View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Boston University Institutional Repository (OpenBU)

Boston University

OpenBU | **http://open.bu.edu**

Theses & Dissertations | Boston University Theses & Dissertations

2016

Privacy-preserving queries on encrypted databases

https://hdl.handle.net/2144/19739

Boston University

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

PRIVACY-PRESERVING QUERIES ON ENCRYPTED DATABASES

by

XIANRUI MENG

B.S., Bloomsburg University of Pennsylvania, 2010
M.S., Boston University, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2016

© Copyright by
XIANRUI MENG
2016

Approved by

First Reader ______________________________

George Kollios, Ph.D.
Professor of Computer Science

Second Reader ______________________________

Seny Kamara, Ph.D.
Associate Professor of Computer Science
Brown University, Computer Science Department

Third Reader ______________________________

Steven Homer, Ph.D.
Professor of Computer Science

To my lovely grandmother

Acknowledgments

Getting a Ph.D. is a wonderful occasion to thank the numerous people who have taught, helped, supported, and inspired me. It is, of course, impossible to thank every one of them, and I am sure I am inadvertently leaving someone out, for which I apologize in advance.

It was my pleasure and an honor to work with George Kollios these past years. As an advisor, he patiently and skillfully guided me, always recommending the relevant papers to read and the appropriate problems on which to focus. He has always given me the freedom to pursue my own ideas, all the while making sure I stayed productive. I will unquestionably miss those late nights that we were working on the research papers.

I am very grateful to my mentor and friend Seny Kamara, who has kindly agreed to be a reader for this thesis. I still remembered the day when Seny and I were sitting in a caf shop, trying to figure out a research project, which eventually leaded to a great paper. At Microsoft Research, I missed the days that we were watching the World Cup. I also would like to thank Steve Homer for agreeing to be another reader for this thesis, and for providing valuable career advice. I very much enjoyed having both academic and non-academic conversations with him and feasting on the delicious and authentic Thai food he and his wife served at his house. Special thanks to Alina Oprea and Evimaria Terzi for serving on my thesis committee and for helping me make this dissertation more accessible to a wider audience.

I am lucky to work with many outstanding coauthors and researchers that constantly surprise and impress me with their creativity and originality. I enjoyed all of those fruitful conversations with Ran Canetti, Benjamin Fuller, Kobbi Nissim, Leo Reyzin. Graduate school would not have been nearly as rewarding without my peers and my time at Boston would not have been the same without my friends at BU's CS department and its visitors: Qinxun Bai, Yilei Chen, Dora Erdos, Benjamin Fuller, Kun He, Ye Li, Shugao Ma, Charalampos Mavroforakis, Dimitris Papadopoulos, Davide Proserpio, Natali Runchasky, Mehrnoosh Sameki, Larissa Spinelli, Yuefeng Wang, Zheng Wu, Haohan Zhu, and many

others. I will miss your laugh and smile.

I would like to thank my dear, Snow, for her unconditionally love during my Ph.D. studies. Thank you for your delicious food, your beautiful smile, and your incomparable support. Lastly, thank you my Mom and Dad. There's no way that I can concisely articulate how supportive and caring you are. I would not have been where I am without your invaluable truly and deeply love and encouragement!

PRIVACY-PRESERVING QUERIES ON ENCRYPTED DATABASES

XIANRUI MENG

Boston University, Graduate School of Arts and Sciences, 2016

Major Professor: George Kollios, Professor of Computer Science

ABSTRACT

In today's Internet, with the advent of cloud computing, there is a natural desire for enterprises, organizations, and end users to outsource increasingly large amounts of data to a cloud provider. Therefore, ensuring security and privacy is becoming a significant challenge for cloud computing, especially for users with sensitive and valuable data. Recently, many efficient and scalable query processing methods over encrypted data have been proposed. Despite that, numerous challenges remain to be addressed due to the high complexity of many important queries on encrypted large-scale datasets. This thesis studies the problem of privacy-preserving database query processing on structured data (e.g., relational and graph databases). In particular, this thesis proposes several practical and provable secure structured encryption schemes that allow the data owner to encrypt data without losing the ability to query and retrieve it efficiently for authorized clients. This thesis includes two parts. The first part investigates graph encryption schemes. This thesis proposes a graph encryption scheme for approximate shortest distance queries. Such scheme allows the client to query the shortest distance between two nodes in an encrypted graph securely and efficiently. Moreover, this thesis also explores how the techniques can be applied to other graph queries. The second part of this thesis proposes secure top-k query processing schemes on encrypted relational databases. Furthermore, the thesis develops a scheme for the top-k join queries over multiple encrypted relations. Finally, this thesis demonstrates the practicality of the proposed encryption schemes by prototyping the encryption systems to perform queries on real-world encrypted datasets.

Contents

1 Introduction **1**

2 Notation and Preliminaries **6**

2.1 Notations 6

2.2 Cryptographic Tools 7

2.2.1 Encryption 7

2.2.2 Pseudo-random functions 10

3 Graph Encryption for Approximate Shortest Distance Queries **11**

3.1 Graph Encryption 11

3.2 Related Work 13

3.2.1 Graph privacy 13

3.2.2 Distance oracles 15

3.3 Distance Oracles for Shortest Distance Computation 15

3.3.1 Sketch-based oracles. 16

3.3.2 The Das Sarma *et al.* oracle. 16

3.3.3 The Cohen *et al.* oracle 17

3.3.4 Shortest distance queries 17

3.4 Distance Oracle Encryption 17

3.4.1 Security 18

3.4.2 Leakage 20

3.4.3 Efficiency 22

3.5 GRECS Constructions 22

3.5.1 A Computationally-Efficient Scheme 22
3.5.2 Security and efficiency. 23
3.5.3 A Communication-Efficient Scheme 24
3.5.4 Error Detection 30
3.5.5 A Space-Efficient Construction 32
3.6 Experimental Evaluation 37
3.6.1 Datasets 38
3.6.2 Overview 39
3.6.3 Performance of $\mathsf{GraphEnc}_1$ 39
3.6.4 Performance of $\mathsf{GraphEnc}_2$ 40
3.6.5 Performance of $\mathsf{GraphEnc}_3$ 41
3.6.6 Approximation errors 44
3.7 Application to Other Graph Queries 46
3.7.1 All-Distance Sketches 46
3.7.2 Graph Similarity Queries using All-Distance Sketches 47
3.7.3 Graph Encryption based on ADS 48

4 Top-k Query Processing on Encrypted Relational Databases 52
4.1 Introduction 52
4.2 Related Works and Background 55
4.3 Preliminaries 56
4.3.1 Problem Definition 56
4.3.2 The Architecture 57
4.3.3 Cryptographic Tools 58
4.3.4 No-Random-Access (NRA) Algorithm 59
4.4 Scheme Overview 61
4.5 Encrypted Hash List (EHL) 62
4.6 Database Encryption 66

4.7 Query Token 67
4.8 Top-k Query Processing 67
4.8.1 Query Processing: SecQuery 68
4.8.2 Building Blocks 70
4.9 Security Discussion 79
4.10 Query Optimization 80
4.10.1 Efficient SecDupElim 80
4.10.2 Batch Processing for SecQuery 81
4.10.3 Efficiency 82
4.11 Experiments 82
4.11.1 Evaluation of the Encryption Setup 84
4.11.2 Query Processing Performance 84
4.12 Top-k Join 90
4.12.1 Secure Top-k Join 91
4.12.2 Encryption Setup for Multiple databases 91
4.12.3 Query Token 92
4.12.4 Query Processing for top-k join 92
4.12.5 Related works on Secure Join 98
4.13 Top-k Query Processing Conclusion 99

5 Conclusions **100**
5.1 Future Directions 101

List of Journal Abbreviations **104**

Bibliography **107**

Curriculum Vitae of Xianrui Meng **114**

List of Tables

3.1 The graph datasets used in our experiments 38

3.2 The space, setup, communication, and query complexities of our constructions (α is set to be in $O(\log n)$). 39

3.3 A full performance summary for $\mathsf{GraphEnc}_1$, $\mathsf{GraphEnc}_2$, and $\mathsf{GraphEnc}_3$ . . . 40

4.1 Encrypted `patients` Heart-Disease Data 53

4.2 Notation Summarization 60

4.3 Total Communication Network Latency for each dataset when $k = 20$, $m = 4$ 90

List of Figures

3.1 Two sketches for nodes u and v. The approximate shortest distance $d = 5$. . . 17

3.2 One node's encrypted hash table. 25

3.3 Example of encrypting $\mathsf{Sk}_u = \{(a, d_1), (b, d_2), (c, d_3)\}$. 35

3.4 Collision probabilities for different datasets 41

3.5 Construction time and size overhead ($\mathbf{DO}_1$) 43

3.6 Construction time and size overhead ($\mathbf{DO}_2$) 43

3.7 Average Query time 44

3.8 Mean of Estimated Error with Standard Deviation 45

3.9 Absolute error histogram $\mathbf{DO}_2$ and $\rho = 3$ 46

4.1 An overview of our model 58

4.2 Encrypted Hash List for the object o. 63

4.3 Overview of the SecDedup protocol 77

4.4 Encryption using EHL vs. EHL$^+$. 83

4.5 Encryption EHL vs. EHL$^+$ on real data 83

4.6 Qry_F query performance 86

4.7 Qry_E query optimization performance 86

4.8 Qry_Ba query optimization performance 87

4.9 Comparisons ($k = 5$, $m = 2$, and $p = 500$) 88

4.10 Communication bandwidth evaluation 89

4.11 Top-k join: $\bowtie_{\mathsf{sec}}$ 98

List of Abbreviations

ADS	....	All Distance Sketch
BFS	....	Breath First Search
BGN	....	Boneh Goh Nissim Encryption Scheme
CPA	....	Chosen Plaintext Attack
CQA	....	Chosen Query Attack
DJ	....	Damgård-Jurik Cryptosystem
DO	....	Distance Oracle
DP	....	Differential Privacy
EHL	....	Encrypted Hash List
EO	....	Oralce Encryption
FHE	....	Fully Homomorphic Encryption
FPR	....	False Positive Rate
GraphEnc	....	Graph Encryption
GRECS	....	Graph Encryption for approximate shortest distance queries
HMAC	....	Keyed-hash Message Authentication Code
kNN	....	k Nearest Neighbor
MPC	....	Multiparty Computation
NRA	....	No Random Access Algorithm
ORAM	....	Oblivious RAM
PAMAP	....	Physical Activity Monitoring Dataset
PIR	....	Private Information Retrieval

PRF	....	Pseudo-random Function
PRP	....	Pseudo-random Permutation
QP	....	Query Pattern
RDF	....	Resource Description Framework
SE	....	Structured Encryption
SecBest	....	Secure Best Score Protocol
SecDedup	....	Secure Deduplication Protocol
SecJoin	....	Secure Join Protocol
SecUpdate	....	Secure Update Protocol
SecWorst	....	Secure Worse Score Protocol
SSE	....	Searchable Symmetric Encryption
SWHE	....	Somewhat Homomorphic Encryption

List of Symbols

$G = (V, E)$	....	An undirected graph with node set V and edge set E.
$\mathsf{dist}(u, v)$	....	Shortest Distance between u and v.
DX	....	A dictionary.
SKE	....	A symmetric key encryption scheme.
pk, sk	....	Public key and secret key.
Sk_v	....	A sketch for node v.
Ω_G	....	A distance oracle data structure for graph G.
DO	....	A distance oracle.
σ	....	Sampling parameter for the distance oracle.
α	....	Approximation factor for the distance oracle.
ε	....	Error parameter.
EO	....	An encrypted graph
S	....	The maximum sketch size.
ρ	....	The ranking parameter for ADS.
$\mathcal{L}_{\mathsf{Setup}}$	....	Setup leakage.
$\mathcal{L}_{\mathsf{Query}}$	....	Query leakage.
Ideal	....	The Ideal world of the execution.
Real	....	The Real world of the execution.
$\mathsf{GraphEnc}_1$	....	The computation-efficient GE.
$\mathsf{GraphEnc}_2$	....	The communication-efficient GE.
$\mathsf{GraphEnc}_3$	....	The computation and communication-efficient GE.

