management. ESPRESSO needs to provide the flexibility for both CSPs and users. Since ESPRESSO can be used by different CSPs, it should support multiple encryption algorithms. A CSP may choose its preferred algorithms to process users' data, e.g., Swift may use AES while Eucalyptus may use Blowfish. For users, the service should allow them to specify a desired critical level for their data. Currently, the CSPs, which offer server-side encryption, support only a single key length option, e.g., Google Cloud Storage uses 128-bit keys. However, users may have different levels of security. Financial or medical records need to be more securely protected using a longer key such as 256-bit keys than entertainment data like musics or movies using a shorter key such as 128-bit keys.

Choosing supported encryption algorithms and critical levels of data: Given that CSPs offer different encryption algorithms and key lengths, choosing the supported encryption algorithms and critical levels of data is also important to achieve the flexibility. There exists many encryption algorithms in the literature including symmetric and asymmetric algorithms with their own advantages and disadvantages. A symmetric algorithm eases the implementation, however, it may not provide high level of security while an asymmetric algorithm is more complex to manage its key pair. Additionally, an asymmetric algorithm may take longer time for data encryption and decryption. On the critical level of data, the longer key length is, the higher security level is guaranteed, however, it also takes longer time for encryption and decryption of data. Therefore, choosing the key length for each security level should take into account the tradeoff between the security level and the processing time.

APIs for integration to cloud storage platforms: As a last requirement of ESPRESSO, a well-designed integration API is also important since this allows CSPs to integrate and to use ESPRESSO easily without heavy modification of the architecture and implementation of their infrastructure. For instance, to provide an enhanced encryption service with a flexible critical level, the critical level should be one of API parameters along with data to be stored and user identification. Depending on the design, other parameters could be added. However, they should be carefully chosen since it may be difficult for CSPs to integrate ESPRESSO with redundant parameters.

5.3 System Architecture of ESPRESSO

In this section, we first present the detailed architecture and then describe the method to handle the flexibility and support multi-user scheme in ESPRESSO.

5.3.1 Architecture of ESPRESSO

The overall architecture of ESPRESSO is depicted in Fig. 5.1 with two components: Data Encryption Management and Keys Management. The request flow is as follows. Universal API is the gate of ESPRESSO, which can provide a wide range of interaction protocols allowing multiple CSPs to integrate ESPRESSO into their infrastructures. After receiving a request, Universal API delivers the request to Data Encryptor, which is responsible for processing users' data using algorithms implemented in Encryption Algorithms. Data Encryptor requests encryption key from Keys Management through Key Generator that is the starting point of the Keys Management component. Key Generator retrieves the key stored in Encrypted Key Storage if it already exists, and sends it to Key Encryptor to decrypt using a master key. If the requested key does not exist in the database, that means the user is new on the system or the key for that specific critical level is not yet generated, Key Generator creates a new key and sends

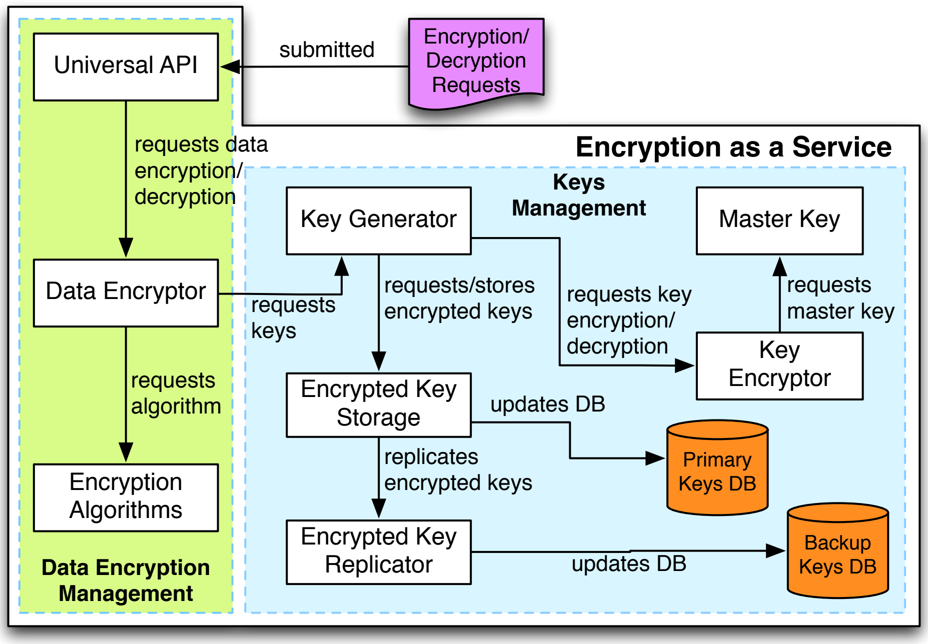


Figure 5.1: ESPRESSO overall architecture.

a key encryption request to Key Encryptor. The new key is then encrypted by the master key and sent back to Key Generator to store in Encrypted Key Storage. To assure the availability of encryption keys, encrypted keys are replicated and stored in Backup Keys DB.

5.3.2 Handling the Flexibility and Multi-user Scheme

To provide CSPs the flexibility in choosing a preferred encryption algorithm, ESPRESSO currently supports two algorithms: AES and Blowfish that are symmetric. By choosing symmetric algorithms, we eliminate the complexity of managing encryption key pairs, which are supposed to be stored on different servers. Moreover, they are less intensive than asymmetric algorithms in terms of processing time. Additional algorithms can also be integrated into the system without breaking the architecture of ESPRESSO thanks to its agile design.

On the critical level of data, ESPRESSO provides three different critical levels by using three key lengths: 128, 192 and 256 bits for all supported encryption algorithms. The longer key length guarantees the higher critical level of data. A less than 128-bit key may be broken by the modern machine while a more than 256-bit key increases the latency of the service. Thus, each user can have up to three keys corresponding to three critical levels: the highest level uses 256-bit keys and the lowest level uses 128-bit keys, respectively. For a certain user, all the data with the same critical level are encrypted by the same key. Since a CSP serves multiple users, each user is therefore identified by a user identification. We tie the user identification to the critical level and the encryption key by a tuple of $< \mathsf{user\_id}, \mathsf{critical\_level}, \mathsf{key\_string} >$ in the encryption key database. Additionally, keys are generated on request of the CSP for a specific user and critical level.

Table 5.1: Structure of the encryption keys table in MySQL

Field	**Type**	**Description**
key_id	Integer	Key identification: auto increment field
user_id	String	User identification
critical_level	Character	Critical level of user's data
key_string	String	Key string for encryption and decryption.

5.4 Implementation of ESPRESSO

We use Python to implement ESPRESSO based on its broad adoption and efficiency. We implement in Universal API the Web Server Gateway Interface (WSGI), which allows CSPs to deploy ESPRESSO as a WSGI service. The implementation of Universal API handles the WSGI requests, i.e., extracting user_id, critical_level and data, and converts them to internal requests, which are then forwarded to Data Encryptor. There are two functions in Data Encryptor: encrypt_data and decrypt_data. The encrypt_data function, which has three parameters: user_id, critical_level and data, prepares the encryption. It includes instructions for requesting the encryption key from Key Generator, initializing the encryption algorithm instance and finally invoking the execution of the encrypt_data function implemented in Encryption Algorithms. The algorithm selected by the CSP is saved in an INI configuration file with simple format, for example, [algorithm]name = AES. Instead of implementing all encryption algorithms by ourselves, we use a library namely PyCrypto [Litzenberger, 2014], which provides the implementation of various algorithms such as AES, DES, RSA, ElGamal.

Since the critical_level parameter is needed to retrieve the encryption key for data decryption in the future; however, users may not remember which level was set for the data in the past, we include this parameter in the encrypted data. For a data retrieval request, the CSP gets the encrypted data from the storage server and passes it to ESPRESSO with the user_id parameter in a decryption request. Data Encryptor first extracts the critical_level parameter from the encrypted data and then invokes the decryption by calling the decrypt_data function.

To provide users a friendly manner to specify the data critical level, we decode three proposed critical levels by three letters: **A** stands for the high level, **B** stands for the medium level and **C** stands for the low level. Theses three symbolic letters hide the complex technical details of critical levels from users who are not expert in the security domain. The CSPs integrating ESPRESSO should provide a usage guideline to make their users aware of the trade-off between the strength and required processing time of each level, i.e., **A** is the strongest level but it requires longer time to complete the encryption.

Encryption keys are generated by the Random library supported in Python. Each is a string including alphabet and numbers with length depending on the critical level. All keys are stored in a MySQL database whose the structure of the key table is shown in Table 5.1. Key Encryptor uses the same algorithm, i.e., AES or Blowfish, to encrypt the users' keys with a master key retrieved from Master Key. The implementation of Encrypted Key Storage and Encrypted Key Replicator handles the interaction with MySQL database, i.e., formulating the SQL query statements and executing the query.

Algorithm 8 Encryption and Decryption calls

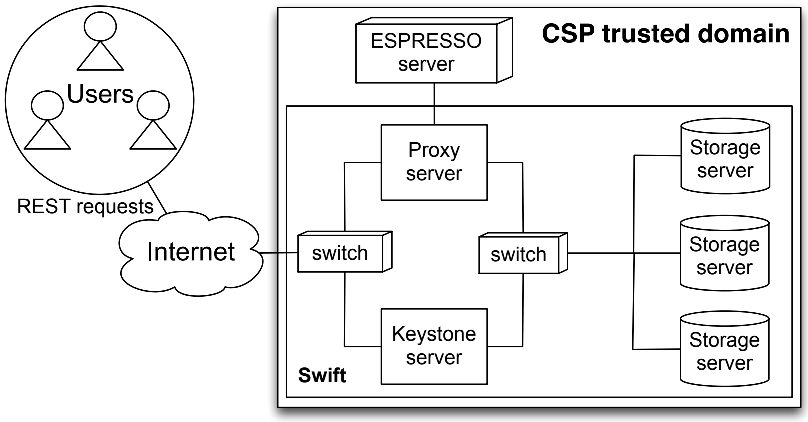
```
Input: data, user_id and critical_level for an encryption; encrypted_data and user_id for a decryption
    request; the ESPRESSO server address: server for both requests.
Output: Encrypted data for an encryption; plaintext data for a decryption request.
 1: connection = HTTPConnection(server); /*Create an HTTP/HTTPS connection*/
 2: connection.putrequest('EN', ''); /*'EN' for encryption and 'DE' for decryption*/
 3: for header in headers do /*Send all HTTP headers: user_id, critical_level*/
 4:     connection.putheader(header_name, header_value);
 5: end for
 6: for chunk in data do /*Send data by chunks*/
 7:     connection.send(chunk);
 8: end for
 9: response = connection.getresponse(); /*Waiting for response*/
10: Extract encrypted data or plaintext data from the response;
11: return
```

5.5 Integration of ESPRESSO

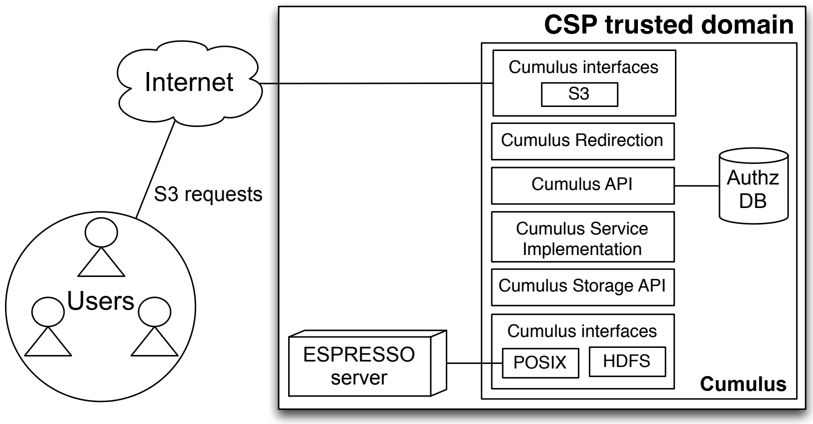
We choose Swift [OpenStack, 2014] and Cumulus [Bresnahan et al., 2011] to integrate ESPRESSO. These systems are open-source cloud platforms and they are widely used in both research community for experimental purpose and industry for commercial purpose. The integration involves determining a proper place in the source code of the storage systems where ESPRESSO is connected by using provided APIs and adding code instructions to realize that connection. The abstract pseudocode for encryption and decryption invocations from the storage systems is presented in Algorithm 8. Its detailed implementation depends on the target systems, programming language and supported library, e.g., Swift and Cumulus use Python while Eucalyptus uses Java. Generally, since ESPRESSO is implemented as a WSGI service, when a storage server requests for an encryption, a WSGI connection will be established (line 1). User's information and the data critical level are then passed by the connection header (lines $3-5$). The data file is divided into chunks and sent to ESPRESSO (lines $6-7$). When the data transmission is completed, ESPRESSO processes the data on its side while the storage server waits for the result (line 9) and continues the process after receiving data.

5.5.1 Integration of ESPRESSO into Swift

The integration of ESPRESSO into Swift is presented in Fig. 5.2a where we add the ESPRESSO server as a novel component of the Swift platform. ESPRESSO is deployed on a separate server rather than becoming an internal component of Swift. This avoids breaking down the Swift's code structure. Since the encryption and decryption happen only when users have downloading, uploading or updating requests, which correspond to GET and PUT methods in the RESTful API supported by Swift, all of modifications were made to the swift/proxy/controllers/obj.py module in the proxy server at two functions: GET(self, req) and PUT(self, req).



(a) With OpenStack/Swift.



(b) With Nimbus/Cumulus.

Figure 5.2: Integration of ESPRESSO into cloud storage platforms.

On the user's side, this integration does not complicate the data management operation. Only the uploading and updating requests require one more parameter to be added: the data critical level. For instance, if users use cURL [cURL, 2014] to interact with Swift for data management, the data critical level will be added as a novel header: `-H 'x-critical-level:A'`.

5.5.2 Integration of ESPRESSO into Cumulus

ESPRESSO has also been similarly integrated into the Cumulus storage system. The encryption and decryption invocations, presented in Algorithm 8 are added in the cumulus/cb/pycb/cbRequest.py module at two classes: cbGetObject(cbRequest) and cbPutObject(cbRequest). Like Swift, the total number of code lines added is less than 50 for both methods. This assesses the easy and light adoption of ESPRESSO in any cloud storage platform.

Since Cumulus supports the Amazon's S3 REST protocol, many client libraries and tools, including s3cmd [s3cmd, 2014], boto [boto, 2014] and jets3t [jets3t, 2014] can be leveraged by Cumulus users. For instance, if user uses s3cmd, a novel header will be added to specify the data critical level: `--add-header "critical-level: A"`. With the integrated system, if users do not specify the critical level, ESPRESSO will automatically use the highest level, i.e., **A**, to encrypt the user's data.