$\mathcal{L}_{SP}$	....	The Sketch pattern leakage.
ESk_v	....	Encrypted sketch for node v.
$\mathcal{H}$	....	Universal hash family.
Coll_v	....	Hash collision for node v.
$\mathsf{XColl}_{u,v}$	....	Inter-hash collision between node v and u.
J^*	....	Dijkstra Rank Closeness.
$\mathsf{Enc}(m)$	....	Paillier encryption of m.
$\mathsf{Dec}(c)$	....	Paillier decryption of c.
$\mathsf{E}^2(m)$	....	Damgård-Jurik (DJ) encryption of m.
$\mathsf{E}(x) \sim \mathsf{E}(y)$	....	Denotes $x = y$, i.e. $\mathsf{Dec}(\mathsf{E}(x)) = \mathsf{Dec}(\mathsf{E}(y))$.
$\mathsf{EHL}(o)$	....	Encrypted Hash List of the object o.
$\mathsf{EHL}^+(o)$	....	Efficient Encrypted Hash List of the object o
$\ominus, \odot$	....	EHL and EHL^+ operations.
I_i^d	....	The data item in the ith sorted list L_i at depth d.
$E(I_i^d)$	....	Encrypted data item I_i^d
$F_W(o)$	....	Cost function in the query token
$B^d(o)$	....	The best score (upper bound) of o at depth d
$W^d(o)$	....	The worst score (lower bound) of o at depth d
$\mathbf{E}(I)$	....	The encrypted item that contains $(\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$

Chapter 1

Introduction

As remote storage and cloud computing services emerge, such as Amazon's EC2, Google AppEngine, and Microsoft's Azure, many enterprises, organizations, and end users may outsource their data to those cloud service providers for reliable maintenance, lower cost, and better performance. In fact, a number of database systems on the cloud have been developed recently that offer high availability and flexibility at relatively low costs.

However, despite these benefits, there are still a number of reasons that make many users to refrain from using these services, especially users with sensitive and valuable data. Undoubtedly, the main issue for this is related to security and privacy concerns [Agrawal et al., 2011]. Indeed, data owner and clients may not fully trust a public cloud since the hackers, or the cloud's administrators with root privilege can fully access all data for any purpose. Sometimes the cloud provider may sell its business to an untrusted company, which will have full access to the data. Therefore, ensuring security and privacy is becoming a significant challenge for cloud computing, especially for users with sensitive and valuable data. In addition, the benefits of big data - including advances in machine learning, e-commerce, social sciences, and marketing - are well-publicized, but the various privacy and security problems it presents have received less attention from the public at large.

One approach to address these issues is to encrypt the data before outsourcing them to the cloud. For example, electronic health records (EHRs) should be encrypted before outsourcing in compliance with regulations like HIPAA[1]. Encrypted data can bring an en-

[1] HIPAA is the federal Health Insurance Portability and Accountability Act of 1996.

hanced security into the Database-As-Service environment [Hacigümüs et al., 2002]. However, it also introduces significant difficulties in querying and computing over these data. In recent years, many works have been proposed for computing over encrypted data. In general, the main technical difficulty is to query an encrypted database without ever having to decrypt it. A number of techniques related to practical query processing over encrypted data have been proposed recently, including keyword search queries [Song et al., 2000, Cash et al., 2013b, Curtmola et al., 2011], range queries [Shi et al., 2007, Hore et al., 2012, Li et al., 2014], k-nearest neighbor queries [Wong et al., 2009, Elmehdwi et al., 2014, Yao et al., 2013, Choi et al., 2014], as well as other aggregate queries.

This thesis proposes practical and scalable encryption schemes for a number of important database queries. All of the encryption schemes are practical and provably secure. In particular, these encryption schemes allow the data owner to encrypt data without losing the ability to query and retrieve it efficiently for authorized clients. This thesis mainly focuses on two different type of real-world databases: graph databases and relational databases.

The first part of this thesis focuses on the problem of designing graph encryption schemes that support one of the most fundamental and important graph operations: finding the shortest distance between two nodes. Shortest distance queries are a basic operation in many graph algorithms but also have applications of their own. For instance, on a social network, shortest distance queries return the shortest number of introductions necessary for one person to meet another. In protein-protein interaction networks they can be used to find the functional correlations among proteins [Przulj et al., 2004] and on a phone call graph (i.e., a graph that consists of phone numbers as vertices and calls as edges) they return the shortest number of calls connecting two nodes. The thesis develops graph encryption schemes that efficiently support *approximate* shortest distance queries on large-scale encrypted graphs. Shortest distance queries are one of the most fundamental graph operations and have a wide range of applications. Using such graph encryption schemes, a client can outsource large-scale privacy-sensitive graphs to an untrusted server without losing the

ability to query it. Other applications include encrypted graph databases and controlled disclosure systems. In particular, the thesis proposes **GRECS** (stands for GRaph EnCryption for approximate Shortest distance queries) which includes three schemes that are provably secure against any semi-honest server. Furthermore, this thesis also shows that the building blocks in GRECS can be used for other graph queries.

The second part focuses on relational databases. In particular, this thesis proposes secure and efficient processing protocols of top-k queries over outsourced relational databases under the non-colluding semi-honest clouds model. The thesis also formulates and constructs several novel secure sub-protocols, such as secure best/worst score and secure deduplication, which can be adapted as stand-alone building blocks for many other applications. In particular, during the querying phase the computation performed by the client is very small. The client only needs to compute a simple token for the server and all of the relatively heavier computations are performed by the cloud side. The schemes are implemented and have been demonstrated to be very efficient by running a set of experiments on a number of real-world databases. Furthermore, the thesis shows that the techniques can be adopted for handling top-k join on multiple encrypted databases, as well as general join queries.

Contributions. To summarize, the contributions of this thesis are listed below:

- This thesis proposes three graph encryption schemes for approximate shortest distance queries (GRECS). In particular,
  - The first scheme only makes use of symmetric-key operations and, as such, is computationally-efficient.
  - The second scheme makes use of somewhat-homomorphic encryption and achieves optimal communication complexity.
  - The third scheme is computationally-efficient, achieves optimal communication complexity and produces compact encrypted oracles at the cost of some extra

leakage.

- All of the proposed constructions are adaptively semantically-secure with reasonable leakage functions.

- The encryption schemes are implemented and evaluated over real large-scale graphs and it is demonstrated that the constructions are practical and scalable.

- The thesis proposes a new practical protocol designed to answer top-k queries over encrypted relational databases.

- The thesis proposes two encrypted data structures called EHL and EHL^+ which allow the servers to homomorphically evaluate the equality relations between two objects.

- The thesis proposes several independent sub-protocols such that the cloud can securely computes the best/worst scores and de-duplicate replicated encrypted objects with the use of another non-colluding server. These protocols are building blocks when computing the top-k queries.

- The thesis proposes schemes for secure top-k join queries and generic join queries.

- All of the schemes are experimentally evaluated using real-world datasets and result shows that the scheme is efficient and practical.

The thesis is organized as follows:

- Chapter 2 introduces the preliminaries and notation we used.

- Chapter 3 proposes GRECS which includes three schemes. This chapter first discusses the formal security definitions for the graph encryption scheme and distance oracle data structures. It then describes the three schemes in details and demonstrates the experimental results. In addition, it also briefly introduces other graph queries on encrypted graphs.

- Chapter 4 discusses secure top-k query processing on encrypted relations and describes the scheme in detail. This chapter also proposes the top-k join query processing on multiple encrypted relations, and briefly introduces the protocol for generic secure join queries.
- Chapter 5 concludes the thesis.

Chapter 2

Notation and Preliminaries

2.1 Notations

Let S be a set, then $|S|$ refers to its cardinality. The notation $\{I_i\}$ represents a set of items with the same form of the item I_i. The notation $[n]$ represents the set of integers $\{1, \ldots, n\}$. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ. We write $x \stackrel{\$}{\leftarrow} X$ to represent an element x being uniformly sampled at random from a set X. The output x of a probabilistic algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$ and that of a deterministic algorithm $\mathcal{B}$ by $x := \mathcal{B}$. Given a sequence of elements $\mathbf{v}$, we define its ith element either as v_i or $\mathbf{v}[i]$ and its total number of elements as $|\mathbf{v}|$. Throughout, $\lambda \in \mathbb{N}$ will denote the security parameter and we assume all algorithms take λ implicitly as input. A function $\nu : \mathbb{N} \to \mathbb{N}$ is negligible in λ if for every positive polynomial $p(\cdot)$ and all sufficiently large $\lambda, \nu(\lambda) < 1/p(\lambda)$. We write $f(\lambda) = \mathsf{poly}(\lambda)$ to mean that there exists a polynomial $p(\cdot)$ such that $f(\lambda) \leq p(\lambda)$ for all sufficiently large $\lambda \in \mathbb{N}$; and we similarly write $f(\lambda) = \mathsf{negl}(\lambda)$ to mean that there exists a negligible function $\nu(\cdot)$ such that $f(\lambda) \leq \nu(\lambda)$ for all sufficiently large λ. A dictionary DX is a data structure that stores label/value pairs $(\ell_i, v_i)_{i=1}^{n}$. Dictionaries support insert and lookup operations defined as follows: an insert operation takes as input a dictionary DX and a label/value pair (ℓ, v) and adds the pair to DX. We denote this as $\mathsf{DX}[\ell] := v$. A lookup operation takes as input a dictionary DX a label ℓ_i and returns the associated value v_i. We denote this as $v_i := \mathsf{DX}[\ell_i]$. Dictionaries can be instantiated using hash tables and various kinds of search trees. Given an undirected graph $G = (V, E)$, we denote its total number of nodes as $n = |V|$ and its

number of edges as $m = |E|$. A shortest distance query $q = (u, v)$ asks for the length of the shortest path between u and v which we denote $\mathsf{dist}(u, v)$.

2.2 Cryptographic Tools

2.2.1 Encryption

In this work, we make use of several kinds of encryption schemes including standard symmetric-key encryption and homomorphic encryption. A symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a set of three polynomial-time algorithms that work as follows. Gen is a probabilistic algorithm that takes a security parameter k as input and returns a secret key K; Enc is a probabilistic algorithm that takes as input a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes as input a key K and a ciphertext c and returns m if K was the key under which c was produced. A public-key encryption scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is similarly defined except that Gen outputs a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ and Enc encrypts messages with the public key pk. Informally, an encryption scheme is CPA-secure (*Chosen-Plaintext-Attack-secure*) if the ciphertexts it outputs do not reveal any partial information about the messages even to an adversary that can adaptively query an encryption oracle. We refer the reader to [Katz and Lindell, 2008] for formal definitions of symmetric-key encryption and CPA-security.

A public-key encryption scheme SWHE is homomorphic if, in addition to the three algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, it also includes an evaluation algorithm Eval that takes as input a function f and a set of ciphertexts $c_1 \leftarrow \mathsf{SWHE.Enc}_{\mathsf{pk}}(m_1)$ through $c_n \leftarrow \mathsf{SWHE.Enc}_{\mathsf{pk}}(m_n)$ and returns a ciphertext c such that $\mathsf{Dec}_{\mathsf{sk}}(c) = f(m_1, \ldots, m_n)$. If a homomorphic encryption scheme supports the evaluation of any polynomial-time function, then it is a fully-homomorphic encryption (FHE) scheme [Rivest et al., 1978, Gentry, 2009b] otherwise it is a somewhat homomorphic encryption (SWHE) scheme. In this dissertation, we make use of only "low degree" homomorphic encryption. In particular, in Chap-

ter 3, we require the encryption scheme support the evaluation of quadratic polynomials, In particular, we need the evaluation algorithm to support any number of additions: $\mathsf{SWHE.Enc}_{\mathsf{pk}}(m_1 + m_2) = \mathsf{SWHE.Eval}\big(+, \mathsf{Enc}_{\mathsf{pk}}(m_1), \mathsf{Enc}_{\mathsf{pk}}(m_2)\big)$; and a *single* multiplication: $\mathsf{SWHE.Enc}_{\mathsf{pk}}(m_1 \cdot m_2) = \mathsf{SWHE.Eval}(\times, \mathsf{Enc}_{\mathsf{pk}}(m_1), \mathsf{Enc}_{\mathsf{pk}}(m_2))$, that is, a ciphertext that results from a homomorphic multiplication cannot be used in another homomorphic multiplication. Concrete instantiations of such schemes include the scheme of Boneh, Goh and Nissim (BGN) [Boneh et al., 2005] based on bilinear maps and the scheme of Gentry, Halevi and Vaikuntanathan [Gentry et al., 2010] based on lattices. In Chapter 4, as we only require the encryption scheme to be additively homomorphic, we choose the Paillier encryption scheme. Below we describe the two encryption schemes, Paillier encryption and Boneh-Goh-Nissim encryption.