5.6 Experiments and Performance Evaluation

5.6.1 Experiment Setup

The integrated storage systems were deployed using two dedicated physical servers on the same rack of the Communications and Networks Lab (CNL) at the National University of Singapore. Swift and Cumulus were installed on the server `xx.xx.xx.64` and ESPRESSO was installed on the server `xx.xx.xx.65`. The servers are PowerEdge C6220 with Intel(R) Xeon(R) Processor E5-2640 2.50GHz, 24GB RAM. We used real data files which are downloaded from the Wikipedia archive [Wikipedia, 2014]. The file size varies from several MB to 4GB that allows us to evaluate the efficiency of the encryption algorithm with different loads. Three following performance metrics were considered for evaluation:

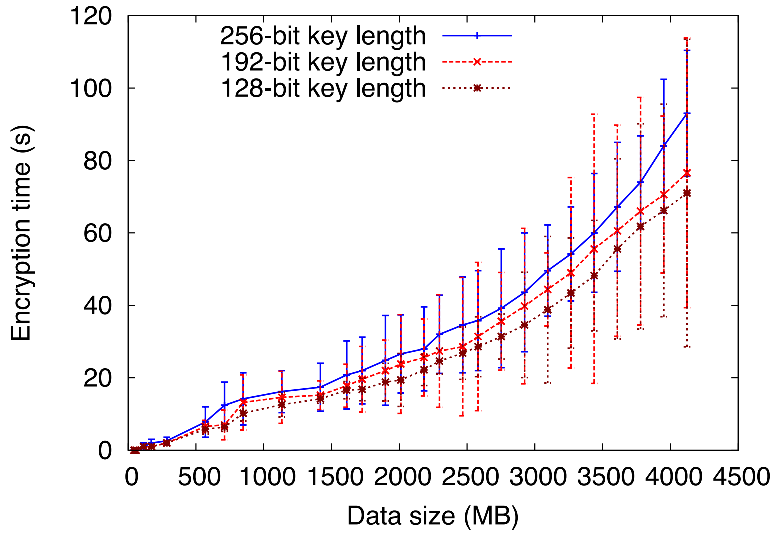
- Latency of encryption algorithms: To show the efficiency of encryption algorithms, we measured the encryption time with different key lengths for the same algorithm. In addition, we compared the encryption time of two different algorithms with the same key length.
- Latency of the integrated system with and without ESPRESSO: To show the transparency of ESPRESSO, the total operation time, i.e., sum of the data uploading time from the client to the storage server and the data encryption time of Swift with and without ESPRESSO were compared.
- Impact of network bandwidth: In this experiment, a remote client, which uses the Internet backbone for transferring data was deployed. Two different network connections: WiFi and wired connection were applied.

For each experiment, we performed 5 times to measure the average and standard deviation values of performance metrics. The second and third experiments were performed on both systems. However, due to the space limit and to avoid the redundancy, only results on Swift is shown. A comparison of total operation time between Swift and Cumulus is given in the analysis of the third experiment.

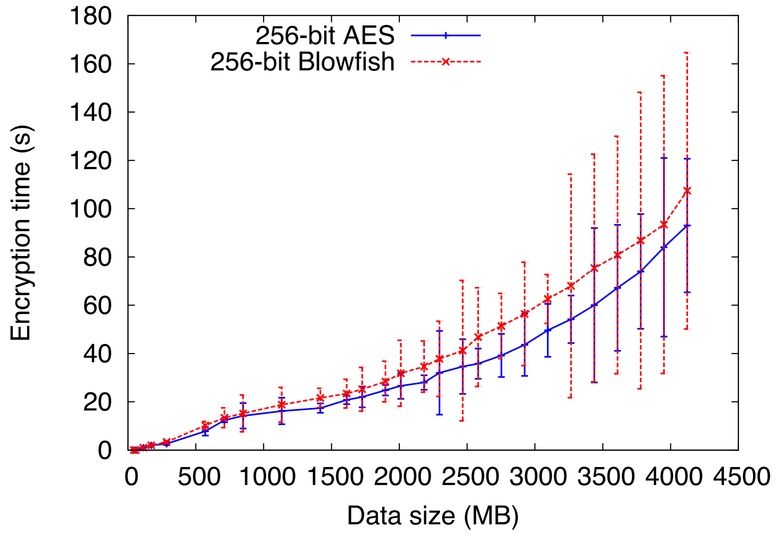
5.6.2 Performance Analysis

Evaluation of Encryption Algorithms: Fig. 5.3a presents the encryption latency of the AES algorithm with respect to the data size. We executed AES with three different key lengths: 128, 192 and 256-bits. It is expected that with the same key length, the larger data volume is, the longer time is needed to complete the encryption. With the largest file of 4GB, the encryption time with 256-bit key is 93s. Comparing the latency of AES with three key lengths, it is trivial that the longer key needs longer time to complete but it generates a more robust encryption, i.e., the data is more securely protected.

We also measured the encryption time of Blowfish and observed that there is the same behavior as AES. In Fig. 5.3b, we present the encryption time of AES and Blowfish with respect to the data size and with the same key length, 256 bits. The results show that Blowfish needs a longer time to complete the encryption for the same data compared to that of AES. Indeed, since Blowfish uses a 64-bit block size while AES uses a 128-bit block size, the number of blocks processed by Blowfish is doubled compared to that of AES. The processing transition between blocks leads to the overhead



(a) Encryption time of AES.

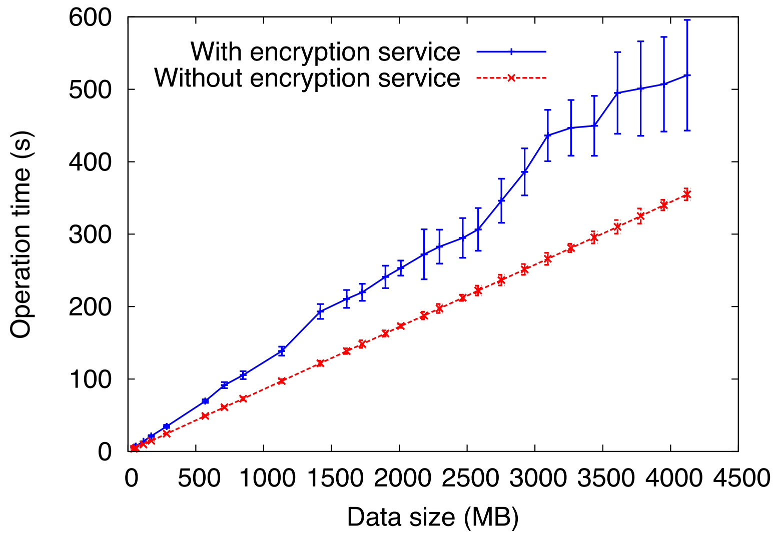


(b) AES vs. Blowfish.

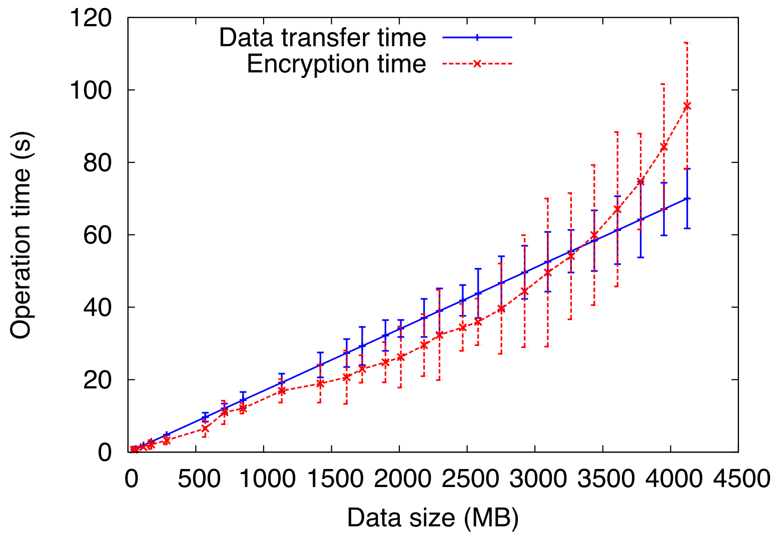
Figure 5.3: Performance of encryption algorithms.

of Blowfish. The results also show the nature of the encryption algorithms that the decryption time is almost the same as the encryption time as expected. Thus, to avoid the redundancy, we do not present the results on decryption time here. Additionally, the standard deviation of Blowfish is relatively higher than that of AES. Comparing the robustness of AES and Blowfish is out of scope of this work. Thus, choosing AES or Blowfish is based only on the preference of CSPs. This results also show that there is a large variation in encryption time of big data volume resulted in by the two encryption algorithms that perform stably with small-sized data. This may be due to the implementation of the encryption algorithms (AES and Blowfish) that we used the library supporting by Ubuntu.

Integrated System Validation: To validate the integrated system, we run the Swift client on a machine located in the same LAN to reduce the data transfer time between the client and Swift. The encryption



(a) With and without ESPRESSO.



(b) Details of the encryption time.

Figure 5.4: Uploading time with/without ESPRESSO and details of encryption time.

algorithm is AES and the critical level is **A**. Fig. 5.4a depicts the total operation time for uploading requests of Swift with and without ESPRESSO. In the case without ESPRESSO, the total operation time can be considered as the data transfer time from the client to the Swift server. It is expected that the total operation time of Swift with ESPRESSO is longer than that without ESPRESSO since an additional time is needed for data encryption. This overhead includes data transfer time from Swift to ESPRESSO, the encryption time and the transfer time from ESPRESSO back to Swift for resulted data. In the worst case, the total operation time increases 63.95%. The details of the encryption time overhead are presented in Fig. 5.4b. While the data transfer time between the Swift and ESPRESSO servers is small and not affected by other users since the servers are installed on the same rack, the encryption time dominates when the file size is large, i.e., larger than 3.5GB. Even though we assume 4GB files as the worst case scenario, which roughly corresponds to the total content of a single-sided DVD, one may

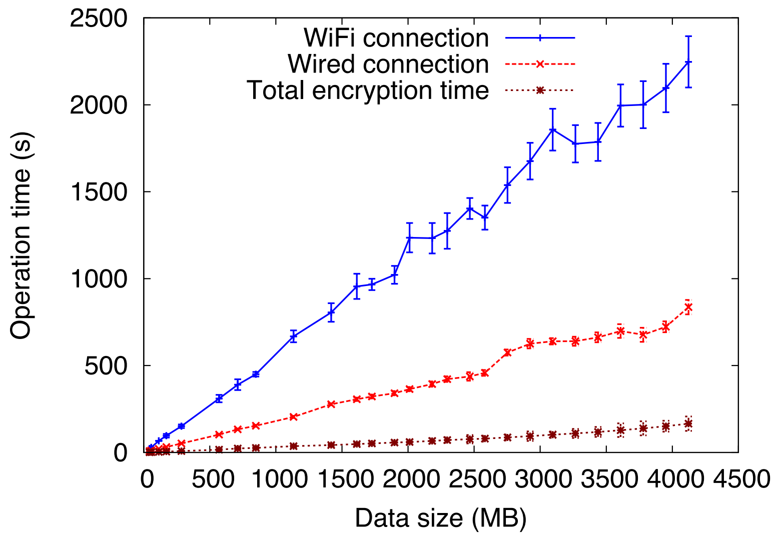


Figure 5.5: Upload time from a distant client.

have larger files to store. However, the results show that it is strongly discouraged to store large files to not significantly degrade the performance of the system.

Impact of Network Bandwidth: In practice, users are not always located nearby the cloud. Therefore, the data transfer time from the user's location to the cloud is much larger than that presented in previous experiment. Indeed, we did the third experiment by running the client machine locating 3 kms from the Swift/Cumulus servers, using the Internet backbone for transferring data. In Fig. 5.5, we present the total operation time of Swift for uploading requests when the client uses the WiFi and wired connection. The average uploading speed is 1.54 Mbps and 6.72 Mbps, respectively. The results show that the data transfer time from the client to the Swift server dominates in both connections. With the largest file with the WiFi connection, the total operation time is 37.45 mins while the encryption time overhead is only 2.75 mins, corresponding to 7.34% of the total operation time. From the point of view of a user who is sensitive with the latency, he may still not accept such overhead. However, considering the security aspect that the user's data is securely protected by CSPs, we believe that the cost represented by the time overhead is worth for such a security service. Fig. 5.6 presents the comparison of operation time between Swift and Cumulus when the remote client uses wired connection. The operation times of both systems are almost the same. While Swift needs longer time for replicating the data with 3 copies, Cumulus does not provide the replication service. However, this overhead on Swift is compromised by the fluctuation of the data transfer time.

5.7 Chapter Summary

In this chapter, we proposed ESPRESSO, a standalone and transparent encryption service for cloud storage systems [Kang et al., 2014a]. It provides CSPs the flexibility of choosing their preferred encryption algorithm by supporting two algorithms: AES and Blowfish. With the flexible design, CSPs can easily integrate ESPRESSO without heavy modification and implementation of their infrastructures. It is an

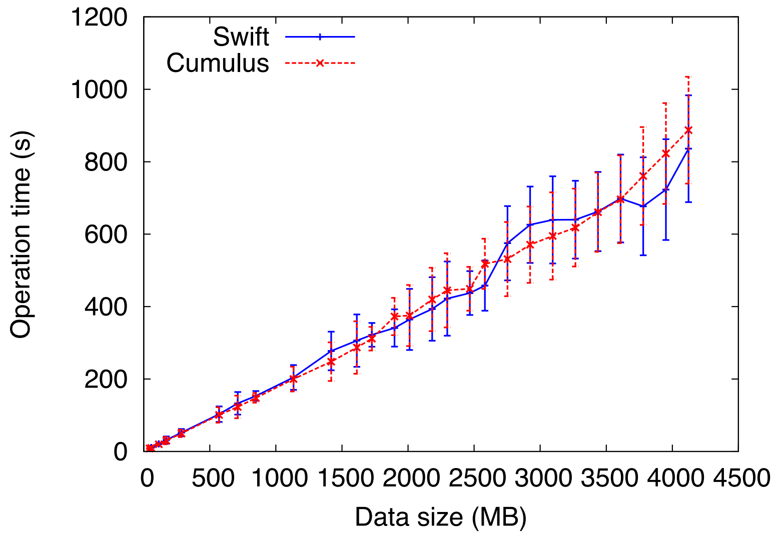


Figure 5.6: Swift vs. Cumulus performance.

important contribution that the integrated system does not require much effort from users to make their data protected, i.e., users only need to specify one of three data critical levels, which is provided by ESPRESSO. All these advantages assess the effectiveness of ESPRESSO to be integrated into any CSP on the production level. Nevertheless, this chapter only addresses the privacy issue for data at rest, i.e., the data stored in public clouds only for backup purpose but not for computation and analysis purpose. Protecting privacy of data in use in public clouds requires advanced methods that enable the computation on encrypted data without the need of secret key for decrypting data so as to not reveal the plaintext data during the computation. We present in the next chapter our proposed solution applying to a specific case study: genomic computation in public clouds.