Paillier Crypotsystem The Paillier cryptosystem is a semantically secure public key encryption scheme based on the Decisional Composite Residuosity assumption. More specifically, the Paillier cryptosystem is defined as follows:

- *Key Generation.* To construct the public key, set an RSA modulus $n = pq$ of k bits where p and q large primes such that $gcd(pq, (p-1)(q-1)) = 1$. Let $K = lcm((p-1)(q-1))$ and pick generator $g \in \mathbb{Z}_n^*$. The public key is the pair $\mathsf{pk} = (n, g)$ and the secret is $\mathsf{sk} = K$.

- *Encryption.* To encrypt a message $m \in \mathbb{Z}_n$: choose $r \xleftarrow{\$} \mathbb{Z}_n^*$ and compute $\mathsf{Enc}_{\mathsf{pk}}(m) = g^m r^n \mod n^2$.

- *Decryption.* To decrypt a ciphertext $c = \mathsf{Enc}(m)$ compute:

$$m = \frac{L(c^{\mathsf{sk}}) \mod n^2}{L(g^{\mathsf{sk}}) \mod n^2} \mod n, \text{ where} L(u) = \frac{u-1}{n}$$

Boneh-Goh-Nissim (BGN) Crypotsystem The cryptosystem devised by Boneh, Goh, and Nissim [Boneh et al., 2005] was the first to allow both additions and multiplica-

tions with a constant-size ciphertext. The encryption uses pairings on elliptic curves. The scheme makes use of certain finite groups of composite order that support a bilinear map. We use the following notation

- G and G_1 are two (multiplicative) cyclic groups of finite order n.
- g is a generator of G.
- e is a bilinear map $e : G \times G \to G_1$. In other words, for all $u, v \in G$ and $a, b \in \mathbb{Z}$, we have $e(u^a, v^b) = e(u, v)^{ab}$. We also require that $e(g, g)$ is a generator of G_1.

We say that G is a bilinear group if there exists a group G_1 and a bilinear map as above. In the next section we also add the requirement that the group action in G, G_1, and the bilinear map can be computed in polynomial time. We refer to [Boneh et al., 2005] on how to generate the public parameters (q_1, q_2, G, G_1, e), where q_1 and q_2 are two primes with k-bit. We now describe the three key algorithm that makes up the system:

- *Key Generation.* Generate the parameters (q_1, q_2, G, G_1, e) as described in [Boneh et al., 2005], where q_1 and q_2 are two primes with k-bit. Let $n = q_1 q_2$. Pick two random generators $g, u \xleftarrow{\$} G$ and set $h = uq_2$. Then h is a random generator of the subgroup of G of order q_1. The public key is $\mathsf{pk} = (n, G, G1, e, g, h)$. The private key is $\mathsf{sk} = q_1$.
- *Encryption.* We assume the message space consists of integers in the set $\{0, 1, ..., T\}$ with $T < q_2$. To encrypt a message m: choose $r \xleftarrow{\$} \mathbb{Z}_n$ and compute $\mathsf{Enc}_{\mathsf{pk}}(m) = g^m h^r \mod n^2$.
- *Decryption.* To decrypt a ciphertext C using the private key $\mathsf{sk} = q_1$, observe that

$$C^{q_1} = (g^m h^r)^{q_1} = (g^{q_1})^m$$

  Let $g' = g^{q_1}$. To recover m, it suffices to compute the discrete log of C^{q_1} base g'.

Since $0 \leq m \leq T$ this takes expected time $\tilde{O}(\sqrt{T})$ using Pollards lambda method (see [Menezes et al., 1996], p.128).

2.2.2 Pseudo-random functions

A pseudo-random function (PRF) from domain $\mathcal{D}$ to co-domain $\mathcal{R}$ is a function family that is computationally indistinguishable from a random function. In other words, no computationally-bounded adversary can distinguish between oracle access to a function that is chosen uniformly at random in the family and oracle access to a function chosen uniformly at random from the space of all functions from $\mathcal{D}$ to $\mathcal{R}$. A pseudo-random permutation (PRP) is a pseudo-random family of permutations over $\mathcal{D}$. We refer the reader to [Katz and Lindell, 2008] for formal definitions of PRFs and PRPs.

Chapter 3

Graph Encryption for Approximate Shortest Distance Queries

3.1 Graph Encryption

Graph databases that store, manage, and query large graphs have received increased interest recently due to many large-scale database applications that can be modeled as graph problems. Example applications include storing and querying large Web graphs, online social networks, biological networks, RDF datasets, and communication networks. As a result, a number of systems have been proposed to manage, query, and analyze massive graphs both in academia (e.g., Pregel [Malewicz et al., 2010], GraphLab [Low et al., 2010], Horton [Sarwat et al., 2012], Trinity [Shao et al., 2013], TurboGraph [Han et al., 2013], and GraphChi-DB [Kyrola and Guestrin, 2014]) and industry (e.g., Neo4j, Titan, DEX, and GraphBase). Furthermore, with the advent of cloud computing, there is a natural desire for enterprises and startups to outsource the storage and management of their databases to a cloud provider. Like any large-scale database, as cloud computing has emerged as important infrastructure, a lot of graph dataset has been outsourced to the cloud provider from the user/client. Graph database contains very sensitive information about the entities in the graph, therefore, one would like to protect those information against the adversarial cloud service provider. Increasing concerns about data security and privacy in the cloud, however, have curbed many data owners' enthusiasm about storing their databases in the cloud.

To address this, Chase and Kamara [Chase and Kamara, 2010] introduced the notion of graph encryption. Roughly speaking, a graph encryption scheme encrypts a graph in such a way that it can be privately queried. Using such a scheme, an organization can safely outsource its encrypted graph to an untrusted cloud provider without losing the ability to query it. Several constructions were described in [Chase and Kamara, 2010] including schemes that support adjacency queries (i.e., given two nodes, do they have an edge in common?), neighbor queries (i.e., given a node, return all its neighbors) and focused subgraph queries on web graphs (a complex query used to do ranked web searches). Graph encryption is a special case of *structured encryption*, which are schemes that encrypt data structures in such a way that they can be privately queried. The most well-studied class of structured encryption schemes are *searchable symmetric encryption* (SSE) schemes [Song et al., 2000, Chang and Mitzenmacher, 2005, Goh, 2003, Curtmola et al., 2006, Kamara et al., 2012, Kamara and Papamanthou, 2013, Cash et al., 2013a, Cash et al., 2014, Naveed et al., 2014, Stefanov et al., 2014] which, roughly speaking, encrypt search structures (e.g., indexes or search trees) for the purpose of efficiently searching on encrypted data.

In this work, we focus on the problem of designing graph encryption schemes that support one of the most fundamental and important graph operations: finding the shortest distance between two nodes. Shortest distance queries are a basic operation in many graph algorithms but also have applications of their own. For instance, on a social network, shortest distance queries return the shortest number of introductions necessary for one person to meet another. In protein-protein interaction networks they can be used to find the functional correlations among proteins [Przulj et al., 2004] and on a phone call graph (i.e., a graph that consists of phone numbers as vertices and calls as edges) they return the shortest number of calls connecting two nodes.

The techniques Computing shortest distance queries on massive graphs (e.g., the Web graph, online social networks or a country's call graph) can be very expensive.

For example, it takes $O(|E| + |V|\log|V|)$to compute the shortest distance using Dijkstra's algorithm. Therefore, in practice, one typically pre-computes a data structure from the graph called a *distance oracle* that answers shortest distance queries *approximately* [Thorup and Zwick, 2005, Das Sarma et al., 2010, Cohen et al., 2013]; that is, given two vertices v_1 and v_2, the structure returns a distance d that is at most $\alpha \cdot \mathsf{dist}(v_1, v_2) + \beta$, where $\alpha, \beta > 1$ and $\mathsf{dist}(v_1, v_2)$ is the exact distance between v_1 and v_2.

In this work, we focus on designing structured encryption schemes for a certain class of distance oracles referred to as *sketch-based* oracles. Below we summarize our contributions:

- We propose three distance oracle encryption schemes. Our first scheme only makes use of symmetric-key operations and, as such, is very computationally-efficient. Our second scheme makes use of somewhat-homomorphic encryption and achieves optimal communication complexity. Our third scheme is computationally-efficient, achieves optimal communication complexity and produces compact encrypted oracles at the cost of some leakage.

- We show that all our constructions are adaptively semantically-secure with reasonable leakage functions.

- We implement and evaluate our solutions on real large-scale graphs and show that our constructions are practical.

3.2 Related Work

3.2.1 Graph privacy

Privacy-preserving graph processing has been considered in the past. Most of the work in this area, however, focuses on privacy models that are different than ours. Some of the proposed approaches include structural anonymization to protect neighborhood information [Gao et al., 2011, Liu and Terzi, 2008, Cheng et al., 2010], use differential privacy [Dwork et al., 2006] to query graph statistics privately [Kasiviswanathan et al., 2013,

Shen and Yu, 2013], or use private information retrieval (PIR) [Mouratidis and Yiu, 2012] to privately recover shortest paths. We note that none of these approaches are appropriate in our context where the graph itself stores sensitive information (and therefore must be hidden unlike in the PIR scenario) and is stored remotely (unlike the differential privacy and anonymization scenarios). Structured and graph encryption was introduced by Chase and Kamara in [Chase and Kamara, 2010]. Structured encryption is a generalization of searchable symmetric encryption (SSE) which was first proposed by Song, Wagner and Perrig [Song et al., 2000]. The notion of adaptive semantic security was introduced by Curtmola, Garay, Kamara and Ostrovsky in [Curtmola et al., 2006] and generalized to the setting of structured encryption in [Chase and Kamara, 2010]. One could also encrypt and outsource the graph using fully homomorphic encryption [Gentry, 2009b], which supports arbitrary computations on encrypted data, but this would be prohibitively slow in practice. Another approach is to execute graph algorithms over encrypted and outsourced graphs is to use Oblivious RAM [Goldreich and Ostrovsky, 1996] over the adjacency matrix of the graph. This approach, however, is inefficient and not practical even for small graphs since it requires storage that is quadratic in the number of nodes in the graph and a large number of costly oblivious operations. Recent work by [Wang et al., 2014] presents an oblivious data structure for computing shortest paths on planar graphs using ORAM. For a sparse planar graph with $O(n)$ edges, their approach requires $O(n^{1.5})$ space complexity at the cost of $O(\sqrt{n}\log n)$ online query time. Recent works based on ORAM, such as [Liu et al., 2014, Liu et al., 2015b], also propose oblivious secure computation frameworks that can be used to compute single source shortest paths. However, these are general purpose frameworks and are not optimized to answer shortest distance queries. Note that these works solve more generic problems on oblivious secure computation rather than just shortest path distance queries. Other techniques, such as those developed by Blanton, Steele and Aliasgari [Blanton et al., 2013] and by Aly et al. [Aly et al., 2013] do not seem to scale to sparse graphs with millions of nodes due to the quadratic complexity of the underlying operations which are instantiated with secure multi-party computation

protocols.

3.2.2 Distance oracles

Computing shortest distances on large graphs using Dijkstra's algorithm or breadth first search is very expensive. Alternatively, it is not practical to store all-pairs-shortest-distances since it requires $O(n^2)$ space for n nodes. To address this, in practice, one pre-computes a data structure called a *distance oracle* that supports *approximate* shortest distance queries between two nodes with logarithmic query time. Solutions such as [Das Sarma et al., 2010, Potamias et al., 2009, Qi et al., 2013, Cohen et al., 2013, Chechik, 2014, Cohen, 2014, Cohen et al., 2003] carefully select seed nodes (also known as landmarks) and store the shortest distances from all the nodes to the seeds. The advantage of using such a data structure is that they are compact and the query time is very fast. For example, the distance oracle construction of Das Sarma, Gollapudi, Najork and Panigrahy [Das Sarma et al., 2010] requires $\tilde{O}(n^{1/c})$ work to return a $(2c-1)$-approximation of the shortest distance for some constant c.

3.3 Distance Oracles for Shortest Distance Computation

At a high-level, our approach to designing graph encryption schemes for shortest distance queries consists of encrypting a distance oracle in such a way that it can be queried privately. A distance oracle is a data structure that supports approximate shortest distance queries. A trivial construction consists of pre-computing and storing all the pairwise shortest distances between nodes in the graph. The query complexity of such a solution is $O(1)$ but the storage complexity is $O(n^2)$ which is not practical for large graphs.

We consider two practical distance oracle constructions. Both solutions are sketch-based which means that they assign a sketch Sk_v to each node $v \in V$ in such a way that the approximate distance between two nodes u and v can be efficiently (sublinear) computed from the sketches Sk_u and Sk_v. The first construction is by Das Sarma *et*

al. [Das Sarma et al., 2010] which is itself based on a construction of Thorup and Zwick [Thorup and Zwick, 2005] and the second is by Cohen *et al.* [Cohen et al., 2013]. The two solutions produce sketches of the same form and distance queries are answered using the same operation.

3.3.1 Sketch-based oracles.

More formally, a sketch-based distance oracle $\mathsf{DO} = (\mathsf{Setup}, \mathsf{Query})$ is a pair of efficient algorithms that work as follows. Setup takes as input a graph G, an approximation factor α and an error bound ε and outputs an oracle $\Omega_G = \{\mathsf{Sk}_v\}_{v \in V}$. Query takes as input an oracle Ω_G and a shortest distance query $q = (u, v)$. We say that DO is (α, ε)-correct if for all graphs G and all queries $q = (u, v)$, $\Pr[\,\mathsf{dist}(u, v) \leq d \leq \alpha \cdot \mathsf{dist}(u, v)\,] \geq 1 - \varepsilon$, where $d := \mathsf{Query}(\Omega_G, u, v)$. The probability is over the randomness of algorithm Setup.

3.3.2 The Das Sarma *et al.* oracle.

The Setup algorithm makes $\sigma = \widetilde{\Theta}(n^{2/(\alpha+1)})$ calls to a Sketch sub-routine with the graph G. Throughout, we refer to σ as the oracle's *sampling parameter* and we note that it affects the size of the sketches. During the ith call, the Sketch routine generates and returns a collection of sketches $(\mathsf{Sk}^i_{v_1}, \ldots, \mathsf{Sk}^i_{v_n})$, one for every node $v_j \in V$. Each sketch $\mathsf{Sk}^i_{v_j}$ is a set constructed as follows. During the ith call to Sketch, it samples uniformly at random $\zeta = \log n$ sets of nodes $S_0, \ldots, S_{\zeta-1}$ of progressively larger sizes. In particular, for all $0 \leq z \leq \zeta - 1$, set S_z is of size 2^z. $\mathsf{Sk}^i_{v_j}$ then consists of ζ pairs $\{(w_z, \delta_z)\}_{0 \leq z \leq \zeta-1}$ such that w_z is the closest node to v_j among the nodes in S_z and $\delta_z = \mathsf{dist}(v_j, w_z)$. Having computed σ collections of sketches $(\mathsf{Sk}^i_{v_1}, \ldots, \mathsf{Sk}^i_{v_n})_{i \in [\sigma]}$, Setup then generates, for each node $v_j \in V$, a final sketch $\mathsf{Sk}_{v_j} = \bigcup_{i=1}^{\sigma} \mathsf{Sk}^i_{v_j}$. Finally, it outputs a distance oracle $\Omega_G = (\mathsf{Sk}_{v_1}, \ldots, \mathsf{Sk}_{v_n})$. Throughout, we refer to the uniformly sampled nodes stored in the node/distance pairs of the sketches as *seeds*.

3.3.3 The Cohen *et al.* oracle

The Setup algorithm assigns to each node $v \in V$ a sketch Sk_v that includes pairs (w, δ) chosen as follows. It first chooses a random rank function $\mathsf{rk} : V \to [0,1]$; that is, a function that assigns to each $v \in V$ a value distributed uniformly at random from $[0,1]$. Let $N_d(v)$ be the set of nodes within distance $d-1$ of v and let $\rho = \Theta(n^{2/(\alpha+1)})$. Throughout, we refer to ρ as the oracle's *rank parameter* and note that it affects the size of the sketches. For each node $v \in V$, the sketch Sk_v includes pairs (w, δ) such that $\mathsf{rk}(w)$ is less than the ρth value in the sorted set $\{\mathsf{rk}(y) : y \in N_{\mathsf{dist}(u,v)}(v)\}$. Finally it outputs a distance oracle $\Omega_G = (\mathsf{Sk}_{v_1}, \ldots, \mathsf{Sk}_{v_n})$. Like above, we refer to the nodes stored in the node/distance pairs of the sketches as seeds.

3.3.4 Shortest distance queries

The two oracle constructions share the same Query algorithm which works as follows. Given a query $q = (u, v)$, it finds the set of nodes $\mathbf{I}$ in common between Sk_u and Sk_v and returns the minimum over $s \in \mathbf{I}$ of $\mathsf{dist}(u,s) + \mathsf{dist}(s,v)$. If there are no nodes in common, then it returns $\perp$.

$\mathsf{Sk}(u)$: $\{(a,3), (b,3), (e,6), (g,3), (h,4)\}$
$\mathsf{Sk}(v)$: $\{(b,2), (d,1), (e,3), (h,3), (f,7)\}$

Figure 3.1: Two sketches for nodes u and v. The approximate shortest distance $d = 5$.

3.4 Distance Oracle Encryption

In this section we present the syntax and security definition for our oracle encryption schemes. There are many variants of structured encryption, including interactive and non-interactive, response-revealing and response-hiding. We consider interactive and response-hiding schemes which denote the fact that the scheme's query operation requires at least two messages (one from client and a response from server) and that queries output no

information to the server.

Definition 3.4.1 (Oracle Encryption)**.** *A distance oracle encryption scheme* $\mathsf{Graph} = (\mathsf{Setup}, \mathsf{distQuery})$ *consists of a polynomial-time algorithm and a polynomial-time two-party protocol that work as follows:*

- $(K, \mathsf{EO}) \leftarrow \mathsf{Setup}(1^\lambda, \Omega, \alpha, \varepsilon)$*: is a probabilistic algorithm that takes as input a security parameter* λ*, a distance oracle* Ω*, an approximation factor* α*, and an error parameter* ε*. It outputs a secret key* K *and an encrypted oracle* EO*.*

- $(d, \perp) \leftarrow \mathsf{distQuery}_{C,S}\big((K, q), \mathsf{EO}\big)$*: is a two-party protocol between a client* C *that holds a key* K *and a shortest distance query* $q = (u, v) \in V^2$ *and a server* S *that holds an encrypted oracle* EO*. After executing the protocol, the client receives a distance* $d \geq 0$ *and the server receives* $\perp$*. We sometimes omit the subscripts* C *and* S *when the parties are clear from the context.*

For $\alpha \geq 1$ *and* $\varepsilon < 1$*, we say that* Graph *is* (α, ε)*-correct if for all* $k \in \mathbb{N}$*, for all* Ω *and for all* $q = (u, v) \in V^2$*,*

$$\Pr\left[\, d \leq \alpha \cdot \mathsf{dist}(u, v) \,\right] \geq 1 - \varepsilon,$$

where the probability is over the randomness in computing $(K, \mathsf{EO}) \leftarrow \mathsf{Setup}(1^\lambda, \Omega, \alpha, \varepsilon)$ *and then* $(d, \perp) \leftarrow \mathsf{distQuery}\big((K, q), \mathsf{EO}\big)$*.*

3.4.1 Security

At a high level, the security guarantee we require from an oracle encryption scheme is that: (1) given an encrypted oracle, no adversary can learn any information about the underlying oracle; and (2) given the view of a polynomial number of $\mathsf{distQuery}$ executions for an adaptively generated sequence of queries $\mathbf{q} = (q_1, \ldots, q_n)$, no adversary can learn any partial information about either Ω_G or $\mathbf{q}$.

Such a security notion can be difficult to achieve efficiently, so often one allows for some form of leakage. Following [Curtmola et al., 2006, Chase and Kamara, 2010], this

is usually formalized by parameterizing the security definition with leakage functions for each operation of the scheme which in this case include the Setup algorithm and $\mathsf{distQuery}$ protocol.

We adapt the notion of adaptive semantic security from [Curtmola et al., 2006, Chase and Kamara, 2010] to our setting to the case of distance oracle encryption.

Definition 3.4.2. *Let* $\mathsf{Graph} = (\mathsf{Setup}, \mathsf{distQuery})$ *be an oracle encryption scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is a semi-honest adversary,* $\mathcal{C}$ *is a challenger,* $\mathcal{S}$ *is a simulator and* $\mathcal{L}_{\mathsf{Setup}}$ *and* $\mathcal{L}_{\mathsf{Query}}$ *are (stateful) leakage functions:*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$**:**

- $\mathcal{A}$ *outputs an oracle* Ω*, its approximation factor* α *and its error parameter* ε*.*
- *Given* $\mathcal{L}_{\mathsf{Setup}}(\Omega)$*,* 1^λ*,* α *and* ε*,* $\mathcal{S}$ *generates and sends an encrypted graph* EO *to* $\mathcal{A}$*.*
- $\mathcal{A}$ *generates a polynomial number of adaptively chosen queries* $(q_1, \ldots, q_m)$*. For each* q_i*,* $\mathcal{S}$ *is given* $\mathcal{L}_{\mathsf{Query}}(\Omega, q_i)$ *and* $\mathcal{A}$ *and* $\mathcal{S}$ *execute a simulation of* $\mathsf{distQuery}$ *with* $\mathcal{A}$ *playing the role of the server and* $\mathcal{S}$ *playing the role of the client.*
- $\mathcal{A}$ *computes a bit* b *that is output by the experiment.*

$\mathbf{Real}_{\mathcal{A}}(1^\lambda)$**:**

- $\mathcal{A}$ *outputs an oracle* Ω*, its approximation factor* α *and its error parameter* ε*.*
- $\mathcal{C}$ *computes* $(K, \mathsf{EO}) \leftarrow \mathsf{Setup}(1^\lambda, \Omega, \alpha, \varepsilon)$ *and sends the encrypted graph* EO *to* $\mathcal{A}$*.*
- $\mathcal{A}$ *generates a polynomial number of adaptively chosen queries* $(q_1, \ldots, q_m)$*. For each query* q_i*,* $\mathcal{A}$ *and* $\mathcal{C}$ *execute* $\mathsf{distQuery}_{\mathcal{C},\mathcal{A}}\big((K, q), \mathsf{EO}\big)$*.*
- $\mathcal{A}$ *computes a bit* b *that is output by the experiment.*

We say that Graph *is adaptively* $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}})$*-semantically secure if for all ppt adversaries* $\mathcal{A}$*, there exists a ppt simulator* $\mathcal{S}$ *such that*

$$\left|\Pr\left[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)\right] = 1\right| = \mathsf{negl}(k).$$

The definition above captures the fact that, given the encrypted oracle and its view of the query protocol, an adversarial server cannot learn any information about the oracle beyond the leakage.

3.4.2 Leakage

All the distance oracle encryption schemes we discuss in this work leak information. We describe and formalize these leakages below.

3.4.2.1 Setup leakage

The setup leakage of our first and second constructions, $\mathsf{GraphEnc}_1$ and $\mathsf{GraphEnc}_2$ in Sections 3.5.1 and 3.5.3, includes the total number of nodes in the underlying graph n, the maximum sketch size $S = \max_{v \in V} |\mathsf{Sk}_v|$ and the maximum distance over all seeds $D = \max_{v \in V} \max_{(w,\delta) \in \mathsf{Sk}_v} \delta$. The setup leakage of our third construction, $\mathsf{GraphEnc}_3$ in Section 3.5.5, includes n, S, D and the total number of seeds $Z = \sum_{v \in V} |\mathsf{Sk}_v|$.