Chapter 6

Secure and Fast Mapping of Genomic Sequences on Public Clouds

This chapter presents our case study to demonstrate the need of public clouds for handling big amount of data and the need of security mechanisms to protect the data privacy when processing in an untrusted computing platform. We consider the genomic computation, a specific research domain that is currently facing the exponential growth of genomic data and also requires the highest level of security for the data privacy. Furthermore, genomic sequences are also divisible into multiple chunks so that we can apply the proposed approaches presented in the previous chapters to improve the perfomance of genomic applications. We design and implement an entire secure framework for genomic data processing on public clouds. Based on this framework, we propose a 3-encryption-scheme model for genomic sequence mapping (3EGSM), an important phase of genomic computation. The model protects not only genomic sequences but also the intermediate and final computation results when processing on clouds. We evaluate the proposed framework through intensive experiments using real genomic data.

6.1 Research Motivation and Objectives

The rapid advances in genomic technologies have changed the scale and scope of genomic data processing. While the conventional Sanger machine spends an entire year to sequence a human genome, the next-generation genomic sequencing machines such as 454 Life Sciences, Illumina and Applied Biosystems can sequence a genome in a few days at a lower cost [Shaffer, 2007]. This evolution allows the research community to generate genomic data more easily and to perform the analysis more efficiently. Genomic research is therefore progressing at a fast pace and becoming integrated into mainstream medicine, leading to the share of genomic data among institutions for collaborations. To protect the privacy of genomic donors, who face the risk of revealing sensitive information such as ethnic heritage, disease predispositions and many other phenotypic traits, a number of regulations declaring health information privacy have been edited and applied in reality such as the US HIPAA [Kulynych and Korn, 2003] and the European Data Protection Directive [Robinson et al., 2009]. To comply with these data protection standards, different approaches have been proposed such as differential privacy and data anonymization. While differential privacy guarantees the data security, it reduces the accuracy of genomic computation due to the external information added [Fienberg et al., 2011]. On the other hand, data anonymization, which removes the confidential information, poses the threat of re-identification through examining individual phenotypes [Zhou et al., 2011]. This drives the need for a novel approach to protect data privacy more securely while ensuring the accuracy of genomic computation.

While the emergence of genomic technologies eases the sequencing and processing of genomic sequences, it poses a new challenge concerning the exponential growth of petabyte-scale genomic data. For instance, the 1000 Genomes Project has aimed to generate about 15 terabytes of sequences using next generation sequencing technologies [Li et al., 2009]. Storing and processing this huge volume of data exceed the current capacities of in-house trusted servers of institutions. Scaling up the existing private infrastructures might be a solution for each institution, but it costs a lot to do so and the infrastructure may be under-utilized when demand is low. With the immense computational capacities, quasi-unlimited storage space and guaranteed bandwidth connections, public clouds have become an attractive candidate for genomic data storage and processing at low usage cost. For instance, taking the advantages of public clouds, Seven Bridges Genomics has adopted Amazon Web Services cloud to process petabytes of genomic data for thousands of users since 2012 [Rilak et al., 2014].

Moving genomic data from in-house trusted servers to public clouds for storage and processing makes the security breach become bigger. With the intrinsic characteristics of public clouds such as multi-user environment and physical resource sharing among users' requests, preventing the data leakage and loss becomes harder and exceeds the capacity of the current protection standards. Adversaries on public clouds being either honest-but-curious or malicious can collect the leaked data and infer useful information. To solve this problem, several approaches have been studied in the literature [Bruekers et al., 2008, Jha et al., 2008, Lauter et al., 2014], which proposed to use the cryptographic solution such as *secure multi-party computation* (SMC) [Huang et al., 2011a] and *homomorphic encryption* (HE) [Gentry, 2009]. While SMC is designed for a secure computation that involves multiple parties who do not want to reveal their respective input data to others, HE is a more general solution, which allows one to seal data in a metaphorical vault that can be opened by the owner having the secret key. In HE, the encrypted data can be processed without the secret key on untrusted servers such as public clouds. Compared to SMC, HE has a number of advantages since it enables more flexible scenarios and functionalities. Indeed, it requires less interactions or no interaction is required for single-key encryption applications, thereby reducing the communication complexity.

Due to recent improvements in HE including the techniques that avoid the costly bootstrapping procedure for fixed number of computations, we advocate for HE to protect the data privacy in this work. However, purely using HE may result in very low performance due to its heavy computation, we therefore propose to use other encryption schemes in the computation where HE is not needed, thereby improving the performance of the system. For instance, a *keyed hash function* can be used to encrypt genome sequences. An *order-preserving encryption* (OPE) scheme, which preserves the numerical ordering of plaintexts, can be used to encrypt the position of important nucleotides. A concrete model, which combines existing approaches, therefore needs to be designed for large-scale and efficient computation of genomic data on public clouds while securely protecting the data privacy. We aim at providing genomic researchers efficient means to carry out their research without any concern about the technical issue of protection of data privacy and application performance.

In this chapter, we present an entire secure framework for genomic data processing on public clouds. The framework allows clinics and research institutions to perform computations at large scale and to handle large amount of genomic data, leveraging on public cloud resources. Based on this framework, we propose a 3-encryption-scheme model for genomic sequence mapping (3EGSM). The model, which combines the three encryption schemes mentioned above, protects not only genomic sequences but also the intermediate and final computation results when processing on public clouds. We focus on genomic sequence mapping also known as *read mapping* [Trapnell and Salzberg, 2009] since it is one of the most important steps in the area of genomic computation. Indeed, after collecting from sequencing machines, the raw genomic data, which consists of millions of short nucleotide sequences (25–250 bp) called *reads*, is featureless if there is no other supplement information. To make reads become meaningful, researchers need to perform the read mapping that aligns these reads to a reference genome to find the locations where each read occurs. The results then can be used for further analysis such as paternity test, personalized medicine, single nucleotide polymorphism (SNP) discovery, etc.

Among existing read mapping algorithms, we advocate for the *seed-and-extend* algorithm [Baeza-yates and Perleberg, 1992], the most efficient mapping technique [Schbath et al., 2012]. The *seeding* step finds the sub-sequences (called *seeds*) that exactly match in both the read and the reference genome, and the *extension* step extends these matching seeds into a longer alignment for the whole read. The length of seeds depends on the number of mismatches allowed between the read and the segment on the reference genome where the read aligns. While the original design of the seed-and-extend algorithm considers the case that both seeding and extension steps are run on the same server, to leverage on public clouds, we need to adopt the algorithm in the way that heavy computations will be delegated to public clouds. Therefore, we propose to split the mapping procedure along the two steps of the seed-and-extend algorithm. The seeding step is then performed on public clouds, whereas the extension step is performed on the trusted server. To reduce the workload on the trusted server, we propose the technique to reduce as much as possible the number of extensions needed to be performed.

In summary, the contributions of this chapter are as follows.

- We design an entire secure framework for genomic data processing on public clouds. The framework provides means for researchers to perform large-scale computations by leveraging on the immense and cheap computing resources of public clouds.

- We propose a 3-encryption-scheme model for genomic sequence mapping (3EGSM) to protect input data, intermediate and final results against adversaries on public clouds. Furthermore, this fine-grained model allows us to improve the performance of the system.

- We implement and evaluate our prototype through intensive experiments using real genomic data. The results assess the validity and feasibility of the proposed approach in realistic systems.

6.2 Threat Model

Moving genomic data for storage and processing on public clouds poses severe security threats. The main threat is the identification of the genomic sequences. Since the genome is produced by clinical studies, the identification of its donor, once identified, could be linked to the disease under the study. The donor then suffers the consequence risks such as denial of access to health insurance, education and employment. Even though HIPAA has required to anonymize the genomic data before making it public, this protection approach is not sufficient as re-identification can be performed through examining individual phenotypes [Zhou et al., 2011]. For the reads, even though they are just short nucleotide sequences, the donor's phenotypes could be observable after their locations on the reference genome are discovered. Thus, it is important to protect not only the genomic sequence itself but also all the meta-data of the sequence such as mapping locations, i.e., offsets, SNP, etc.

The cloud adversaries, either providers or the users sharing the same infrastructure, can be malicious or honest-but-curious. While a malicious adversary tries to compromise and change the results of genomic applications, making the results no longer accurate, an honest-but-curious adversary purposely collects the data publicly exposed and then infers useful information from the obtained data. On the aspect of data integrity and correctness of computation, malicious adversaries cause more significant consequences since they can still compromise the data by modifying any bit even though data is encrypted. Ensuring data integrity and preventing malicious adversary from changing data therefore require a more complex mechanism, e.g., zero knowledge approach [Chiesa et al., 2015]. In this chapter, since we address the problem of data privacy and leakage issue, we focus on the security threat coming from honest-but-curious adversaries, which may use the leaked data for their own purpose. For instance, an insurance company collects sensitive medical data of its clients and performs denial of access to services. We propose the solution to prevent the data from being leaked to the honest-but-curious adversaries and to ensure that the leaked data is not useful for them.

We also assume that there exists a trusted server, which is responsible for performing the computations involving the secret key such as data encryption/decryption and conversion from one encryption scheme to another scheme. We argue that the 3EGSM model ensures the confidentiality of genomic data when storing and processing on public clouds. Indeed, cloud adversaries can only see the encrypted genomic sequences represented by keyed hash values and computation results, i.e., intermediate and final results, which are also encrypted by HE or OPE. The adversaries can collect encrypted data and send to a remote attacker for further actions. However, such an attacker would not have the legitimate secret key, thus would not be able to decrypt the data. Furthermore, it is infeasible to generate the original genome sequences from their keyed hash values. Inferring useful information directly from encrypted data is also impossible since the encryption schemes used in the 3EGSM model are proved not to reveal any information about the plaintext data due to recent improvements [Menezes et al., 1996]. Finally, we assume that there exist network security techniques for securely transferring data from the trusted server to public clouds. This adds another security layer to protect the data even though the data has been encrypted. Thus, man-in-the-middle attack can collect the data but would not be able to perform any analysis or inference to retrieve useful information.

6.3 Data Encryption Techniques

In this section, we give an introductive background on the data encryption techniques used in the 3EGSM model. We describe how the model can protect the data privacy during computation and the purpose of each encryption scheme in our proposed framework.

6.3.1 Hash

Cryptographic hash function, i.e., keyed hash function, which uses a secret key to encrypt the data, has been widely used as one of encryption techniques [Bellare et al., 1996]. As formally defined in [Bakhtiari et al., 1995], a keyed hash function $H(\cdot)$ is a class of hash functions $\{h_k : k \in V_n\}$ indexed by a key k such that function $h_k(\cdot) : M \to V_m$ maps an arbitrary length message M to a fixed length message digest V_m of length m. The larger the value of m, the stronger the hash function against brute-force attacks [Menezes et al., 1996]. For instance, SHA-1 is more secure when using a 160-bit hash function compared to a 128-bit one.

In read mapping using the seed-and-extend algorithm, since all seeds from the reads as well as the reference genome need to be protected when moving to public clouds, keyed hash function becomes an appropriate technique to encrypt the seeds. Regardless of the length of seeds, keyed hash function always produces fixed length keyed hash values, which are different for each individual depending on the secret key. This eases the searching of the exact matches of seeds in both the read and the reference genome, thereby improving the performance of the entire framework.

6.3.2 Homomorphic Encryption

Homomorphic encryption has been extensively studied in the past few years since it allows complex mathematical operations to be performed on encrypted data without first decrypting the data and without any knowledge of the secret decryption key. This feature makes homomorphic encryption to be used for protecting the privacy of data that involves the computation performed in untrusted domains such as public clouds without revealing the original plaintext data. As formally defined in [Fontaine and Galand, 2007], an HE scheme has the following correctness property:

$$\begin{aligned} \forall m_1, m_2 \in \mu, \quad & E(m_1 +_\mu m_2) \leftarrow E(m_1) +_\varrho E(m_2), \\ & E(m_1 *_\mu m_2) \leftarrow E(m_1) *_\varrho E(m_2), \end{aligned}$$

where μ is the plaintext space and E is the encryption function; $+_\mu$ and $+_\varrho$ are addition operations in the plaintext and ciphertext spaces, respectively. Similarly, $*_\mu$ and $*_\varrho$ are multiplication operations in the plaintext and ciphertext spaces, respectively. This means that the result is directly computed from input ciphertexts and it is decrypted to obtain the plaintext result only by the legitimate secret key.

In all known homomorphic encryption schemes, ciphertexts inherently contain a certain amount of noises. This noise accumulates during homomorphic operations. If the noise is too large, the ciphertext cannot be decrypted even with the correct secret key. Due to this characteristic, the first construction of homomorphic encryption is a *somewhat homomorphic encryption* (SWHE) scheme, which can perform a limited number of operations before the noise grows large enough to cause the decryption faillure [Naehrig et al., 2011]. To make SWHE become a *fully homomorphic encryption* (FHE) scheme that can evaluate an arbitrary number of additions and multiplications, a technique is needed to refresh ciphertext constantly to reduce the noise level before the encrypted value is used for another arithmetic operation. Depending on the construction of the homomorphic encryption scheme, a different technique is used for this purpose. If the scheme construction is based on the so called "learning with errors" (LWE) problem [Regev, 2005], the scheme is then bootstrappable. Thus, the bootstrapping technique is used to refresh the ciphertexts [Brakerski et al., 2014]. If the scheme uses ideal lattices for its construction, it is not bootstrappable by the bootstrapping technique. The "squashing" technique is then used to reduce the noise in the ciphertexts [van Dijk et al., 2010].

However, the bootstrapping and squashing techniques are heavily computational, thereby degrading the overall performance of the system. The *leveled fully homomorphic encryption* (Leveled FHE) scheme has been designed to avoid the use of these heavyweight techniques. Leveled FHE allows one to set the parameters of the scheme so that it can perform the computation of any specified function. It uses a modulus switching technique to reduce the noise of encrypted data without knowing the secret key and this method is much simpler than bootstrapping [Brakerski et al., 2014].

During the read mapping process, since the offsets of the seeds on both the read and the reference genome involve in computations on public clouds where they need to be protected, we use homomorphic encryption as a solution to encrypt the offsets. However, it is infeasible to perform the comparison between two plaintext integers by their ciphertexts, we thus apply the order-preserving encryption scheme to encrypt data when comparison is required. The data therefore needs to be sent back to the trusted server for conversion from HE to OPE during the mapping process.

6.3.3 Order-Preserving Encryption

Order-preserving encryption (OPE) is a deterministic symmetric encryption scheme that preserves numerical ordering of the plaintexts [Boldyreva et al., 2011]. It was firstly proposed in the database community to support efficient range queries on encrypted data [Agrawal et al., 2004]. For $A, B \subseteq \mathbb{N}$ with $|A| \leq |B|$, a function $f : A \rightarrow B$ is order-preserving if for all $i, j \in A, f(i) > f(j)$ iff $i > j$. In [Popa et al., 2013], the authors analyzed the security of OPE and proved that the encrypted values do not reveal additional information about the plaintext values besides their order, i.e., IND-OCPA [Boldyreva et al., 2009, Popa et al., 2013, Yum et al., 2011]. Therefore, we use OPE to support the comparison operations during execution of the seed-and-extend algorithm on encrypted data.

6.4 A Secure Framework for Read Mapping on Public Clouds

6.4.1 System Design

Typically, specialized clinics and genome research institutions are the sources of raw genomic sequences. These institutions conventionally operate several computers or a small computing cluster for genomic data storage and analysis. This private infrastructure is usually deployed in the local (trusted) domain of the institutions. Thus, the data can be processed in the raw (plaintext) format without any security concern. However, the exponential growth of genomic data makes the storage and computing

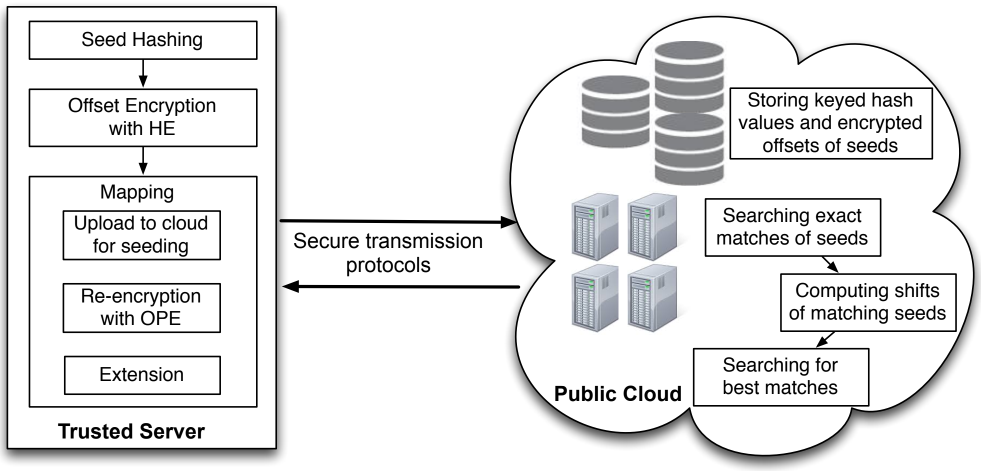
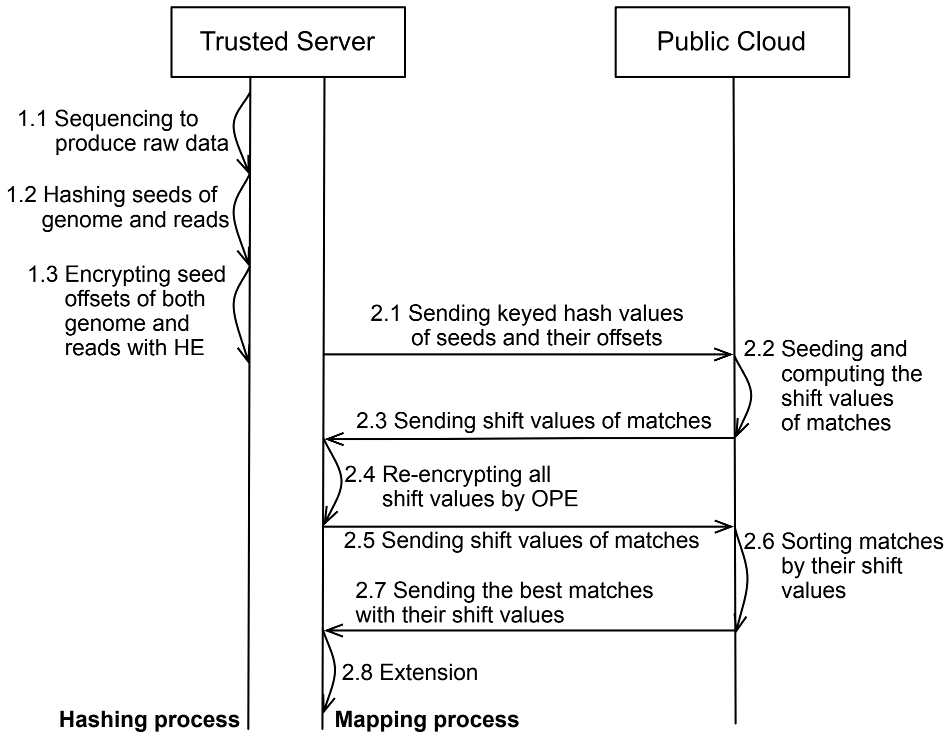

Seed Hashing
Offset Encryption with HE
Mapping
Upload to cloud for seeding
Re-encryption with OPE
Extension
Trusted Server
Secure transmission protocols
Storing keyed hash values and encrypted offsets of seeds
Searching exact matches of seeds
Computing shifts of matching seeds
Searching for best matches
Public Cloud

Figure 6.1: Proposed system architecture.

requirements exceed the capacity of the private infrastructure. In the architecture shown in Fig. 6.1, we propose to delegate all heavyweight computations to the public cloud, which has large-scale storage and computing resources. The private infrastructure continues to be used for the light computations, which involve the plaintext data, and data encryption/decryption to protect the privacy before sending to the public cloud for analysis. The computing platform deployed in the public cloud should compose of both storage and computing resources. For instance, clinics can request for storage space from Amazon S3[1] and virtual machine instances for computing resources from Amazone EC2[2] to deploy their own platforms. Depending on the storage and computational requirements, clinics may adjust the amount of resources requested on the public cloud to achieve the desired performance and to save the cost paid for cloud providers. As previously mentioned, the connection between the trusted server and the public cloud can be established by any secure transmission protocol even though all genomic data exchanged between the trusted server and the public cloud has been encrypted.

Framework Generalization for Other Computations: We believe that the proposed architecture is generalized not only for read mapping but also for any other large-scale data intensive genomic application. On the architectural aspect, any genomic application needs the trusted server to perform computations that involve plaintext data and encryption keys. The public cloud component is responsible for heavy computations that involve encrypted data. On the deployment aspect, since the trusted server is under the administration of the users, deployment of new software and applications is straightforward. On public clouds, since applications need to be installed on each virtual machine (VM), which will be released after a reservation period, a VM image that includes all necessary applications needs to be created to ease the deployment step and to avoid unforced errors. Another effort from users is required to adapt the application before integration. While most of genomic algorithms have been designed for running on a single computer, adapting the application to be able to run on two remotely connected machines (clusters) requires expertise from software designers for the algorithm partition process. Furthermore, the algorithms also need to be adapted and integrated with Hash, OPE and HE to be able to

[1]Amazon S3: `http://aws.amazon.com/s3`
[2]Amazon EC2: `http://aws.amazon.com/ec2`


Trusted Server
Public Cloud
1.1 Sequencing to produce raw data
1.2 Hashing seeds of genome and reads
1.3 Encrypting seed offsets of both genome and reads with HE
2.1 Sending keyed hash values of seeds and their offsets
2.2 Seeding and computing the shift values of matches
2.3 Sending shift values of matches
2.4 Re-encrypting all shift values by OPE
2.5 Sending shift values of matches
2.6 Sorting matches by their shift values
2.7 Sending the best matches with their shift values
2.8 Extension
Hashing process
Mapping process
Figure 6.2: Interaction diagram between trusted server and public cloud for read mapping.

handle encrypted data. We also argue that the 3EGSM model is sufficient for any other genomic application to protect the data privacy. An application, which manipulates the genomic sequences, just needs the hash function for protecting the sequences. In case that the application needs to perform the computation with each single nucleotide, HE is then used to encrypt the encodes of nucleotides, e.g., $A \rightarrow 00, C \rightarrow 01, T \rightarrow 10, G \rightarrow 11$. Furthermore, HE and OPE are also used for all mathematical operations that involve the offset of nucleotides.

6.4.2 Interaction Diagram for Read Mapping

In this section, we describe the detailed interaction between the trusted server and the public cloud for the entire read mapping process including the data preparation phase.

Hashing Process: As shown in Fig. 6.2, the data preparation phase is performed by the hashing process on the trusted server. It composes of three steps from step 1.1 to 1.3 where step 1.1 is performed by sequencing machines such as Illumina/Solexa. Steps 1.2 and 1.3 are mapping-specific tasks required by the seed-and-extend algorithm. The trusted server needs to divide the reads and the reference genome into multiple seeds and encrypt them. The seeds are encrypted by a keyed hash function in step 1.2. The offsets of seeds, i.e., the position they occur in the corresponding read or reference genome, are encrypted by homomorphic encryption in step 1.3.

Mapping Process: The mapping process, depicted by step 2.1 to step 2.8 as shown in Fig. 6.2, involves the public cloud and requires the data communication between the trusted server and the public cloud.

In step 2.1, the trusted server sends the encrypted data including keyed hash values of seeds and their offsets on both the read and the reference genome to the public cloud. This step can be performed well ahead in time of the seeding process and the encrypted data of the reference genome is sent only once, except there is any change in the length of seeds. For any read that will be mapped, its encrypted data also needs to be sent to the cloud. Given that all necessary data has been transferred to the public cloud, step 2.2 will search for the exact matches of seeds in both the read and the reference genome. While the traditional seed-and-extend implementations perform an extension for every match found even though the extension may not be successful because of the number of mismatches, we apply the *q-gram filtering* method to reduce the number of extensions needed to be performed, thereby reducing the load on the trusted server. To do so, the read needs to be chopped in the overlapping manner to extract all l-mers, i.e., a seed with l bp. For instance, in the read of ACTG, the 2-mers will be AC, CT and TG. If a sufficient number of l-mers map in a small segment of the reference genome, the segment is then chosen to perform an extension with the read. We refer the reader to [David et al., 2011] for the mathematical formula to compute such threshold and its detailed description.

To count the number of matches of read seeds in a segment of the reference genome, we compute a value so called *shift*, which is the position where the read aligns on the reference genome with respect to the position of the exact match of the seed. Mathematically, let m and n denote the offsets of the matching seeds on the read and the reference genome, respectively. The shift value is computed as $s = n - m$. If multiple matches have the same shift value, then these matches are in the same segment of the reference genome. However, since the shift values are computed from the offsets that are encrypted by HE, we cannot perform the comparison of the shift values to verify whether they are equal or not (with the current implementation of HE). Therefore, the public cloud has to send the shift values back to the trusted server for conversion from HE to OPE, which preserves the numerical order of the shift values. These two steps are shown in step 2.3 and 2.4 in Fig. 6.2.

All the shift values re-encrypted by OPE are sent to the public cloud again for verifying with the threshold (step 2.5). The public cloud sorts all the shift values in the ascending order, then checks each shift value for its number of matches (step 2.6). It is worth mentioning that, by using this approach, one can choose to perform only one extension for the segment of the reference genome, which has the highest number of matches based on the shift values. Thus, as shown in Fig. 6.2, the public cloud also sends back the best matching position of the read on the reference genome (step 2.7). The extension (step 2.8) performed on the trusted server can be done by any dynamic algorithm as presented in [Schbath et al., 2012]. Obviously, the user needs to extract the segment on the reference genome, which is in the plaintext format to perform the extension along with the read.

6.5 Security Analysis

In this section, we analyze the security provided by the employed encryption schemes and the proposed framework. We present an informal proof to show that the integration of the 3EGSM model into the proposed framework provides complete security for genomic data.

Proposition 6.5.1 *The three encryption schemes (keyed hash function, homomorphic encryption and order-preserving encryption) used in the proposed framework ensure that no sensitive information is* *leaked or publicly exposed in their respective processing step.*

Proof:*(Sketch.)* The three encryption schemes have been well proven in the literature that they are persistent against attacks. We refer the readers to [Menezes et al., 1996] for the security proof of keyed hash function, [Fontaine and Galand, 2007] for HE and [Popa et al., 2013] for OPE. Given the "correct" integration of each scheme into respective programs presented in the previous section, the genomic data including genomic sequences, offset values and shift values are then correctly encrypted and they are persistent against attacks. We argue that the correct integration depends on the behavior of program developers. If the developers behave maliciously or collude with external attackers, this scenario is then out of scope of the paper since the developers are no longer honest-but-curious users. It is also noted that the encryption process, which involves encryption keys, is executed in the private trusted server that is assumed to be secure under the administration of the users. We thus claim that the encryption schemes individually ensure the privacy of the data processed. ■

We now analyze the security provided by the entire framework. We show that the combination of all three encryption schemes into the framework protects the privacy and reveals no sensitive information during the processing on clouds.

Proposition 6.5.2 *Given the correct implementation and integration of the 3EGSM model, the proposed framework ensures that no data including input data, intermediate and final results, have been leaked during the processing on clouds.*

Proof: Let $p = F(r, R)$ denote the function that takes two input parameters: r as the short read and R as the reference genome. The function results in p as the position that the short read aligns on the reference genome. $F(r, R)$ is a combination of all programs that are described in the previous section: `hashing_read`, `hashing_genome`, `seeding`, `re_encrypt_ope`, `sorting`, `ope_decryption` and `extension`. Given that $F(r, R)$ is correctly implemented according to the protocol presented in Fig. 6.2, we prove that the execution of $F(r, R)$ will not expose any information to other users on public clouds. Indeed, let us hypothesize that honest-but-curious adversaries are able to collect sensitive information during the execution of $F(r, R)$ on public clouds. Since r and R have been encrypted on the trusted server, the data, which are stored and processed on public clouds, are $H(r)$, $H(R)$, $E(offs)$, $E(shifts)$ and $E(p)$ where $H(r)$ is the keyed hash value of short read, $H(R)$ is the keyed hash value of the reference genome, $E(offs)$ are the encrypted values of the offsets of the seeds, $E(shifts)$ are the encrypted values of the shifts, and $E(p)$ is the encrypted value of the mapping position. Since all data exposed to adversaries are encrypted, the adversaries can collect meaningful information if and only if one of the three following scenarios happens:

- Adversaries can break the encryption schemes by a particular attack;
- Adversaries have the encryption key and perform decryption of encrypted data; and
- Data is decrypted during the processing on clouds.

If the first scenario happens, it then implies that Proposition 6.5.1 does not hold true. If the second scenario happens, it implies that the encryption key has been leaked. Since the encryption key is managed by the user on the trusted server, this scenario is then contradictory with our assumption that the trusted server is secure and the user is honest to not reveal the encryption key to a third party. If the third scenario happens, it implies that the implementation of the framework is not correct since all the encryption schemes have been proven to be able to handle the encrypted data. Furthermore, as we describe in Section 6.6, the input parameters of each program running on public clouds do not include the encryption key that is then not available on public clouds. These contradictions confirm that the initial hypothesis must be false, thereby proving Proposition 6.5.2. ■

6.6 System Implementation

In this section, we present the implementation of the proposed framework. We first present the libraries used in the 3EGSM model to protect the data privacy. We then describe the implementation of the modules in the system as well as their input/output interfaces. We discuss the challenges raised by the application of the 3EGSM model into the proposed framework and propose workaround solutions to overcome these challenges.

6.6.1 Helper Libraries

We implement our system in C/C++, using the following supporting libraries for the encryption purpose.

6.6.1.1 Hash Library

To produce keyed hash value of data, we used `HMAC`, which has been included as a module of the `openssl` library. It provides cryptographic hash functions such as `MD5` and `SHA-1`. In this paper, we used `HMAC-SHA1` that produces a 160-bit hash value from a string with arbitrary length. The secret key is given by users and the longer secret key makes the hash stronger against the brute-force attack.