3.4.2.2 Query pattern leakage

The query leakage of our first two constructions, $\mathsf{GraphEnc}_1$ and $\mathsf{GraphEnc}_2$, reveals whether the nodes in the query have appeared before. We refer to this as the *query pattern leakage* and formalize it below.

Definition 3.4.3 (Query pattern)**.** *For two queries q, q' define $Sim(q, q') = (u = u', u = v', v = u', v = v')$, i.e., whether each of the nodes $q = (u, v)$ matches each of the nodes of $q' = (u', v')$. Let $\mathbf{q} = (q_1, \ldots, q_m)$ be a non-empty sequence of queries. Every query $q_i \in \mathbf{q}$ specifies a pair of nodes u_i, v_i. The query pattern leakage function $\mathcal{L}_{QP}(\mathbf{q})$ returns an $m \times m$ (symmetric) matrix with entry i, j equals $Sim(q_i, q_j)$. Note that $\mathcal{L}_{QP}$ does not leak the identities of the queried nodes.*

We do not claim that it is always reasonable for a graph encryption scheme to leak the query pattern - it may convey sensitive information in some settings. Furthermore,

Definition 3.4.2 does not attempt to capture all possible leakages. As with many similar definitions, it does not capture side channels, and, furthermore, it does not capture leakage resulting from the client's behavior given the query answers, which, in turn may be affected by the choice of an approximation algorithm (see also [Feigenbaum et al., 2006, Halevi et al., 2001] for a discussion of privacy of approximation algorithms).

3.4.2.3 Sketch pattern leakage

Our third construction, $\mathsf{GraphEnc}_3$, leaks the query pattern and an additional pattern we refer to as the *sketch pattern*. The sketch pattern reveals which seeds are shared between the different sketches of the oracle and the size of the sketches. We formalize this below by revealing randomized "pseudo-ids" of the seeds in each sketch.

Definition 3.4.4 (Sketch pattern leakage)**.** *The sketch pattern leakage function $\mathcal{L}_{SP}(\Omega_G, q)$ for a graph G and a query $q = (u, v)$ is a pair (X, Y), where $X = \{f(w) : (w, \delta) \in \mathsf{Sk}_u\}$ and $Y = \{f(w) : (w, \delta) \in \mathsf{Sk}_v\}$ are* multi-sets *and f is a uniformly random from the family of functions: $\{0,1\}^{\log n} \rightarrow \{0,1\}^{\log n}$.*

It is not clear what this leakage implies in practice but we note that the leakage is not (directly) over the graph but over the *sketches* which contain a random subset of nodes. Therefore, it may be possible to add some form of noise in the sketches (e.g., using fake sketch elements) to guarantee some level of privacy to the original graph. We note that leakage is revealed in all SSE constructions such as [Song et al., 2000, Chang and Mitzenmacher, 2005, Goh, 2003, Curtmola et al., 2006, Chase and Kamara, 2010, Kamara et al., 2012, Kurosawa and Ohtaki, 2012, Kamara and Papamanthou, 2013, Cash et al., 2013a, Naveed et al., 2014, Cash et al., 2014]. However, in all these constructions the leakage is over a data structure (e.g., an inverted index) that holds *all* of the original data (i.e., all the keywords and documents). In our case, the leakage is over a structure that holds only a random subset of the data. This could provide additional help with respect to privacy

but this is a topic for future work and is not the main focus of this paper.

3.4.3 Efficiency

We evaluate the efficiency and practicality of our constructions according to the following criteria:

- *Setup time:* the time for the client to pre-process and encrypt the graph;
- *Space complexity:* the size of the encrypted graph;
- *Query time:* The time to execute a shortest distance query on the encrypted graph;
- *Communication complexity:* the number of bits exchanged during a query operation.

3.5 GRECS Constructions

In this section, we describe our three oracle encryption schemes. The first scheme, $\mathsf{GraphEnc}_1$, is computationally efficient, but has high communication overhead. Our second scheme, $\mathsf{GraphEnc}_2$, is communication efficient but has high space overhead. Our third scheme, $\mathsf{GraphEnc}_3$, is computationally efficient with optimal communication complexity. $\mathsf{GraphEnc}_1$ and $\mathsf{GraphEnc}_2$ do not leak anything besides the Query Pattern, and $\mathsf{GraphEnc}_3$ also leaks the Sketch Pattern.

3.5.1 A Computationally-Efficient Scheme

We now describe our first scheme which is quite practical. The scheme, described below, makes use of symmetric-key primitives which results in a simple and very efficient construction. The scheme $\mathsf{GraphEnc}_1 = (\mathsf{Setup}, \mathsf{distQuery})$ makes use of a symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a PRP P. The Setup algorithm works as follows. Given a $1^\lambda, \Omega_G$, α and ε:

- It pads each sketch to the maximum sketch size S by filling them with dummy values (S can be determined after all the sketches are computed).

- It then generates keys K_1, K_2 for the encryption scheme and PRP respectively and sets $K = (K_1, K_2)$. For all $v \in V$, it computes a label $P_{K_2}(v)$ and creates an encrypted sketch $\mathsf{ESk}_v = (c_1, ..., c_\lambda)$, where $c_i \leftarrow \mathsf{Enc}_{K_1}(w_i \| \delta_i)$ is a symmetric-key encryption of the ith pair (w_i, δ_i) in Sk_v.
- It then sets up a dictionary DX in which it stores, for all $v \in V$, the pairs $(P_{K_2}(v), \mathsf{ESk}_v)$, ordered by the labels. The encrypted graph is then simply $\mathsf{EO} = \mathsf{DX}$.

The $\mathsf{distQuery}$ protocol works as follows. To query EO on $q = (u, v)$, the client sends a token $\mathsf{tk} = (\mathsf{tk}_1, \mathsf{tk}_2) = (P_{K_2}(u), P_{K_2}(v))$ to the server which returns the pair $\mathsf{ESk}_u := \mathsf{DX}[\mathsf{tk}_1]$ and $\mathsf{ESk}_v := \mathsf{DX}[\mathsf{tk}_2]$. The client then decrypts each encrypted sketch and computes $\min_{s \in \mathbf{I}} \mathsf{dist}(u, s) + \mathsf{dist}(s, v)$ (note that the algorithm only needs the sketches of the nodes in the query).

3.5.2 Security and efficiency.

It is straightforward to see that the scheme is adaptively $(\mathcal{L}, \mathcal{L}_{QP})$-semantically secure, where $\mathcal{L}$ is the function that returns n, S and D. *Proof Sketch:* Consider the simulator $\mathcal{S}$ that works as follows. Given leakage $\mathcal{L}_{\mathsf{Setup}}(\Omega_G) = (S, D)$, it starts by generating $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SWHE.Gen}(1^\lambda)$. For all $1 \leq i \leq n$, it then samples $\ell_i \xleftarrow{\$} \{0,1\}^{\log n}$ without repetition and sets $\mathsf{DX}[\ell_i] := \mathtt{T}_i$, where $\mathtt{T}_i$ is an array that holds $t = 2 \cdot S_m^2 \cdot \varepsilon^{-1}$ homomorphic encryptions of $0 \in 2^N$, where $N = 2 \cdot D + 1$. It outputs $\mathsf{EO} = \mathsf{DX}$. Given leakage $\mathcal{L}_{\mathsf{Query}}(\Omega_G, q) = \mathcal{L}_{QP}(\Omega_G, q)$ it checks if either of the query nodes u or v appeared in any previous query. If u appeared previously, $\mathcal{S}$ sets tk_1 to the value that was previously used. If not, it sets $\mathsf{tk}_1 := \ell_i$ for some previously unused ℓ_i. It does the same for the query node v; that is, it sets tk_2 to be the previously used value if v was previously queried or to an unused ℓ_i if it was not. The theorem follows from the pseudo-randomness of P and the CPA-security of the symmetric encryption scheme. ■

The communication complexity of the $\mathsf{distQuery}$ protocol is linear in S, where S is the maximum sketch size. Note that even though S is sub-linear in n, it could still be large

in practice. For example, in the Das Sarma *et al.* construction $S = O(n^{2/\alpha} \cdot \log n)$. Also, in the case of multiple concurrent queries, this could be a significant bottleneck for the scheme.

In the following Section, we show how to achieve a solution with $O(1)$ communication complexity and in Section 3.6 we experimentally show that it scales to graphs with millions of nodes.

3.5.3 A Communication-Efficient Scheme

We now describe our second scheme $\mathsf{GraphEnc}_2 = (\mathsf{Setup}, \mathsf{distQuery})$ which is less computationally efficient than our first but is optimal with respect to communication complexity.

Algorithm 1: Setup algorithm for $\mathsf{GraphEnc}_2$

Input : $1^\lambda, \Omega_G, \alpha, \varepsilon$
Output: EO

1. **begin** Setup
2. Sample $K \xleftarrow{\$} \{0,1\}^k$;
3. Initialize a dictionary DX;
4. Generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SWHE.Gen}(1^\lambda)$;
5. Set $S := \max_{v \in V} |\mathsf{Sk}_v|$;
6. Set $D := \max_{v \in V} \left\{ \max_{(w,\delta) \in \mathsf{Sk}_v} \delta \right\}$;
7. Set $N := 2 \cdot D + 1$ and $t = 2 \cdot S^2 \cdot \varepsilon^{-1}$;
8. Sample a hash function $h : V \rightarrow [t]$ from $\mathcal{H}$;
9. **foreach** $v \in V$ **do**
10. compute $\ell_v := P_K(v)$;
11. initialize an array T_v of size t;
12. **foreach** $(w_i, \delta_i) \in \mathsf{Sk}_v$ **do**
13. set $\mathsf{T}_v[h(w_i)] \leftarrow \mathsf{SWHE.Enc}_{\mathsf{pk}}(2^{N-\delta_i})$;
14. fill remaining cells of T_v with encryptions of 0; set $\mathsf{DX}[\ell_v] := \mathsf{T}_v$;
15. Output K and $\mathsf{EO} = \mathsf{DX}$

The details of the construction are given in Algorithms 1 and 2. It makes use of a SWHE scheme $\mathsf{SWHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, a pseudo-random permutation P and a family of universal hash functions $\mathcal{H}$.

The Setup algorithm works as follows. Given 1^λ, Ω_G, α, and ε as inputs, it generates

a public/secret-key pair $(\mathsf{pk}, \mathsf{sk})$ for SWHE. Let D be the maximum distance over all the sketches and S be the maximum sketch size. Setup sets $N := 2 \cdot D + 1$ and samples a hash function $h \xleftarrow{\$} \mathcal{H}$ with domain V and co-domain $[t]$, where $t = 2 \cdot S^2 \cdot \varepsilon^{-1}$.

It then creates a hash table for each node $v \in V$. More precisely, for each node v, it processes each pair $(w_i, \delta_i) \in \mathsf{Sk}_v$ and stores $\mathsf{Enc}_{\mathsf{pk}}(2^{N-\delta_i})$ at location $h(w_i)$ of a t-size array T_v. In other words, for all $v \in V$, it creates an array T_v such that for all $(w_i, \delta_i) \in \mathsf{Sk}_v$, $\mathsf{T}_v[h(w_i)] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(2^{N-\delta_i})$. It then fills the empty cells of T_v with homomorphic encryptions of 0 and stores each hash table T_{v_1} through T_{v_n} in a dictionary DX by setting, for all $v \in V$, $\mathsf{DX}[P_K(v)] := \mathsf{T}_v$. Finally, it outputs DX as the encrypted oracle EO.

Fig. 3.2 below provides an example of one of the hash tables T_v generated from a sketch $\mathsf{Sk}_v = \{(w_1, \delta_1), \ldots, (w_s, \delta_s)\}$, where s is the size of the sketch. For all $i \in [s]$, the ciphertext $\mathsf{Enc}_{\mathsf{pk}}(2^{N-\delta_i})$ is stored at location $h(w_i)$ of the table T_v. For example, we place $\mathsf{Enc}_{\mathsf{pk}}(2^{2-\delta_j})$ to $\mathsf{T}_v[h(w_j)]$ since $h(w_j) = 1$. Finally, all remaining locations of T_v are filled with SWHE encryptions of 0. Notice that, since we are using probabilistic encryption, the encryptions of 0 are different, and are indistinguishable from the encryptions of the other values.