6.6.1.2 Homomorphic Encryption with `HElib`

As far as we know, the most efficient fully homomorphic encryption scheme has been implemented by the IBM research team conducted by S. Halevi and V. Shoup. The implementation is called Homomorphic-Encryption Library (`HElib`) and it can be found at the address: `https://github.com/shaih/HElib`. `HElib` implements the Learning With Errors over Rings (RLWE) encryption scheme along with many optimizations to make homomorphic evaluations run faster. It is implemented in C++ and uses the NTL mathematical library. For the detailed description on the design and implementation of `HElib`, we refer the reader to [Shoup and Halevi, 2012].

6.6.1.3 OPE Library

For the OPE library, we used the implementation extracted from the CryptDB project [Popa et al., 2011]. It is noted that a more recent version of OPE is shown to be more secure and faster [Popa et al., 2013]. However, we did not use the version in [Popa et al., 2013] due to its non-availability.

6.6.2 Module Implementation

We now present the detailed implementation of each module in the proposed framework. We focus on the input/output interface of the modules rather than on the integration of the supporting libraries into each module as many available tutorials have been published to guide the use of the libraries.

6.6.2.1 Hashing of Reads and Reference Genome

As previously mentioned, the seeds of a read and the reference genome need to be hashed before being sent to public clouds to protect the privacy. The length of a seed depends on the number of mismatches

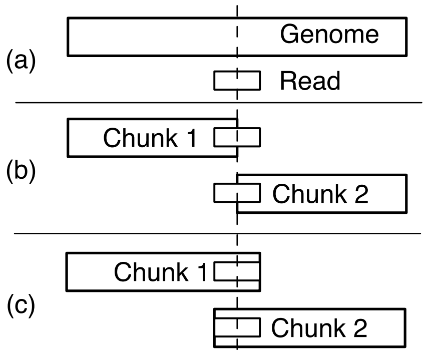


Figure 6.3: Division example. (a) Genome and short read. (b) Division without overlapping. (c) Division with overlapping.

allowed between the read and a segment of the reference genome. This value also known as *edit distance* is defined as the minimum number of editing operations including insertion, deletion and substitution, needed to convert one into the other. Let d denote the allowed edit distance, the length of a seed, denoted as l, is defined as $l = L/(d+1)$ where L is the length of the read. A seed of length l is also called l-mer. To avoid a separate function for determining the length of a seed, we include this process as an initial step of the read hashing program, namely `hashing_read`. The program therefore takes two input parameters including the file containing the read sequence and the allowed edit distance.

Given the length of a seed determined by `hashing_read`, the genome hashing program, `hashing_genome`, also takes two parameters including the length of a seed and the file containing the genome sequence. As mentioned in the previous section, while we should perform the hashing for every read needed to be mapped, the hashing of the reference genome is seldom performed since a reference genome can be used for multiple reads. The genome hashing needs to redo only when the length of a seed changes due to the changes of the allowed edit distance or the length of the read.

Given all necessary parameters, the hashing process chops the read or reference genome into multiple identical-sized seeds and invokes `HMAC-SHA1` to produce the keyed hash values. As previously mentioned, the only difference between hashing of the reads and that of the reference genome is that the reads are not chopped into non-overlapping l-mers but every l-mer is extracted and mapped on the reference genome. If a sufficient number of l-mers map in a small segment of the reference genome, the hit is then chosen for the extension step [David et al., 2011]. This improvement significantly reduces the number of extensions performed on the users servers, thereby improving the overall performance.

6.6.2.2 Encryption of Seed Offsets and Computation of Shift Values with Homomorphic Encryption

In the seed hashing process, we need to store not only their keyed hash values but also their offsets, i.e., the position of the seeds on the read or the reference genome. As previously mentioned, the position of seeds is a sensitive information, which needs to be encrypted to securely protect its privacy. However, we also need the offsets for computation of the matching position of the read on the reference genome. We integrate `HElib` into the two hashing programs presented above to perform encryption of the offsets.

For the `HElib` context, mainly defined by the plaintext base (p), the security parameter (λ) and the circuit depth (D), we choose the following parameter values: $p = 547499$, $\lambda = 80$ and $D = 8$.

While we can use the default parameters set by `HElib` for λ and D, the plaintext base (p) needs to be defined by the developer since it represents the range of plaintext values, i.e., the offset values. Since the larger the p's value, the longer the execution time of operations performed on encrypted data as experimented in [Togan and Plesca, 2014], we set the value of $p = 547499$, the smallest prime number that is larger than the length of the longest reference genome in our experiment. It is noted that this value is large enough for representing the seed offsets of the reads, which usually consist of a few tens pairs of nucleotides. However, it may not be sufficient for the reference genome, which has a million pairs of nucleotides. To overcome this problem, one of solutions can be division of the reference genome into multiple short chunks. Two consecutive chunks should have an overlapping region whose size is larger than the length of reads to avoid the case that reads may be partially mapped on both chunks. Fig. 6.3 illustrates this solution where Fig. 6.3a presents the entire genome with a short read that best maps at the position of the dashed line. The division without overlapping in Fig. 6.3b makes the mapping failed. It is because the number of matching seeds of the left part of the read on Chunk 1 may not satisfy the threshold of q-gram filtering thus extension will not be executed. Fig. 6.3c shows the division with overlapping that results in a successful mapping.

Another challenging issue concerning the application of `HElib` is the size of ciphertexts. For each integer from 0 to 547499, the size of its ciphertext is roughly 2.1 MB. Storing all ciphertexts in memory (RAM) and sending them to the public cloud is not possible since the required amount of memory is too large even for a modern server. For instance, storing the offsets of 2000 seeds of the reference genome requires 6 GB memory. To solve this problem, we save ciphertexts into files stored on external diskspace. To avoid the overwrite of files, we use the keyed hash values of seeds as filenames. Since a seed can appear multiple times in a read or the reference genome, a file may contain multiple ciphertexts of all offsets of the seed. By saving ciphertexts into files, the hashing of the reference genome becomes a one-time-cost task, and the results can be transferred to the public cloud for future use.

Given that all encrypted data has been transferred to the public cloud where all seeds of the read are stored in a separate location with that of the reference genome, the seeding step is then securely performed. We implement `seeding` that searches for all matches between the seeds of the read and those of the reference genome by simply finding two files having the same name from the two locations. `seeding` also includes `HElib` to perform the computation of the shift value from two encrypted offset values. All encrypted shift values are also saved to files, which will be transferred back to the trusted server for re-encryption with OPE.

6.6.2.3 Sorting Shift Values Re-encrypted by OPE

Since we cannot perform the comparison with data encrypted by HE, all the shift values need to be re-encrypted with OPE that supports comparison over encrypted values. We implement `re_encrypt_ope` for the re-encryption purpose. For each shift value, `re_encrypt_ope` first decrypts the ciphertext using `HElib` and then re-encrypts by OPE. Since the shift values encrypted by OPE are also integer, we store all shift values into the same text file that will be sent to the public cloud for the next step. For the security parameters of OPE, the secret key is given by the user while the size of plaintext and ciphertext are set by the default values of OPE, 32 and 64 bits, respectively.

Given all the shift values encrypted by order-preserving encryption, the next step of the mapping process is to determine all possible extensions and the best matching positions of the read on the reference genome. To do so, we sort all the shift values in the ascending order and count for the number of seed matches that have the same shift values. The shift values of the possible extensions are then sent back to the trusted server for the extension step. The process is implemented in the program file `sorting`. For decrypting the shift values of the possible extensions, we implement a program namely `ope_decryption`, which takes a file as input containing all the ciphertexts encrypted by OPE and generates a file containing all the respective shift values in the plaintext format.

6.6.2.4 Extension

The extension step is performed on the trusted server with the plaintext data. The `extension` program takes three parameters: the file containing the read, the file containing the reference genome and the file containing all the shift values, which are the positions where the read aligns on the reference genome. The program first loads the nucleotide sequences from both the read and the reference genome to memory. For each shift value, the program extracts the segment on the reference genome and computes the edit distance between the read and the segment. We implement the dynamic algorithm to compute the Levenshtein distance [Schbath et al., 2012] between two sequences.

6.7 Performance Evaluation

6.7.1 Experiment Setup

6.7.1.1 System

We deployed the experimental system by using two computers playing the role of the public cloud and the trusted server, respectively. Both computers are Dell Optilex 990, Intel(R) Core(TM) Processor i7 − 2600, 3.40GHz, 8 GB RAM. The connection between the trusted server and the public cloud is established by the Internet backbone with the bidirectional speed of 17 Mbps with the distance of 3 kms. We carried out 20 runs for each experiment to obtain the average performance metric values.

6.7.1.2 Data

We utilized the public available dataset that contains the Feb. 2009 assembly of the human genome (hg19, Genome Reference Consortium Human Reference 37[3]). We randomly sampled 10 thousand reads that are 36 bp long from all chromosomes. Over this dataset, we ran our prototype allowing up to 3 mismatches to map all reads onto different reference genomes. Thus, all seeds are 9-mers. We measured the running time of the entire mapping process to evaluate the overall performance as well as the impact of the 3EGSM model. Due to the limited disk space on both computers, we could not run our prototype for all 22 available chromosomes. Instead, we selected several chromosomes with different size to show the change in performance of the proposed framework.

[3]Experiment Dataset: http://hgdownload.cse.ucsc.edu/goldenpath/hg19/chromosomes

Table 6.1: One-time Cost for Hashing Reference Genomes with 9-mers, i.e., allowing 3 mismatches.

Chromosome	Original Size (KB)	#Seeds	Required Space (MB)	Hashing Time (s)
chr18	8	474	969	83.149
chr21	28	3076	6284	493.193
chr19	160	17686	36124	2653.498
chr9	188	20782	42454	3048.999
chr1	548	60833	124258	9227.129
Total	**932**	**102851**	**210089**	**4:18:25**

6.7.2 Result Analysis

6.7.2.1 One-time Cost for Hashing Reference Genomes

In the first experiment, we evaluate the one-time cost for hashing reference genomes and encrypting the offsets of their seeds. For each reference genome, we measure the running time of the program to complete the hashing process and the size of disk space needed to store offset ciphertexts. In Table 6.1, we present the results of this experiment. It is obvious that the larger the size of the reference genome, the higher the number of seeds, therefore the larger the disk space needed to store all ciphertexts of offsets. With the largest reference genome whose size is 548 KB, the required disk space is 124258 MB, i.e., 121.35 GB. Even though the required disk space is quite large, the disk drive prices are really cheap nowadays. Furthermore, the hashing phase can be performed on the trusted server well ahead of the mapping on the public cloud, the keyed hash values and encrypted offsets can be moved to the public cloud storage space without suffering any security risk. The occupied disk space on the trusted server can be then freed for another task if needed.

We obtained the same behavior when considering the running time of the program for hashing the reference genomes. With the largest reference genome, the running time is 2 hours 33 mins 47 seconds. Fortunately, this is a one-time cost as a reference genome after being hashed can be used for multiple reads if the size of reads or the number of mismatches allowed do not change. It is noted that the running time includes the initialization of `HElib`, which is about 14.30 seconds. The main portion of the running time comes from the scanning throughout of the reference genome to hash and encrypt the offset for every seed. For the largest reference genome, the program needs to process 60833 seeds, corresponding to almost the same number of files opened on the disk. This prolongs the running time of the hashing program as the I/O operation is usually time consuming. Comparing with the running time without offset encryption, which is in the order of few minutes [Chen et al., 2012], the running time with offset encryption is much longer. Nevertheless, this one-time cost is completely affordable with regards to the achieved benefit of data privacy protection.

The last row of Table 6.1 shows the summary of the hashing and encrypting process to prepare the dataset. With 5 reference genomes with a total size of 932 KB, we obtained 102851 9-mers, which consume more than 205 GB of disk space to store their encrypted offsets and keyed hash values. The whole process takes more than 4 hours to complete.

6.7.2.2 Overall Performance

We now evaluate the overall performance of the framework by running the entire mapping process. Given a reference genome, we map all 10 thousand reads onto the reference genome one by one. We evaluate the performance of the prototype at a fine-grained level by measuring the execution time of each step: the hashing of read on the trusted server, seeding on the public cloud, re-encryption of shift values with OPE on the trusted server, sorting and determining the possible extension positions on the public cloud, and extension on the trusted server. For the reads of 36 bp corresponding to 28 overlapping 9-mers, the total hashing and encryption time of seed offsets is 18.504 seconds including 14.30 seconds spent for the initialization of `HElib`. This initialization is a one time cost, so it can be affordable when performing the hashing for thousand or even milion of reads.

In Fig. 6.4, we present the average processing time of each step of the mapping process over 10 thousand reads. It is observed that the longer the reference genome, the longer the time needed for each step in the mapping process, leading to the increase in the overall mapping time. The results show that the largest portion of the processing time of the mapping process comes from the seeding step. With the longest reference genome, the seeding spends 53.57% of the total mapping time, excluding the communication time to transfer seed offsets of the read and intermediate results between the trusted server and the public cloud. The second largest portion of the mapping time is the OPE encryption of the shift values performed on the trusted server. As the reference genome is longer, the number of occurrences of seeds on the reference genome increases, i.e., there is a higher probability that there are more matches of seeds on a longer reference genome. The time spent for sorting and determining the possible extension positions is less than one second. The extension time on the trusted sever is 0.004s per extension. With the q-gram filtering method, the number of extensions is reduced significantly, the longest extension time is only 0.015s. In total, the mapping time of a read on the longest reference genome is 75 seconds including the hashing time, the time of all steps in the mapping process and the data transmission time. Compared to a baseline approach [Atallah et al., 2003], which takes about 5 minutes to compute the edit distance between two 25 bp sequences through fully homomorphic encryption and oblivious transfers, the proposed framework reduces the processing time by up to 75%.

It is noted that while the seeding step is performed for all reads, the OPE re-encryption and sorting steps are performed only when there is at least one exact match of the seeds on the reference genome. The number of extensions is further reduced by the q-gram filtering method since the number of matches of several reads does not satisfy the threshold. Thus, with the proposed framework, we successfully delegated the most data-intensive task to the public cloud without any concern about the privacy of input data as well as computation results, which may be exploited by adversaries to infer useful information.