	0	$h(w_i)$	2		$h(w_j)$	$h(w_k)$	t-1
T_v	$Enc_{pk}(0)$	$Enc_{pk}(2^{N-\delta_i})$	$Enc_{pk}(0)$	......	$Enc_{pk}(2^{N-\delta_j})$	$Enc_{pk}(2^{N-\delta_k})$	$Enc_{pk}(0)$

Figure 3.2: One node's encrypted hash table.

The $\mathsf{distQuery}$ protocol works as follows. Given a query $q = (u, v)$, the client sends tokens $(\mathsf{tk}_1, \mathsf{tk}_2) = (P_K(u), P_K(v))$ to the server which uses them to retrieve the hash tables of nodes u and v by computing $\mathsf{T}_u := \mathsf{DX}[\mathsf{tk}_1]$ and $\mathsf{T}_v := \mathsf{DX}[\mathsf{tk}_2]$. The server then homomorphically evaluates an inner product over the hash tables. More precisely, it computes $c := \sum_{i=1}^{t} \mathsf{T}_u[i] \cdot \mathsf{T}_v[i]$, where $\sum$ and $\cdot$ refer to the homomorphic addition and multiplication operations of of the SWHE scheme. Finally, the server returns only c to the client who decrypts it and outputs $2N - \log_2(\mathsf{Dec}_{\mathsf{sk}}(c))$.

Note that the storage complexity at the server is $O(n \cdot t)$ and the communication

Algorithm 2: DistQuery algorithm for $\mathsf{GraphEnc}_2$

Input : Client's input is (K, q) and server's input is EO.
Output: Client's output is dist_q and server's output is $\perp$.

1 **begin** distQuery
2 C: client parses q as (u, v);
3 $C \Rightarrow S$: client sends $\mathsf{tk} = (\mathsf{tk}_1, \mathsf{tk}_2) = (P_K(u), P_K(v))$;
4 S: server retrieves $\mathsf{T}_1 := \mathsf{DX}[\mathsf{tk}_1]$ and $\mathsf{T}_2 := \mathsf{DX}[\mathsf{tk}_2]$;
5 **foreach** $i \in [t]$ **do**
6 Server computes $c_i \leftarrow \mathsf{SWHE.Eval}(\times, \mathsf{T}_1[i], \mathsf{T}_2[i])$;
7 $S \Rightarrow C$: server sends $c \leftarrow \mathsf{SWHE.Eval}(+, c_1, \ldots, c_t)$;
8 C: client computes $m \leftarrow \mathsf{SWHE.Dec}_{\mathsf{sk}}(c)$;
9 C: client outputs $\mathsf{dist} = 2N - \log m$.

complexity of distQuery is $O(1)$ since the server only returns a single ciphertext. In Section 3.5.3.1, we analyze the correctness and security of the scheme.

Remark. The reason we encrypt $2^{N-\delta_i}$ as opposed to δ_i is to make sure we can get the minimum sum over the distances from the sketches of both u and v. Our observation is that $2^x + 2^y$ is bounded by $2^{\max(x,y)+1}$. As we show Theorem 3.5.2, this approach does not, with high probability, affect the approximation factor from what the underlying distance oracle give us.

Instantiating & optimizing the SWHE scheme. For our experiments (see Section 3.6) we instantiate the SWHE scheme with the BGN construction of [Boneh et al., 2005]. We choose BGN due to the efficiency of its encryption algorithm and the compactness of its ciphertexts and keys (as compared to the lattice-based construction of [Gentry et al., 2010]). Unfortunately, the BGN *decryption* algorithm is expensive as it requires computations of discrete logarithms. To improve this, we make use of various optimizations. In particular, we compute discrete logs during decryption using the Baby step Giant step algorithm [Shanks, 1971] and use a pre-computed table to speed up the computation. More precisely, recall that decryption in BGN requires solving an equation of the form $c^{\mathsf{sk}} = (g^{\mathsf{sk}})^m$, where c is the ciphertext, sk is the secret key and g is the generator

of the underlying group. The messages m we need to decrypt are from a bounded domain $[2^{2N}]$. Here, the value of N corresponds, roughly, to the diameter of the graph. During the pre-computation phase, we set $x = \lceil\sqrt{2^{2N}}\rceil$ and pre-compute g^j for all $j \in [x]$, storing them in a lookup-table. During decryption, we first compute $c \cdot g^{-ix}$ for all $i \in [x]$ and store them in another table. We then compare it with each element of the first look-up table (i.e., for all $j \in [x]$). We set $m = ix + j$ if there is a match, otherwise we return $\perp$.

3.5.3.1 Correctness

Here, we analyze the correctness of $\mathsf{GraphEnc}_2$. We first bound the collision probability of our construction and then proceed to prove correctness in Theorem 3.5.2 below.

Lemma 3.5.1. *Let $q = (u, v)$ be a shortest distance query and let E_q be the event that a collision occurred in the* Setup *algorithm while constructing the hash tables T_u and T_v. Then,* $\Pr[\,\mathsf{E}_q\,] \leq 2 \cdot \frac{S^2}{t}$.

Proof Sketch: Let Coll_v be the event that at least one collision occurs while creating v's hash table T_v (i.e., in Algorithm 1 Setup Line 13). Also, let $\mathsf{XColl}_{u,v}$ be the event that there exists at least one pair of distinct nodes $w_u \in \mathsf{Sk}_u$ and $w_v \in \mathsf{Sk}_v$ such that $h(w_u) = h(w_v)$. For any $q = (u, v)$, we have

$$\Pr[\,\mathsf{E}_q\,] \leq \Pr[\,\mathsf{Coll}_u\,] + \Pr[\,\mathsf{Coll}_v\,] + \Pr[\,\mathsf{XColl}_{u,v}\,]. \tag{3.1}$$

Let s_u be the size of Sk_u an s_v be the size of Sk_v. Since there are $\binom{s_u}{2}$ and $\binom{\mathsf{s}_v}{2}$ node pairs in Sk_u and Sk_v, respectively, and each pair collides under h with probability at most $1/t$, $\Pr[\,\mathsf{Coll}_u\,] \leq \frac{s_u^2}{2 \cdot t}$ and $\Pr[\,\mathsf{Coll}_v\,] \leq \frac{s_v^2}{2 \cdot t}$. On the other hand, if $\mathbf{I}$ is the set of common nodes in Sk_u and Sk_v, then

$$\Pr[\,\mathsf{XColl}_{u,v}\,] \leq \frac{(s_u - |\mathbf{I}|)(s_v - |\mathbf{I}|)}{t}$$

Recall that $s_u = s_v \leq S$, so by combining with Eq. 3.1, we have $\Pr[\,\mathsf{E}_q\,] \leq 2 \cdot \frac{S^2}{t}$. ■

Note that in practice "intra-sketch" collision events Coll_u and Coll_v may or may not affect the correctness of the scheme. This is because the collisions could map the SWHE encryptions to locations that hold encryptions of 0 in other sketches. This means that at query time, these SWHE encryptions will not affect the inner product operation since they will be canceled out. Inter-sketch collision events $\mathsf{XColl}_{u,v}$, however, may affect the results since they will cause different nodes to appear in the intersection of the two sketches and lead to an incorrect sum.

Theorem 3.5.2. *Let $G = (V, E)$, $\alpha \geq 1$ and $\varepsilon < 1$. For all $q = (u, v) \in V^2$ with $u \neq v$,*

$$\Pr\left[\, \alpha \cdot \mathsf{dist}(u, v) - \log|\mathbf{I}| \leq d \leq \alpha \cdot \mathsf{dist}(u, v) \,\right] \geq 1 - \varepsilon,$$

where $(d, \bot) := \mathsf{GraphEnc_2.distQuery}\big((K, q), \mathsf{EO}\big)$, $(K, \mathsf{EO}) \leftarrow \mathsf{GraphEnc_2.Setup}(1^\lambda, \Omega_G, \alpha, \varepsilon)$, *and* $\mathbf{I}$ *is the number of common nodes between* Sk_u *and* Sk_v.

Proof Sketch: Let $\mathbf{I}$ be the set of nodes in common between Sk_u and Sk_v and let $\mathsf{mindist} = \min_{w_i \in \mathbf{I}}\{\delta_i^u + \delta_i^v\}$, where for all $0 \leq i \leq |\mathbf{I}|$, $\delta_i^u \in \mathsf{Sk}_u$ and $\delta_i^v \in \mathsf{Sk}_v$. Note that at line 7 in Algorithm 2 $\mathsf{distQuery}$, the server returns to the client $c = \sum_{i=1}^{t} \mathsf{T}_u[i] \cdot \mathsf{T}_v[i]$.

Let E_q be the event a collision occurred during Setup in the construction of the hash tables T_u and T_v of u and v respectively. Conditioned on $\overline{\mathsf{E}_q}$, we therefore have that

$$\begin{aligned} c &= \sum_{i=1}^{|\mathbf{I}|} \mathsf{Enc_{pk}}(2^{N-\delta_i^u}) \cdot \mathsf{Enc_{pk}}(2^{N-\delta_i^v}) \\ &= \mathsf{Enc_{pk}}\big(2^{2N} \cdot \Sigma_{i=1}^{|\mathbf{I}|} 2^{-(\delta_i^u + \delta_i^v)}\big), \end{aligned}$$

where the first equality holds since for any node $w_i \notin \mathbf{I}$, one of the homomorphic encryptions $\mathsf{T}_u[i]$ or $\mathsf{T}_v[i]$ is an encryption of 0. It follows then that (conditioned on $\overline{\mathsf{E}_q}$) at Step 9 the client outputs

$$d = 2N - \log\left(2^{2N} \cdot \Sigma_{i=1}^{|\mathbf{I}|} 2^{-(\delta_i^u + \delta_i^v)}\right)$$

$$\begin{aligned} &\leq 2N - \log\left(2^{2N - \mathsf{mindist}}\right) \\ &\leq \mathsf{mindist}, \end{aligned}$$

where the first inequality holds since $\mathsf{mindist} \leq (\delta_i^u + \delta_i^v)$ for all $i \in |\mathbf{I}|$. Towards showing a lower bound on d note that

$$\begin{aligned} d &= 2N - \log\left(2^{2N} \cdot \Sigma_{i=1}^{|\mathbf{I}|} 2^{-(\delta_i^u + \delta_i^v)}\right) \\ &\geq 2N - \log\left(2^{2N - \mathsf{mindist}} + |\mathbf{I}|\right) \\ &\geq \mathsf{mindist} - \log|\mathbf{I}|, \end{aligned}$$

where the first inequality also holds from $\mathsf{mindist} \leq (\delta_i^u + \delta_i^v)$ for all $i \in |\mathbf{I}|$. Now, by the (α, ε)-correctness of DO, we have that $\mathsf{mindist} \leq \alpha \cdot \mathsf{dist}(u, v)$ with probability at least $(1 - \varepsilon)$ over the coins of $\mathsf{DO.Setup}$. So, conditioned on $\overline{\mathsf{E}_q}$,

$$\mathsf{mindist} - \log|\mathbf{I}| \leq d \leq \alpha \cdot \mathsf{dist}(u, v).$$

The Theorem follows by combining this with Lemma 3.5.1 which bounds the probability of E_q and noting that Setup sets $t = 2 \cdot S^2 \cdot \varepsilon^{-1}$. ■

Space complexity. Note that to achieve (α, ε)-correctness, our construction produces encrypted sketches that are larger than the original sketches. More precisely, if the maximum sketch size of the underlying distance oracle is S, then the size of *every* encrypted sketch is $t = 2 \cdot S^2 \cdot \varepsilon^{-1}$, which is considerably larger. In Section 3.5.5, we describe a third construction which achieves better space efficiency at the cost of more leakage.