6.7.2.3 Communication Overhead

When running our prototype on the setup infrastructure, we realized that the communication overhead is significant in the overall performance. Due to the low connection bandwidth, i.e., relying on the Internet backbone, as well as the high volume of data transferred to the public cloud, the communication time increases significantly compared to the case of computation with plaintext data. Concerning the one-time cost of transferring the reference genome to the public cloud, while the non-secured approach needs

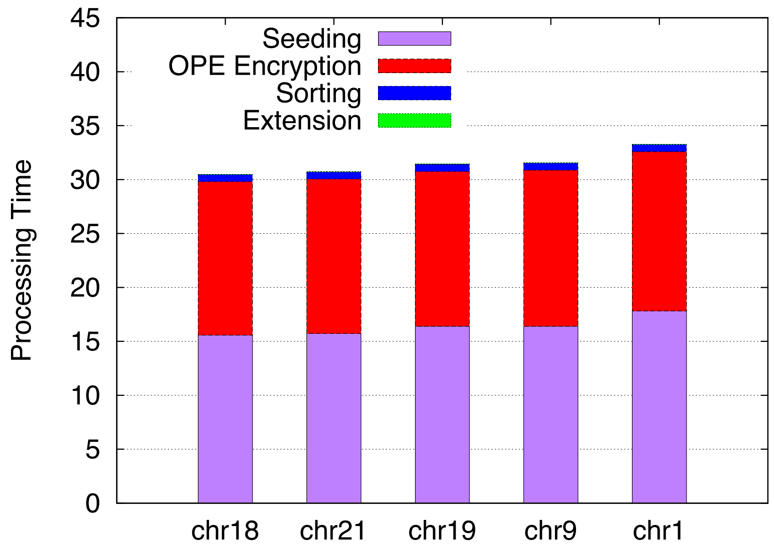


Figure 6.4: Average processing time (in second) of each step in mapping process over 10 thousand reads being mapped onto 5 genomes allowing 3 mismatches.

Table 6.2: One-time Cost for Hashing Reference Genomes with 12-mers, i.e., allowing 2 mismatches.

Chromosome	Original Size (KB)	#Seeds	Required Space (MB)	Hashing Time (s)
chr18	8	356	733	59.156
chr21	28	2307	4750	309.756
chr19	160	13265	27308	1750.376
chr9	188	15587	32089	2213.414
chr1	548	45625	93926	5929.416
Total	**932**	**77140**	**158806**	**2:51:02**

to transfer 5.7 GB of keyed hash values of the largest genome in our experiment [Chen et al., 2012], our proposed approach needs to transfer 121.35 GB. It took roughly 17 hours on the 17 Mbps link. Even though the amount of data is large, we believe that with a dedicated link with higher bandwidth, e.g., large than 100 Mbps, this one-time cost is completely affordable as in reality Amazon routinely receives terabytes of data through its Import/Export service [AWS, 2015].

For every read, we also need to transfer the keyed hash values and encrypted offsets of its seeds to the public cloud. With 36 bp reads generating 28 overlapping 9-mers, the amount of data being transferred to the public cloud is 58.8 MB. It took 17.168 seconds on average to complete the data transmission on the 17 Mbps link. The size of intermediate and final results is small so that it can be negligible compared to the overall processing time. It is also noted that the communication overhead could be amortized by asynchronous communication such that communication and computation are overlapping. In order to achieve this parallelism, implementing the read mapping application as a workflow-based application is one of solutions [Rojas Balderrama et al., 2012].

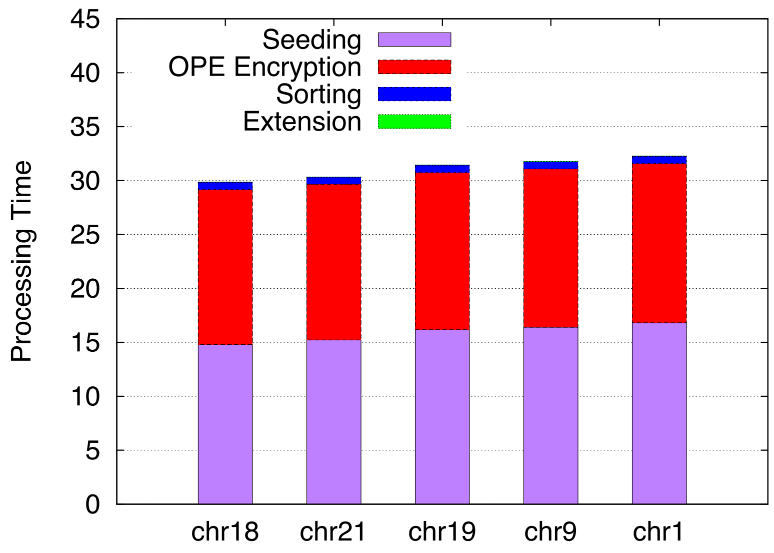


Figure 6.5: Average processing time (in second) of each step in mapping process over 10 thousand reads being mapped to 5 genomes allowing 2 mismatches.

6.7.2.4 Impact of Number of Mismatches

In this experiment, we set the number of mismatches allowed to map a read onto the reference genome to 2 and repeated the same experiments as presented in the previous sections. With 2 allowed mismatches, the length of seeds is 12 bp. Consequently, the processing time, the number of seeds of each read as well as reference genome, and the required disk space are reduced. In Table 6.2, we present the hashing time and required disk space when allowing 2 mismatches. The total hashing time is reduced by 1 hour 27 mins. And the total required disk space is now only 158806 MB, i.e., 155.08 GB. The average processing time of each step also slightly reduces as shown in Fig. 6.5. We did not perform the experiment when allowing more than 3 mismatches due to the limit of our experimental system. We believe that we will obtain the same behavior since increasing the number of allowed mismatches leads to the increase in the number of seeds, the total processing time and the disk space needed for the data.

6.7.2.5 Mapping Speedup with Parallel Execution

Sequential mapping of 10 thousand reads onto a reference genome turned out to be very slow even though the application of the 3EGSM model has significantly improved the performance of the mapping process. With the longest reference genome in our experiment, it takes more than 8 days to complete the task including the data transmission time. Since reads are independent of each other, we can accelerate the processing by concurrently running the mapping process on different machines, each performs the mapping for a portion of reads. We deployed multiple virtual machines on the physical server, i.e., from 2 to 5, and ran the mapping process on these virtual machines. Corresponding to the number of virtual machines, 10 thousand reads are divided evenly into the respective number of portions.

In Fig. 6.6, we present the mapping speedup achieved by the parallel execution. Theoretically, one can say that the speedup is the number of virtual machines used for the processing. In reality, since the trusted server has to communicate with multiple virtual machines at the same time, an overhead

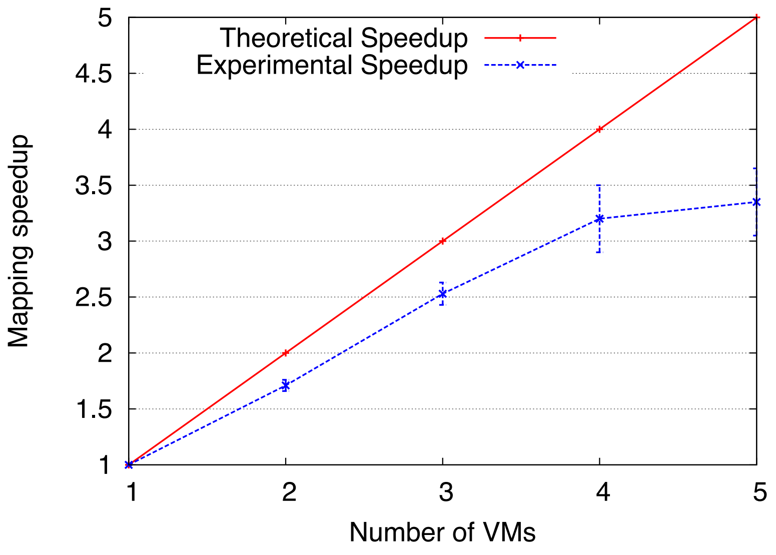


Figure 6.6: Mapping speedup with parallel execution.

therefore incurs. Furthermore, the data transmission becomes the bottleneck since the total bandwidth of the connection is shared among concurrent transmissions with virtual machines. Thus, the processing speedup can attain only 3.35 when running the mapping on 5 virtual machines. Compared to the speedup obtained by *CloudBurst* [Schatz, 2009], the speedup achieved by the proposed framework is lesser due to the communication overhead of the larger volume of data. Nevertheless, the results of this experiment show the effectiveness of the proposed framework to exploit the immense computing capacity of the public cloud at low resource usage cost. While CloudBurst does not provide the data privacy protection, the 3EGSM model used in the proposed framework guarantees that no sensitive information will be leaked when migrating the mapping process to public clouds. Due to the limitted of the experiment testbed, we did not run the experiments beyond 5 VMs. However, we expect that the speedup will increase a bit more and then keeping flat due to the concurrent data transmission from the trusted server to the computing nodes.

6.8 Chapter Summary

In this chapter, we designed an entire secure framework for genomic data processing leveraging on cloud resources to address the exponential growth of genomic data [Kang et al., 2016a]. Based on this framework, we proposed a 3-encryption-scheme model for genomic sequence mapping (3EGSM) to solve the security issue. The model uses keyed hash function for genomic sequence matching, homomorphic encryption for arithmetic computation and order-preserving encryption for numerical comparison. We concretized the framework and integrated the 3EGSM model into the seed-and-extend algorithm, resulting in a secure prototype for read mapping on public clouds. The results show that delegating the computation of genomic data to clouds can accelerate the processing depending on the scale of the computing platform deployed on clouds. The contributions of this work facilitate the exploitation of cloud resources, and protecting a data privacy not only for secure storage but also secure computations.

Chapter 7

Conclusions and future work

7.1 Conclusions

The work presented in this thesis addressed the problem of performance optimization and data privacy protection for large-scale (multi-cloud) storage and computing systems. Public clouds provide users a quasi-unlimited amount of computing, network an d storage resources for dealing with the needs of heavy and large-scale computation requirements of big data applications. However, moving data to public clouds for storing and processing raises new challenges, which are addressed in the thesis: (i) the performance issue, e.g., the delay in data retrieval or processing, and (ii) the security issue, e.g., the data privacy. Combining with complex requirements in resource requests and in the architecture of the systems, these challenges drive the needs for novel data management solutions that consider both the performance and data security issues. Focusing on load scheduling and data placement, the work carried out in this thesis provided the solutions that address the above challenges not only separately but also in a joint problem, i.e., optimization of performance while satisfying the security requirement.

Load scheduling strategies in distributed computing systems have been studied for handling large amount of data with heavy computation. Most of existing approaches do not consider the new features of multi-cloud systems such as dedicated network resource with specific network topology. Furthermore, they consider the sole load source that is not realistic in nowadays big data applications such as environment monitoring and health systems. Motivated by this, we proposed a novel architecture for multi-cloud systems that take into account the network topology, the link capacity, the heterogeneity of computing nodes in terms of computing capacity and release time. Depending on the size of the infrastructure based on the number of load sources, users may decide to use a centralized scheduler or multiple distributed schedulers. Leveraging on the divisible load paradigm and the phase-based scheduling approach, we proposed two scheduling strategies that will be run on the schedulers: a Static Scheduling Strategy (SSS) and a Dynamic Scheduling Strategy (DSS), whose objective is to minimize the total processing time of loads. While both SSS and DSS use a number of mathematical formulations to compute the available time duration of computing nodes to determine the chunk size appropriately, SSS assumes that the release time of computing nodes is known prior to the scheduling procedure. Given such known inputs, SSS repeatedly executes the phase-based scheduling algorithm to generate the scheduling decision for all the loads submitted to the system. The scheduling decision is stored as a lookup table and realized once the nodes are released at the known time. SSS may result in poor resource utilization if the release times of computing nodes are unknown since the nodes, which are released after the moment when scheduling decision have been made, will be wasted.

Towards a better resource utilization and better overall performance of the system, DSS relaxes the assumption of SSS and reacts immediately when a new computing node is released. Instead of generating the scheduling decision for all the loads well ahead to their actual processing, DSS gradually generates the scheduling solutions along processing phases and additional assigns loads to new released nodes even at the middle of the processing phase. Running on distributed schedulers, DSS needs to be aware of the ready time of computing nodes for receiving and processing a new load chunk since the nodes may be currently busy for other load chunks assigned by other schedulers. We proposed to apply a prediction technique on computing nodes, which will estimate the processing time of the load chunks that have been received and they will inform the schedulers during the scheduling process. We performed extensive simulations to evaluate the performance of the proposed scheduling strategies. We also used a real world log trace as the training dataset when evaluating DSS. The results show that the proposed scheduling strategy outperforms the baseline schemes which do not apply divisible load theory by reducing the total processing time of loads up to 44.60%. The results also provide useful insights on the applicability of the proposed architecture and scheduling strategy across a range of realistic systems.

While the SSS and DSS focus only on the performance optimization issue with assumption that data remains in computing nodes for a short duration during the processing, thus suffering lower security risk, on other extreme, users may want to strictly protect the data privacy when the data is stored in cloud storage systems for a long duration, e.g., for backup purpose. This motivates us to design and implement a server-side encryption service for cloud storage systems namely ESPRESSO to protect the data privacy, using advanced encryption algorithms. Not only protecting the privacy of the data, ESPRESSO also protects the privacy of the users' encryption with a secrete master key. Furthermore, ESPRESSO also guarantees the availability of the users' encryption keys by replication, thus ensuring that data can be decrypted whenever users require even in the case the primary key is not available. The proposed encryption service was integrated into two open-source cloud storage platforms: OpenStack/Swift and Nimbus/Cumulus as an additional service deployed in a separate server without breaking down their original code structure. The real experiments were conducted on both Swift and Cumulus storage systems, and the results show that the introduced encryption latency is negligible compared to the total operation time of a data management request even with a large data file of 4GB, thereby demonstrating the effectiveness of the proposed encryption service.

Combining the performance issue and the data privacy issue in a joint problem makes it more complex and hard to achieve. However, many realistic big data applications such as health care systems require both requirements since users would like not only to optimize the performance for paying less cloud resource cost but also to protect the data privacy as the data is very sensitive. Focusing on the data placement in cloud storage systems, we proposed a novel approach to protect the data privacy by using graph theory. The data is assumed to be divisible and the sensitive information is spread over multiple chunks such that a compromised chunk will not make sense to malicious users. We develop a novel data placement algorithm namely Availability and Security-awarE Data placement algorithm for cLOUd storage Systems (A-SEDuLOUS) that (i) minimizes the total retrieval time of a data by determining appropriate size of data chunks that will be stored on respective nodes, (ii) protects the data privacy by selecting the storage nodes that satisfy the security requirement represented by the distance on the network path between two nodes storing the chunks of the same data, and (iii) guarantees the data availability regardless of attacks on nodes or links by replicating data chunks on appropriate nodes. We showed the effectiveness of the proposed algorithm against baseline algorithms by extensive simulations on different cloud storage systems using random network topologies and the realistic Internet2 topology. The proposed algorithm reduces the retrieval time by up to 34% for random topologies and 19% for the Internet2 topology. The results also show that while achieving the best performance and guaranteeing the data privacy and availability, the proposed algorithm does not sacrifice other performance metrics such as the rejection ratio of storage requests.