Remark on the approximation. Note that Theorem 3.5.2 also provides a lower bound of $\alpha \cdot \mathsf{dist}(u, v) - \log|\mathbf{I}|$ for the approximate distance. In particular, the bound depends on the set of common nodes $|\mathbf{I}|$ which varies for different queries but is small in practice. Furthermore, if $\log|\mathbf{I}|$ is larger than $\mathsf{mindist}$, the approximate distance returned could be

negative (we indeed observe a few occurrences of this in our experiments).

To improve the accuracy of the approximation, one could increase the base in the homomorphic encryptions. More precisely, instead of using encryptions of the form $\mathsf{Enc}_{\mathsf{pk}}(2^{N-\delta})$ we could use $\mathsf{Enc}_{\mathsf{pk}}(B^{N-\delta})$ for $B = 3$ or $B = 4$. This would result in an improved lower bound of $\mathsf{mindist} - \log_B |\mathbf{I}|$ but would also increase the homomorphic decryption time since this increases the message space which in turn adds overhead to the decryption algorithm. We leave it as an open problem to further improve this lower bound without increasing the message space.

Remark on error rate. Given the above analysis, a client that makes γ queries will have an error ratio of $\varepsilon \cdot \gamma$. In our experiments we found that, in practice, when using the Das Sarma *et al.* oracle, setting $\sigma \approx 3$ results in a good approximation. So if we fix $\sigma = 3$ and set $t = O(\sqrt{n})$, then the error rate is $O\big(\gamma \cdot \log^2(n)/\sqrt{n}\big)$ which decreases significantly as n grows. In the case of the Cohen *et al.* all-distance sketch, if we fix $\rho = 4$ and set $t = O(\sqrt{n})$, then we achieve about the same error rate $O\big(\gamma \cdot \ln^2(n)/\sqrt{n}\big)$. We provide in Section 3.6 detailed experimental result on the error rate.

3.5.4 Error Detection

We provide a method to detect those inter-collisions. In general, the collisions caused by the hash functions can be *detected* by associating with each encryption of a node a random value and its inverse value that are unique for each node. If two different nodes collide, the product of these values will be a random value, whereas if the same node is mapped to the same entry the product will give 1. More discussion about this technique will appear in the full version of this work. Speficically, for each node, we create two copies of the hash table, say DX and DX'. For each node $v \in V$, we produce the unique random value and it inverse for each w_i in v's sketch. This can be done by computing the PRF F under a different key $K' \xleftarrow{\$} \{0,1\}^\lambda$, i.e. $F_K(w_i)$. Specifically, for each $(w_i, \delta_i) \in \mathsf{Sk}_v$, initialize two arrays T_v and T'_v of size t. Next, we place $\mathsf{T}_v[h(w_i)] \leftarrow \Pi.\mathsf{Enc}(F_K(w_i) \cdot 2^{N-\delta_i})$ and

$\mathsf{T}'_v[h(w_i)] \leftarrow \Pi.\mathsf{Enc}(F_K^{-1}(w_i) \cdot 2^{N-\delta_i})$. In $\mathsf{DistQuery}$, if two different nodes collide due to the hash function then the product of these values will produce some random values, where if the same node is mapped to the same entry the product will give the correct result.

3.5.4.1 Security

In the following theorem, we analyze the security of $\mathsf{GraphEnc}_2$.

Theorem 3.5.3. *If P is pseudo-random and* SWHE *is CPA-secure then* $\mathsf{GraphEnc}_2$*, as described above, is adaptively $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}})$-semantically secure, where $\mathcal{L}_{\mathsf{Setup}}(\Omega_G) = (n, S, D)$ and $\mathcal{L}_{\mathsf{Query}}(\Omega_G, q) = \mathcal{L}_{QP}(\Omega_G, q)$.*

Proof Sketch: Consider the simulator $\mathcal{S}$ that works as follows. Given leakage $\mathcal{L}_{\mathsf{Setup}} = n$, for all $1 \leq i \leq n$, it samples $\ell_i \xleftarrow{\$} \{0,1\}^{\log n}$ (without repetition) and creates an array T_i of size t filled with homomorphic encryption of 0. It then creates a dictionary DX and sets $\mathsf{DX}[\ell_i] = \mathsf{T}_i$ for all $1 \leq i \leq n$. Finally, it outputs $\mathsf{EO} = \mathsf{DX}$. Given leakage $\mathcal{L}_{QP}(q)$, $\mathcal{S}$ first checks whether any of the two query nodes appeared in an earlier query. If the first query node appeared in a previous query, $\mathcal{S}$ sets tk_1 to its stored token. Otherwise $\mathcal{S}$ chooses a fresh token $\mathsf{tk}_1 \xleftarrow{\$} \{0,1\}^{\log n}$ and stores it. $\mathcal{S}$ proceeds similarly with token tk_2 and then sends $\mathsf{tk} = (\mathsf{tk}_1, \mathsf{tk}_2)$.

It now remains to show that the $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$ experiments will output 1 with negligibly-close probability. This can be done using the following sequence of 3 games:

- Game_0: this game corresponds exactly to a $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ experiment.
- Game_1: is the same as Game_0 except that the output of P is replaced with random $(\log n)$-bit strings. Clearly, the pseudo-randomness of P guarantees that

$$|\Pr[\mathsf{Game}_0 = 1] - \Pr[\mathsf{Game}_1 = 1]| \leq \mathsf{negl}(k).$$

- Game_2: is the same as Game_1 except that all the HE encryptions are replaced with

HE encryptions of 0. Clearly, it follows by the CPA-security of SWHE that

$$|\Pr[\mathsf{Game}_1 = 1] - \Pr[\mathsf{Game}_2 = 1]| \leq \mathsf{negl}(k).$$

Note that, by construction, Game_2 corresponds exactly to an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$ experiment so we have

$$\left|\Pr\left[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1\right]\right| \leq \mathsf{negl}(k)$$

from which the Theorem follows. The Theorem follows from the pseudo-randomness of P and the CPA-security of SWHE. ■

3.5.5 A Space-Efficient Construction

Although our second construction, $\mathsf{GraphEnc}_2$, achieves optimal communication complexity, it has two limitations. The first is that it is less computationally-efficient than our first construction $\mathsf{GraphEnc}_1$ both with respect to constructing the encrypted graph and to querying it. The second limitation is that its storage complexity is relatively high; that is, it produces encrypted graphs that are larger than the ones produced by $\mathsf{GraphEnc}_1$ by a factor of $2 \cdot S \cdot \varepsilon^{-1}$. These limitations are mainly due to the need to fill the hash tables with many homomorphic encryptions of 0. This also slows down the query algorithm since it has to homomorphically evaluate an inner product on two large tables.

To address this, we propose a third construction $\mathsf{GraphEnc}_3 = (\mathsf{Setup}, \mathsf{distQuery})$ which is both space-efficient and achieves $O(1)$ communication complexity. The only trade-off is that it leaks more than the two previous constructions.

The details of the scheme are given in Algorithms 3 and 4. At a high-level, the scheme works similarly to $\mathsf{GraphEnc}_2$ with the exception that the encrypted sketches do not store encryptions of 0's, i.e., they only store the node/distance pairs of the sketches constructed by the underlying distance oracle. Implementing this high-level idea is not straightforward, however, because simply removing the encryptions of 0's from the encrypted sketches/hash

Algorithm 3: Setup algorithm for $\mathsf{GraphEnc}_3$

Input : $1^\lambda, \Omega_G, \alpha, \varepsilon$
Output: EO

```
1  begin Setup
2     Sample K_1, K_2 <-$- {0,1}^λ;
3     Initialize a counter ctr = 1;
4     Let Z = Σ_{v∈V} |Sk_v|;
5     Sample a random permutation π over [Z];
6     Initialize an array Arr of size Z;
7     Initialize a dictionary DX of size n;
8     Generate (pk, sk) <- SWHE.Gen(1^λ);
9     Set S := max_{v∈V} |Sk_v|;
10    Set D := max_{v∈V} { max_{(w,δ)∈Sk_v} δ };
11    Set N := 2·D + 1 and t = 2·S^2·ε^{-1};
12    Initialize collision-resistant hash function h : V -> [t];
13    foreach v ∈ V do
14       sample K_v <- {0,1}^λ;
15       foreach (w_i, δ_i) ∈ Sk_v do
16          compute c_i <- SWHE.Enc_pk(2^{N-δ_i});
17          if i ≠ |Sk_v| then
18             Set N_i = ⟨h(w_i) || c_i || π(ctr + 1)⟩;
19          else
20             Set N_i = ⟨h(w_i) || c_i || NULL⟩;
21          Sample r_i <-$- {0,1}^λ;
22          Set Arr[π(ctr)] := ⟨N_i ⊕ H(K_v || r_i), r_i⟩;
23          Set ctr = ctr + 1 ;
24    foreach v ∈ V (in random order) do
25       Set DX[P_{K_1}(v)] := ⟨addr_Arr(h_v) || K_v⟩ ⊕ F_{K_2}(v)
26    Output K = (K_1, K_2, pk, sk) and EO = (DX, Arr);
```

tables reveals the size of the underlying sketches to the server which, in turn, leaks structural information about the graph. We overcome this technical difficulty by adapting a technique from [Curtmola et al., 2006] to our setting. Intuitively, we view the seed/distance pairs in each sketch Sk_v as a linked-list where each node stores a seed/distance pair. We then randomly shuffle all the nodes and place them in an array; that is, we place each node of each list at a random location in the array while updating the pointers so that the "logical" integrity of the lists are preserved (i.e., given a pointer to the head of a list we

can still find all its nodes). We then encrypt all the nodes with a per-list secret key.

The scheme makes use of a SWHE scheme $\mathsf{SWHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$, a pseudo-random permutation P, a pseudo-random function F, a random oracle H and a collision-resistant hash function h modeled as a random function

The Setup algorithm takes as input a security parameter k, an oracle Ω_G , an approximation factor α, and an error parameter $\varepsilon < 1$. As shown in Algorithm 3, it first initializes a counter $\mathsf{ctr} = 1$ and samples a random permutation π over the domain $[Z]$, where $Z = \sum_{v \in V} |\mathsf{Sk}_v|$. It then initializes an Z-size array $\mathtt{Arr}$. It proceeds to create an encrypted sketch ESk_v from each sketch Sk_v as follows. It first samples a symmetric key K_v for this sketch. Then for each seed/distance pair (w_i, δ_i) in Sk_v, it creates a linked-list node $\mathtt{N}_i = \langle h(w_i) \| c_i \| \pi(\mathsf{ctr} + 1) \rangle$, where $c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(2^{N - \delta_i})$, and stores an H-based encryption $\langle N_i \oplus H(K_v \| r_v), r_v \rangle$ of the node at location $\pi(\mathsf{ctr})$ in $\mathtt{Arr}$. For the last seed/distance pair, it uses instead a linked-list node of the form $\mathtt{N}_i = \langle h(w_i) \| c_i \| \mathsf{NULL} \rangle$, it then increments ctr.

Setup then creates a dictionary DX where it stores for each node $v \in V$, the pair $(P_{K_1}(v), \langle \mathsf{addr}_{\mathtt{Arr}}(\mathsf{h}_v) \| K_v \rangle \oplus F_{K_2}(v))$, where $\mathsf{addr}_{\mathtt{Arr}}(\mathsf{h}_v)$ is the location in $\mathtt{Arr}$ of the head of v's linked-list. Figure 3.3 provides a detailed example for how we encrypt the sketch. Suppose node u's sketch Sk_u has the element $(a, d_1), (b, d_2), (c, d_3)$. The locations $\mathsf{ind1}, \mathsf{ind2}, \mathsf{ind3}$ in $\mathtt{Arr}$ are computed according the random permutation π.