While A-SEDuLOUS was evaluated through numerical simulations, we take another step towards demonstrating the feasibility of the proposed approaches in this thesis by a practical case study. We considered a genomic application that represents a big data application, requiring not only to process a large amount of data with heavy computation but also high level of security for genomic data. We designed an entire secure framework for genomic computation on public clouds to exploit the parallel processing on multiple computing nodes so as to reduce the total processing time of the application. We concretized the framework and proposed a 3-encryption-scheme model for genomic sequence mapping (3EGSM) by combining key-hash function, homomorphic encryption and order-preserving encryption to protect not only genomic sequences but also the intermediate and final computation results. The combination of the above encryption schemes and our adoption into genomic computation significantly eliminates the performance degradation from fully homomorphic encryption by using key-hash function and order-preserving encryption for the data that does not involve in the arithmetic operations. The performance of the proposed framework was evaluated through intensive experiments using real genomic data. Compared to a baseline approach that only uses homomorphic encryption and oblivious transfers, the proposed framework reduces the total processing time by up to 75%. The results also show that delegating the computation of genomic data to public clouds can accelerate the processing, achieving a speedup by 3.35 with only 5 virtual machines compared to sequential processing on a dedicated machine. The experimental results show that while existing approaches have to sacrifice one for the other, our proposed approach achieves all the three goals above.

7.2 Future Work

Performance optimization for practical genomic computation. The approach presented in Chapter 6 was evaluated by real experiments using real genome data. Even though we have implemented a prototype and carried out the experiments to demonstrate the effectiveness of the proposed approach, the prototype framework was evaluated only for a specific genomic application, genomic sequence mapping. It would be an interesting research direction to generalize and implement this framework for all genomic applications, making the framework become a completed healthcare systems. Furthermore, the real experiments of this work also showed that the communication overhead is significant and degrades the overall performance of the system, i.e., prolonging the total processing time. This drives the needs for novel approaches to improve the performance of the system, taking into account the communication overhead. Based on the observation that many large-scale and data-intensive applications usually compose of multiple sub-tasks that may be executed in sequential or parallel, we can leverage on the workflow execution approach [Rojas Balderrama et al., 2012] to exploit all the parallelism levels supported by the workflow engines such as data parallelism, task parallelism or pipelining. Data parallelism allows the applications to process multiple data items in parallel, e.g., the genomic sequence mapping can process multiple sequences at the same time on different computing nodes. Task parallel or pipelining represents the concurrent execution of two independent input data items by two different tasks that are sequentially connected. For instance, in the genomic sequence mapping with the seed-and-extend algorithm presented in Chapter 6, the seeding step of a sequence can be executed in parallel with the extension step of another sequence. With the parallel execution, the communication overhead of a data item can be amortized by the actual processing of other data items.

Towards practical secure computation with fully homomorphic encryption. On the research direction for protection of data privacy, this thesis focus on exploiting public clouds for improving the overall performance of the systems with fully homomorphic encryption (FHE). The parallel computation on multiple computing nodes significantly reduces the total processing time of the applications. However, it still dominate the performance of the system when processing with plaintext data even though users gain the data privacy protection. Thus, bringing the proposed approach to practical computation at the production level is still far and needs a lot of research effort. Instead of using traditional computing units provided by public clouds, users can seek to use advance computing system to accelerate the processing speed. On one hand, users may use commodity high performance GPUs to build up high performance FHE engine and the proposed framework will be making use of it to improve the performance. On the other hand, the development of recent instruction set extension for Intel CPUs called SGX allows us to perform privacy-preserving computations in an untrusted environment while minimizing the effort required for algorithm adoption. SGX encrypts the data in memory, protects the data integrity, and ensures that the cloud provider/ OS cannot look into it. We intend to investigate the security properties of this new hardware and propose security policies according to users requirement.

7.3 List of Publications from this Thesis

7.3.1 Published Papers

1. **Kang, S.**, Aung, K. M. M., and Veeravalli, B. Towards Secure and Fast Mapping of Genomic Sequences on Public Clouds. In *Fourth International Workshop on Security in Cloud Computing (SCC)*, Xian, China, May. 2016.

2. **Kang, S.**, Veeravalli, B., and Aung, K. M. M. A Security-aware Data Placement Mechanism for Big Data Cloud Storage Systems. In *IEEE International Conference on Intelligent Data and Security (IEEE IDS 2016)*, New York, USA, Apr. 2016.

3. **Kang, S.**, Veeravalli, B., and Aung, K. M. M. Scheduling Multiple Di- visible Loads in a Multi-cloud System. In *7th ACM/IEEE International Conference on Utility and Cloud Computing (IEEE/ACM UCC 2014)*, pages 371-378, London, UK, Dec. 2014.

4. **Kang, S.**, Veeravalli, B., Aung, K. M. M., and Jin, C. An Efficient Scheme to Ensure Data Availability for a Cloud Service Provider. In *IEEE International Conference on Big Data (IEEE Big Data 2014)*, pages 15-20, Washington, DC, Oct. 2014.

5. **Kang, S.**, Veeravalli, B., and Aung, K. M. M. ESPRESSO: An Encryption as a Service for Cloud Storage Systems. In *8th International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS 2014)*, pages 15-28, Brno, Czech Republic, Jul. 2014.

7.3.2 Under Review Papers

1. **Kang, S.**, Veeravalli, B., and Aung, K. M. M. On the Design of a Dynamic Scheduling Strategy with Efficient Node Availability Prediction for Handling Divisible Loads in Multi-Cloud Systems. *IEEE Transactions on Cloud Computing*, 2016.

2. **Kang, S.**, Veeravalli, B., and Aung, K. M. M. On the Design of a Security-aware Data Placement Algorithm for Data Privacy and Availability in Cloud Storage Systems. *IEEE Transactions on Computers*, 2016.

Bibliography

[Adhikari et al., 2012] Adhikari, V., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L. (2012). Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *IEEE INFOCOM 2012*, pages 1620–1628, Orlando, USA.

[Agrawal et al., 2004] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2004). Order Preserving Encryption for Numeric Data. In *ACM ICMD 2004*, pages 563–574, Paris, France.

[Albanesius, 2012] Albanesius, C. (2012). Service Disruption Hits Google's Gmail, Chrome. http://www.pcmag.com/article2/0,2817,2413035,00.asp. [Online; accessed 20-Nov-2015].

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

[Amazon S3, 2014] Amazon S3 (2014). Amazon simple storage system.

[Atallah et al., 2003] Atallah, M. J., Kerschbaum, F., and Du, W. (2003). Secure and Private Sequence Comparison. In *ACM WPES 2003*, pages 39–44, Washington, DC, USA.

[AWS, 2015] AWS (2015). AWS Import/Export. http://aws.amazon.com/importexport. [Online; accessed 20-Nov-2015].

[Baev et al., 2008] Baev, I., Rajaraman, R., and Swamy, C. (2008). Approximation Algorithms for Data Placement Problems. *SIAM Journal on Computing*, 38(4):1411–1429.

[Baeza-yates and Perleberg, 1992] Baeza-yates, R. A. and Perleberg, C. H. (1992). Fast and Practical Approximate String Matching. In *CPM 1992*, pages 185–192, Tucson, USA.

[Bakhtiari et al., 1995] Bakhtiari, S., Safavi-Naini, R., and Pieprzyk, J. (1995). Cryptographic hash functions: A survey. Technical report, University of Wollongong.

[Bellare et al., 1996] Bellare, M., Canetti, R., and Krawczyk, H. (1996). Keying Hash Functions for Message Authentication. In *CRYPTO'96*, pages 1–15, Santa Barbara.

[Blaze et al., 1998] Blaze, M., Bleumer, G., and Strauss, M. (1998). Divertible Protocols and Atomic Proxy Cryptography. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 1998)*, pages 127–144, Espoo, Finland.

[Boldyreva et al., 2009] Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. (2009). Order-preserving Symmetric Encryption. In *EUROCRYPT 2009*, pages 224–241, Cologne, Germany.

[Boldyreva et al., 2011] Boldyreva, A., Chenette, N., and O'Neill, A. (2011). Order-preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO'11*, pages 578–595, Santa Barbara.

[boto, 2014] boto (2014). In *http://code.google.com/p/boto*.

[Brakerski et al., 2014] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (Leveled) Fully Homomorphic Encryption Without Bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13.

[Braun et al., 1999] Braun, T. D., Siegel, H. J., Beck, N., Boloni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (1999). A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems. In *8th HCW*, pages 15–29, San Juan.

[Bresnahan et al., 2011] Bresnahan, J., Keahey, K., LaBissoniere, D., and Freeman, T. (2011). Cumulus: An Open Source Storage Cloud for Science. In *ScienceCloud'11*, pages 25–32, CA.

[Bruekers et al., 2008] Bruekers, F., Katzenbeisser, S., Kursawe, K., and Tuyls, P. (2008). Privacy-Preserving Matching of DNA Profiles.

[Cachin et al., 2009] Cachin, C., Keidar, I., and Shraer, A. (2009). Trusting the Cloud. *ACM SIGACT News*, 40(2):81–86.

[Chen et al., 2012] Chen, Y., Peng, B., Wang, X., and Tang, H. (2012). Large-scale Privacy-preserving Mapping of Human Genomic Sequences on Hybrid Clouds. In *NDSS 2012*, San Diego, California.

[Cheng and Robertazzi, 1988] Cheng, Y. C. and Robertazzi, T. G. (1988). Distributed Computation with Communication Delays. *IEEE Trans. Aerosp. Electron. Syst.*, 24(6):700–712.

[Chiesa et al., 2015] Chiesa, A., Tromer, E., and Virza, M. (2015). Cluster computing in zero knowledge. In *EUROCRYPT 2015*, pages 371–403, Sofia, Bulgaria.

[Chung et al., 2010] Chung, K.-M., Kalai, Y., and Vadhan, S. (2010). Improved Delegation of Computation Using Fully Homomorphic Encryption. In *CRYPTO 2010*, pages 483–501, Santa Barbara, USA.

[Ciancutti, 2010] Ciancutti, J. (2010). Four Reasons We Choose Amazons Cloud as Our Computing Platform. Technical Report The Netflix Tech Blog, Netflix.

[cURL, 2014] cURL (2014). cURL:A tool and a library for client-side URL transfers.

[David et al., 2011] David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. (2011). SHRiMP2: Sensitive yet Practical Short Read Mapping. *Bioinformatics*, 27(7):1011–1012.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 10–10, San Francisco, CA.

[Dignan, 2008] Dignan, L. (2008). In *http://www.zdnet.com/article/amazon-explains-its-s3-outage/*.

[Drozdowski and Lawenda, 2008] Drozdowski, M. and Lawenda, M. (2008). Scheduling multiple divisible loads in homogeneous star systems. *Journal of Scheduling*, 11(5):347–356.

[ER, 2008] ER, M. (2008). Next-generation DNA sequencing methods. *Annual Review of Genomics and Human Genetics*, 9:387–402.

[Factor et al., 2013] Factor, M., Hadas, D., Hamama, A., Har'el, N., Kolodner, E., Kurmus, A., Shulman-Peleg, A., and Sorniotti, A. (2013). Secure logical isolation for multi-tenancy in cloud storage. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, Long Beach, CA.

[Fang et al., 2010] Fang, Y., Wang, F., and Ge, J. (2010). A Task Scheduling Algorithm Based on Load Balancing in Cloud Computing. In *2010 International Conference on Web Information Systems and Mining (WISM 2010)*, pages 271–277, Sanya, China.

[Fienberg et al., 2011] Fienberg, S. E., Slavkovic, A. B., and Uhler, C. (2011). Privacy Preserving GWAS Data Sharing. In *IEEE ICDMW 2011*, pages 628–635, Vancouver, Canada.

[Fontaine and Galand, 2007] Fontaine, C. and Galand, F. (2007). A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007(15):1–15.

[Gentry, 2009] Gentry, C. (2009). Fully Homomorphic Encryption Using Ideal Lattices. In *ACM STOC 2009*, pages 169–178, Bethesda, USA.

[GoGrid, 2014] GoGrid (2014). GoGrid: Best of Cloud + Best of Hosting.

[Goldberg and Harrelson, 2005] Goldberg, A. V. and Harrelson, C. (2005). Computing the Shortest Path: A Search Meets Graph Theory. In *ACM-SIAM SODA 2005*, pages 156–165, Vancouver, Canada.

[Google, 2014] Google (2014). Google Cloud Storage server-side encryption.

[Goyal et al., 2006] Goyal, V., Pandey, O., Sahai, A., and Waters, B. (2006). Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 89–98, Alexandria, USA.

[Gu and Grossman, 2009] Gu, Y. and Grossman, R. L. (2009). Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. *Philosophical Transactions of The Royal Society A: Mathematical Physical and Engineering Sciences*, 367(1897):2429–2445.

[Guo et al., 2012] Guo, J., Zhu, Z.-M., Zhou, X.-M., and Zhang, G.-X. (2012). An instances placement algorithm based on disk I/O load for big data in private cloud. In *International Conference on Wavelet Active Media Technology and Information Processing*, pages 287–290, Chengdu, China.

[Hale, 1980] Hale, W. (1980). Frequency assignment: Theory and applications. *Proceedings of the IEEE*, 68(12):1497–1514.

[Hao and Han, 2011] Hao, L. and Han, D. (2011). The study and design on secure-cloud storage system. In *2011 International Conference on Electrical and Control Engineering (ICECE)*, pages 5126–5129, Yichang, China.

[Harrin, 2012] Harrin, E. (2012). Cloud Storage Vendors Offering Encryption as a Service. Technical report, Enterprise Networking Planet.

[Hashem et al., 2015] Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., and Ullah Khan, S. (2015). The rise of ”big data” on cloud computing: Review and open research issues. *Information Systems*, 47:98–115.

[Helft, 2009] Helft, M. (2009). Google Confirms Problems With Reaching Its Services.

[Hu and Veeravalli, 2011] Hu, M. and Veeravalli, B. (2011). Requirement-Aware Strategies with Arbitrary Processor Release Times for Scheduling Multiple Divisible Loads. *IEEE Trans. Parallel Distrib. Syst.*, 22(10):1697–1704.

[Huang et al., 2011a] Huang, Y., Evans, D., Katz, J., and Malka, L. (2011a). Faster Secure Two-party Computation Using Garbled Circuits. In *SEC*, page 35, San Francisco, CA.

[Huang et al., 2011b] Huang, Z., Li, Q., Zheng, D., Chen, K., and Li, X. (2011b). YI Cloud: Improving User Privacy with Secret Key Recovery in Cloud Storage. In *2011 IEEE 6th International Symposium on Service Oriented System Engineering (SOSE)*, pages 268–272, Irvine, CA.

[IMEX, 2010] IMEX (2010). The Promise & Challenges of Cloud Storage. Technical report, IMEX Research.

[Internet2, 2016] Internet2 (2016). Internet2 Open Science, Scholarship, and Services Exchange. `http://www.internet2.edu/products-services/advanced-networking/layer-2-services`. [Online: accessed 20-Jan-2016].

[Islam et al., 2012] Islam, S., Keung, J., Lee, K., and Liu, A. (2012). Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155 – 162.

[Itani et al., 2009] Itani, W., Kayssi, A., and Chehab, A. (2009). Privacy as a Service: Privacy-aware Data Storage and Processing in Cloud Computing Architectures. In *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'09)*, pages 711–716, Chengdu, China.

[jets3t, 2014] jets3t (2014). In *http://jets3t.s3.amazonaws.com*.

[Jha et al., 2008] Jha, S., Louis, K., and Shmatikov, V. (2008). Towards Practical Privacy for Genomic Computation. In *IEEE SP 2008*, pages 216–230, Oakland, CA, USA.

[Ji et al., 2012] Ji, C., Li, Y., Qiu, W., Awada, U., and Li, K. (2012). Big Data Processing in Cloud Computing Environments. In *ISPAN*, pages 17–23, San Marcos, Texas.

[Jiang et al., 2011] Jiang, Y., Perng, C.-S., Li, T., and Chang, R. (2011). ASAP: A Self-Adaptive Prediction System for Instant Cloud Resource Demand Provisioning. In *IEEE ICDM 2011*, pages 1104–1109, Vancouver, Canada.

[Jung et al., 2012] Jung, G., Gnanasambandam, N., and Mukherjee, T. (2012). Synchronous Parallel Processing of Big-Data Analytics Services to Optimize Performance in Federated Clouds. In *IEEE 5th International Conference on Cloud Computing*, pages 811–818, Honolulu, USA.

[Kamara and Lauter, 2010] Kamara, S. and Lauter, K. (2010). Cryptographic Cloud Storage. In *FC'10*, pages 136–149, Tenerife, Canary Islands, Spain.

[Kang and Zhang, 2010] Kang, L. and Zhang, X. (2010). Identity-based Authentication in Cloud Storage Sharing. In *MINES 2010*, pages 851 – 855, Nanjing, China.

[Kang et al., 2016a] Kang, S., Aung, K. M. M., and Veeravalli, B. (2016a). Towards Secure and Fast Mapping of Genomic Sequences on Public Clouds. In *Fourth International Workshop on Security in Cloud Computing (SCC)*, Xian, China.

[Kang et al., 2014a] Kang, S., Veeravalli, B., and Aung, K. M. M. (2014a). ESPRESSO: An Encryption as a Service for Cloud Storage Systems. In *8th International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS 2014)*, pages 15–28, Brno, Czech Republic.

[Kang et al., 2014b] Kang, S., Veeravalli, B., and Aung, K. M. M. (2014b). Scheduling Multiple Divisible Loads in a Multi-cloud System. In *7th ACM/IEEE International Conference on Utility and Cloud Computing (IEEE/ACM UCC 2014)*, pages 371–378, London, UK.

[Kang et al., 2016b] Kang, S., Veeravalli, B., and Aung, K. M. M. (2016b). A Security-aware Data Placement Mechanism for Big Data Cloud Storage Systems. In *IEEE International Conference on Intelligent Data and Security (IEEE IDS 2016)*, New York, USA.

[Kang et al., 2016c] Kang, S., Veeravalli, B., and Aung, K. M. M. (2016c). On the Design of a Dynamic Scheduling Strategy with Efficient Node Availability Prediction for Handling Divisible Loads in Multi-Cloud Systems. *IEEE Transactions on Cloud Computing*. under review.

[Kang et al., 2016d] Kang, S., Veeravalli, B., and Aung, K. M. M. (2016d). On the Design of a Security-aware Data Placement Algorithm for Data Privacy and Availability in Cloud Storage Systems. *IEEE Transactions on Computers*. under review.

[Kang et al., 2014c] Kang, S., Veeravalli, B., Aung, K. M. M., and Jin, C. (2014c). An Efficient Scheme to Ensure Data Availability for a Cloud Service Provider. In *IEEE International Conference on Big Data (IEEE Big Data 2014)*, pages 15–20, Washington, DC.

[Kent, 2002] Kent, W. J. (2002). BLAT: The BLAST-like Alignment Tool. *Genome Research*, 12(4):656–664.

[Kulynych and Korn, 2003] Kulynych, J. and Korn, D. (2003). The New HIPAA (Health Insurance Portability and Accountability Act of 1996) Medical Privacy Rule. Technical report, American Heart Association.

[Lauter et al., 2014] Lauter, K., Lopez-Alt, A., and Naehrig, M. (2014). Private Computation on Encrypted Genomic Data. Technical Report MSR-TR-2014-93, Microsoft Research.

[Li et al., 2009] Li, R., Yu, C., Li, Y., Lam, T. W., Yiu, S.-M., Kristiansen, K., and Jun, W. (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967.

[Lin et al., 2007] Lin, X., Lu, Y., Deogun, J., and Goddard, S. (2007). Real-Time Divisible Load Scheduling with Different Processor Available Times. In *ICPP 2007*, Xian, China.

[Litzenberger, 2014] Litzenberger, D. C. (2014). PyCrypto. In *https://www.dlitz.net/software/pycrypto*.

[Maheswaran et al., 1999] Maheswaran, M., Alib, S., Siegel, H. J., Hensgen, D., and Freund, R. F. (1999). Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131.

[Marchal et al., 2006] Marchal, L., Yang, Y., Casanova, H., and Robert, Y. (2006). Steady-State Scheduling of Multiple Divisible Load Applications on Wide-Area Distributed Computing Platforms. *International Journal of High Performance Computing Applications*, 20(3):365–381.

[Menezes et al., 1996] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. (1996). *Handbook of Applied Cryptography*. CRC Press.

[Microsoft Azure, 2014] Microsoft Azure (2014). In *http://www.windowsazure.com/en-us/*.

[Mo and Wang, 2012] Mo, X. and Wang, H. (2012). Asynchronous Index Strategy for high performance real-time big data stream storage. In *3rd IEEE International Conference on Network Infrastructure and Digital Content*, pages 232–236, Beijing, China.

[Moreno-Vozmediano et al., 2013] Moreno-Vozmediano, R., Montero, R., and Llorente, I. (2013). Key Challenges in Cloud Computing: Enabling the Future Internet of Services. *IEEE Internet Computing*, 17(4):18–25.

[Murali Mohan et al., 2015] Murali Mohan, P., Truong-Huu, T., and Gurusamy, M. (2015). TCAM-aware Local Rerouting for Fast and Efficient Failure Recovery in Software Defined Networks. In *IEEE GLOBECOM 2015*, pages 1–6, San Diego, CA.

[Naehrig et al., 2011] Naehrig, M., Lauter, K., and Vaikuntanathan, V. (2011). Can Homomorphic Encryption be Practical? In *3rd ACM Workshop on Cloud Computing Security*, pages 113–124, Chicago, USA.

[Network and et al., 2012] Network, C. G. A. and et al. (2012). Comprehensive molecular portraits of human breast tumours. *Nature*, 490(7418):61–70.

[Ning et al., 2001] Ning, Z., Cox, A. J., and Mullikin, J. C. (2001). SSAHA: A fast search method for large DNA databases. *Genome Research*, 11(10):1725–1729.

[Nuaimi et al., 2012] Nuaimi, K., Mohamed, N., Nuaimi, M., and Al-Jaroodi, J. (2012). A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms. In *IEEE Second Symposium on Network Cloud Computing and Applications (NCCA 2012)*, pages 137–142, London, UK.

[OpenStack, 2014] OpenStack (2014). In *http://swift.openstack.org/.*

[Pardalos et al., 1999] Pardalos, P. M., Mavridou, T., and Xue, J. (1999). The Graph Coloring Problem: A Bibliographic Survey. In Du, D.-Z. and Pardalos, P. M., editors, *Handbook of Combinatorial Optimization*, pages 1077–1141. Springer US.

[Popa et al., 2013] Popa, R. A., Li, F. H., and Zeldovich, N. (2013). An Ideal-Security Protocol for Order-Preserving Encoding. In *IEEE S&P 2013*, pages 463–477, San Francisco.

[Popa et al., 2011] Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H. (2011). CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *23rd ACM Symposium on Operating Systems Principles*, pages 85–100, Cascais, Portugal.

[RackSpace, 2014] RackSpace (2014). Rackspace: Leverage our cloud expertise to run fast and lean.

[Regev, 2005] Regev, O. (2005). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *ACM STOC 2005*, pages 84–93, Baltimore, MD, USA.

[Reiss et al., 2011] Reiss, C., Wilkes, J., and Hellerstein, J. L. (2011). Google cluster-usage traces: format + schema. Technical report, Google Inc.

[Rilak et al., 2014] Rilak, Z., Wernicke, S., and Bogicevic, I. (2014). Keeping Genomic Data Safe on the Cloud. *Journal of Biomolecular Techniques*, 25(Suppl):S5.

[Robinson et al., 2009] Robinson, N., Graux, H., Botterman, M., and Valeri, L. (2009). Review of the European Data Protection Directive. Technical Report TR710, RAND Coporation.

[Rojas Balderrama et al., 2012] Rojas Balderrama, J., Truong-Huu, T., and Montagnat, J. (2012). Scalable and Resilient Workflow Executions on Production Distributed Computing Infrastructures. In *ISPDC 2012*, pages 119–126, Germany.

[s3cmd, 2014] s3cmd (2014). In *http://s3tools.org/s3cmd.*

[Schatz, 2009] Schatz, M. C. (2009). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369.

[Schbath et al., 2012] Schbath, S., Martin, V., Zytnicki, M., Fayolle, J., Loux, V., and Gibrat, J.-F. (2012). Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis. *J Comput Biol.*, 19(6):796–813.

[Schuster, 2007] Schuster, S. C. (2007). Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18.

[Sen, 2013] Sen, J. (2013). Security and Privacy Issues in Cloud Computing. In *Architectures and Protocols for Secure Information Technology Infrastructures*, chapter 1, pages 1–42. IGI Global.

[Shaffer, 2007] Shaffer, C. (2007). Next-generation sequencing outpaces expectations. *Nature Biotechnology*, 25(2):149.

[Shamir, 1979] Shamir, A. (1979). How to Share a Secret. *Commun. ACM*, 22(11):612–613.

[Shoup and Halevi, 2012] Shoup, V. and Halevi, S. (2012). Design and Implementation of a Homomorphic-Encryption Library. Technical report, IBM Research.

[Smith et al., 2008] Smith, A. D., Xuan, Z., and Zhang, M. Q. (2008). Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, 9:128.

[Subashini and Kavitha, 2011] Subashini, S. and Kavitha, V. (2011). Review: A Survey on Security Issues in Service Delivery Models of Cloud Computing. *J. Netw. Comput. Appl.*, 34(1):1–11.

[Taylor et al., 2001] Taylor, J., Choi, E., Foster, C., and Chanock, S. (2001). Using genetic variation to study human disease. *Trends Mol Med*, 7(11):507–512.

[Tian et al., 2010] Tian, L.-Q., Lin, C., and Ni, Y. (2010). Evaluation of User Behavior Trust in Cloud Computing. In *ICCASM 2010*, pages 567–572, Taiyuan.

[Togan and Plesca, 2014] Togan, M. and Plesca, C. (2014). Comparison-based Computations Over Fully Homomorphic Encrypted Data. In *COMM 2014*, pages 1–6, Bucharest, Romania.

[Trapnell and Salzberg, 2009] Trapnell, C. and Salzberg, S. L. (2009). How to map billions of short reads onto genomes. *Nature Biotechnology*, 27(5):455–457.

[Truong-Huu and Tham, 2014] Truong-Huu, T. and Tham, C.-K. (2014). A Novel Model for Competition and Cooperation among Cloud Providers. *IEEE Trans. Cloud Comput.*, 2(3):251–265.

[van Dijk et al., 2010] van Dijk, M., Gentry, C., Halevi, S., and Vaikuntanathan, V. (2010). Fully Homomorphic Encryption over the Integers. In Gilbert, H., editor, *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer Berlin Heidelberg.

[Veeravalli and Wong, 2004] Veeravalli, B. and Wong, H. M. (2004). Scheduling Divisible Loads on Heterogeneous Linear Daisy Chain Networks with Arbitrary Processor Release Times. *IEEE Trans. Parallel Distrib. Syst.*, 15(3):273–288.

[Verma et al., 2013] Verma, M., Gangadharan, G., Ravi, V., and Narendra, N. (2013). Resource Demand Prediction in Multi-Tenant Service Clouds. In *IEEE CCEM 2013*, pages 1–8, Bangalore, India.

[Waddell et al., 2015] Waddell et al., N. (2015). Whole genomes redefine the mutational landscape of pancreatic cancer. *Nature*, 518(7540):495–501.

[Wang et al., 2010] Wang, L., von Laszewski, G., Younge, A. J., He, X., Kunze, M., Tao, J., and Fu, C. (2010). Cloud Computing: a Perspective Study. *New Generation Computing*, 28(2):137–146.

[Wang et al., 2009] Wang, R., Wang, X., Li, Z., Tang, H., Reiter, M. K., and Dong, Z. (2009). Privacy-preserving Genomic Computation Through Program Specialization. In *ACM CCS 2009*, pages 338–347, Chicago, USA.

[Wikipedia, 2014] Wikipedia (2014). Wikipedia archive. In *http://dumps.wikipedia.org*.

[Wingfield, 2009] Wingfield, N. (2009). Microsoft, T-Mobile Stumble With Sidekick Glitch. *The Wall Street Journal*.

[Yang et al., 2013] Yang, Y., Zhou, Y., Sun, Z., and Cruickshank, H. (2013). Heuristic Scheduling Algorithms for Allocation of Virtualized Network and Computing Resources. *Journal of Software Engineering and Applications*, 6(1):1–13.

[Yu et al., 2010] Yu, S., Wang, C., Ren, K., and Lou, W. (2010). Achieving secure, scalable, and fine-grained data access control in cloud computing. In *IEEE INFOCOM 2010*, pages 1–9, San Diego, CA.

[Yum et al., 2011] Yum, D. H., Kim, D. S., Kim, J. S., Lee, P. J., and Hong, S. J. (2011). Order-preserving Encryption for Non-uniformly Distributed Plaintexts. In *WISA 2011*, pages 84–97, Jeju Island, Korea.

[Zeng et al., 2009] Zeng, W., Zhao, Y., Ou, K., and Song, W. (2009). Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 1044–1048, Seoul, Korea.

[Zeng et al., 2016] Zeng, Z., Truong-Huu, T., Veeravalli, B., and Tham, C.-K. (2016). Operational Cost-aware Resource Provisioning for Continuous Write Applications in Cloud-of-Clouds. *Cluster Computing*.

[Zhang et al., 2012] Zhang, G., Li, C., Zhang, Y., Xing, C., and Yang, J. (2012). An efficient massive data processing model in the cloud - a preliminary report. In *Seventh ChinaGrid Annual Conference*, pages 148–155, Beijing, China.

[Zhang et al., 2011] Zhang, Q., Zhu, Q., and R., B. (2011). Dynamic Resource Allocation for Spot Markets in Cloud Computing Environments. In *4th IEEE International Conference on Utility and Cloud Computing (IEEE UCC 2011)*, pages 178–185, Melbourne, Australia.

[Zhao et al., 2010] Zhao, G., Rong, C., Li, J., Zhang, F., and Tang, Y. (2010). Trusted Data Sharing over Untrusted Cloud Storage Providers. In *IEEE CloudCom 2010*, pages 97–103, Indianapolis, IN.

[Zhou et al., 2011] Zhou, X., Peng, B., Li, Y. F., Chen, Y., Tang, H., and Wang, X. (2011). To Release or Not to Release: Evaluating Information Leaks in Aggregate Human-genome Data. In *ESORICS 2011*, pages 607–627, Leuven, Belgium.