The $\mathsf{distQuery}$ protocol, which is shown in Algorithm 4, works as follows. Given a query $q = (u, v)$, the client sends tokens $(\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3, \mathsf{tk}_4) = (P_{K_1}(u), P_{K_1}(v), F_{K_2}(u), F_{K_2}(v))$ to the server which uses them to retrieve the values $\gamma_1 := \mathsf{DX}[\mathsf{tk}_1]$ and $\gamma_2 := \mathsf{DX}[\mathsf{tk}_2]$. The server computes $\langle a_1 || K_u \rangle := \gamma_1 \oplus \mathsf{tk}_3$ and $\langle b_1 || K_v \rangle := \gamma_2 \oplus \mathsf{tk}_4$. Next, it recovers the lists pointed to by a_1 and b_1. More precisely, starting with $i = 1$, it parses $\mathtt{Arr}[a_1]$ as $\langle \sigma_u, r_u \rangle$ and decrypts σ_u by computing $\langle h_i \| c_i \| a_{i+1} \rangle := \sigma_u \oplus H(K_u \| r_u)$ while $a_{i+1} \neq \mathsf{NULL}$. And starting with $j = 1$, it does the same to recover $\langle h'_j \| c'_j \| b_{j+1} \rangle$ while $b_{j+1} \neq \mathsf{NULL}$.

The server then homomorphically computes an inner product over the ciphertexts with the same hashes. More precisely, it computes $\mathsf{ans} := \sum_{(i,j):h_i = h'_j} c_i \cdot c'_j$, where $\sum$ and $\cdot$ refer to the homomorphic addition and multiplication operations of the SWHE scheme.

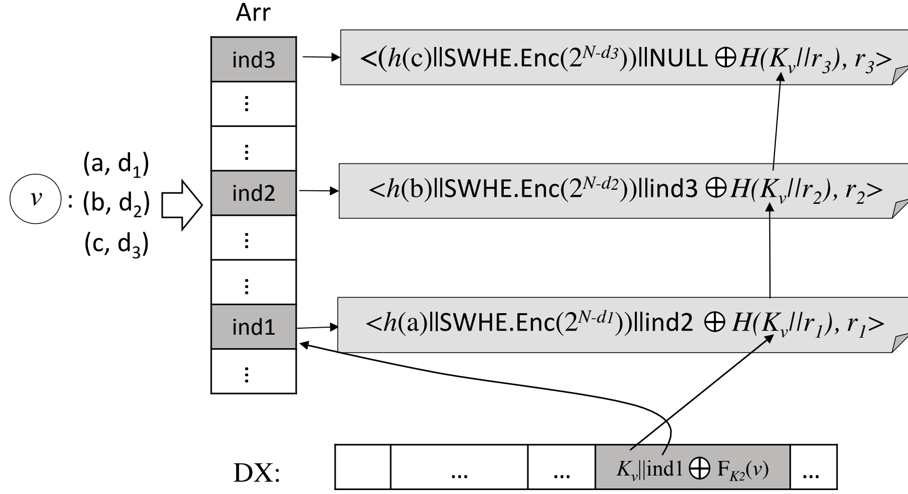


Figure 3.3: Example of encrypting $\mathsf{Sk}_u = \{(a, d_1), (b, d_2), (c, d_3)\}$.

Finally, the server returns only ans to the client which decrypts it and outputs $2N - \log_2(\mathsf{SWHE.Dec_{sk}}(\mathsf{ans}))$.

Note that the storage complexity at the server is $O(m + |V|)$ and the communication complexity of $\mathsf{distQuery}$ is still $O(1)$ since the server only returns a single ciphertext.

3.5.5.1 Correctness and Security

The correctness of $\mathsf{GraphEnc}_3$ follows directly from the correctness of $\mathsf{GraphEnc}_2$. To see why, observe that: (1) the homomorphic encryptions stored in the encrypted graph of $\mathsf{GraphEnc}_3$ are the same as those in the encrypted graph produced by $\mathsf{GraphEnc}_2$ with the exception of the encryptions of 0; and (2) the output d of the client results from executing the same homomorphic operations as in $\mathsf{GraphEnc}_2$, with the exception of the homomorphic sums with 0-encryptions.

We note that $\mathsf{GraphEnc}_3$ leaks only a little more than the previous constructions. With respect to setup leakage it reveals, in addition to (n, S, D), the total number of seeds Z. Intuitively, for a query $q = (u, v)$, the query leakage consists the query pattern leakage in addition to: (1) which seed/distance pairs in the sketches Sk_u and Sk_v are the same; and (2) the size of these sketches. This is formalized in Definition 3.4.4 as the sketch pattern

Algorithm 4: The protocol $\mathsf{distQuery}_{C,S}$.

Input : Client's input is $K, q = (u, v)$ and server's input is EO
Output: Client's output is d and server's output is $\perp$

```
1  begin distQuery
2      C: computes (tk_1, tk_2, tk_3, tk_4) = (P_{K_1}(u), P_{K_1}(v), F_{K_2}(u), F_{K_2}(v));
3      C ⇒ S: sends tk = (tk_1, tk_2, tk_3, tk_4);
4      S: computes γ_1 ← DX[tk_1] and γ_2 ← DX[tk_2];
5      if γ_1 = ⊥ or γ_2 = ⊥ then
6          exit and return ⊥ to the client
7      S: compute ⟨a_1||K_u⟩ := γ_1 ⊕ tk_3;
8      S: parse Arr[a_1] as ⟨σ_u, r_u⟩;
9      S: compute N_1 := σ_u ⊕ H(K_u‖r_u);
10     repeat
11         parse N_i as ⟨h_i‖c_i‖a_{i+1}⟩;
12         parse Arr[a_{i+1}] as ⟨σ_{i+1}, r_{i+1}⟩;
13         compute N_{i+1} := σ_{i+1} ⊕ H(K_u‖r_{i+1});
14         set i = i + 1;
15     until a_{i+1} = NULL;
16     S: compute ⟨b_1||K_v⟩ := γ_2 ⊕ tk_4;
17     S: parse Arr[b_1] as ⟨σ_v, r_v⟩;
18     S: compute N'_1 := σ_v ⊕ H(K_v‖r_v);
19     repeat
20         parse N'_j as ⟨h'_j‖c'_j‖b_{j+1}⟩;
21         parse Arr[b_{j+1}] as ⟨σ_{j+1}, r_{j+1}⟩;
22         compute N'_{j+1} := σ_{j+1} ⊕ H(K_v‖r_{j+1});
23         set j = j + 1;
24     until b_{j+1} = NULL;
25     S: set s := SWHE.Enc_pk(0);
26     foreach (N_i, N'_j) do
27         if h_i = h'_j then
28             compute p := SWHE.Eval(×, c_i, c'_j);
29             compute s := SWHE.Eval(+, s, p);
30     S ⇒ C: send s;
31     C: compute d := SWHE.Dec_sk(s)
```

leakage $\mathcal{L}_{SP}(\Omega_G, q)$. In the following Theorem, we summarize the security of $\mathsf{GraphEnc}_3$.

Theorem 3.5.4. *If P and F are pseudo-random, if SWHE is CPA-secure then $\mathsf{GraphEnc}_3$, as described above, is adaptively $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}})$-semantically secure in the random oracle model, where $\mathcal{L}_{\mathsf{Setup}}(\Omega_G) = (n, S, D, Z)$ and $\mathcal{L}_{\mathsf{Query}}(\Omega_G, q) = (\mathcal{L}_{QP}(\Omega_G, q), \mathcal{L}_{SP}(\Omega_G, q))$.*

Proof Sketch: Consider the simulator $\mathcal{S}$ that works as follows. Given leakage $\mathcal{L}_{\mathsf{Setup}} = (n, S, D, Z)$, for all $1 \leq i \leq Z$ it samples $\Gamma_i \xleftarrow{\$} \{0,1\}^{\log t + g(N) + \log Z + \lambda}$, where $g(\cdot)$ is the ciphertext expansion of SWHE, $t = 2 \cdot S^2 \cdot \varepsilon^{-1}$ and $N = 2 \cdot D + 1$. It then stores all the Γ_i's in a Z-element array $\mathtt{Arr}$. For all $1 \leq i \leq n$, it samples $\ell_i \xleftarrow{\$} \{0,1\}^{\log n}$ without repetition and sets $\mathsf{DX}[\ell_i] \xleftarrow{\$} \{0,1\}^{\log Z + k}$. Finally, it outputs $\mathsf{EO} = (\mathsf{DX}, \mathtt{Arr})$.

Given leakage $\mathcal{L}_{\mathsf{Query}}(G, q) = (\mathcal{L}_{QP}(G, q), \mathcal{L}_{SP}(G, q))$ such that $\mathcal{L}_{SP}(G, q) = (X, Y)$, $\mathcal{S}$ first checks if either of the query nodes u or v appeared in any previous query. If u appeared previously, $\mathcal{S}$ sets tk_1 and tk_3 to the values that were previously used. If not, it sets $\mathsf{tk}_1 := \ell_i$ for some previously unused ℓ_i and tk_3 as follows. It chooses a previously unused $\alpha \in [Z]$ at random, a key $K_u \xleftarrow{\$} \{0,1\}^k$ and sets $\mathsf{tk}_3 := \mathsf{DX}[\mathsf{tk}_1] \oplus \langle \alpha \| K_u \rangle$. It then remembers the association between K_u and X and the sketch size $|\mathsf{Sk}_u|$. It does the same for the query node v, sets tk_2 and tk_4 analogously and associates $|\mathsf{Sk}_v|$ and Y with the key K_v it chooses.

It simulates the random oracle H as follows. Given (K, r) as input, it checks to see if: (1) K has been queried before (in the random oracle); and (2) if any entry in $\mathtt{Arr}$ has the form $\langle s, r \rangle$ where s is a $(\log t + g(N) + \log Z)$-bit string. If K has not been queried before, it initializes a counter $\mathsf{ctr}_K := 0$. If an appropriate entry exists in $\mathtt{Arr}$, it returns $s \oplus \langle \gamma, c, p \rangle$, where γ is the ctr^{th} element of the multi-set X or Y associated with K, c is a SWHE encryption of 0 and p is an unused address in $\mathtt{Arr}$ chosen at random or $\emptyset$ if $\mathsf{ctr} = |\mathsf{Sk}|$, where $|\mathsf{Sk}|$ is the sketch size associated with K. If no appropriate entry exists in $\mathtt{Arr}$, $\mathcal{S}$ returns a random value. The Theorem then follows from the pseudo-randomness of P and F and the CPA-security of SWHE. ■

3.6 Experimental Evaluation

In this section, we present experimental evaluations of our schemes on a number of large-scale graphs. We implement the Das Sarma *et al.* distance oracle ($\mathbf{DO}_1$) and Cohen *et al.* distance oracle ($\mathbf{DO}_2$) and all three of our graph encryption schemes. We use AES-128 in

CBC mode for symmetric encryption and instantiate SWHE with the Boneh-Goh-Nissim (BGN) scheme, implemented in C++ with the Stanford Pairing-Based Library PBC[1]. We use OpenSSL[2] for all basic cryptographic primitives and use 128-bit security for all the encryptions. We use HMAC for PRFs and instantiate the hash function in $\mathsf{GraphEnc}_3$ with HMAC-SHA-256. All experiments were run on a 24-core 2.9GHz Intel Xeon, with 512 GBs of RAM running Linux.

3.6.1 Datasets

We use real-world graph datasets publicly available from the Stanford SNAP website[3]. In particular, we use *as-skitter*, a large Internet topology graph; *com-Youtube*, a large social network based on the Youtube web site; *loc-Gowalla*, a location-based social network; *email-Enron*, an email communication network; and *ca-CondMat*, a collaboration network for scientific collaborations between authors of papers related to Condensed Matter research. Table 3.1 summarizes the main characteristics of these datasets.

Dataset	Nodes	Edges	Diameter	Storage
as-skitter	1,696,415	11,095,298	25	143MB
com-Youtube	1,134,890	2,987,624	20	37MB
loc-Gowalla	196,591	950,327	14	11MB
email-Enron	36,692	367,662	11	1.84MB
ca-CondMat	23,133	186,936	14	158KB

Table 3.1: The graph datasets used in our experiments

Notice that some of these datasets contain millions of nodes and edges and that the diameters of these graphs are small. This is something that has been observed in many real-life graphs [Leskovec et al., 2005] and is true for expander and small-world graphs, which are known to model many real-life graphs. The implication of this, is that the maximum distance D in the sketches generated by the distance oracles is, in practice, small and therefore the value N that we use in $\mathsf{GraphEnc}_2$ and $\mathsf{GraphEnc}_3$ (see Algorithm 1 and 3) is

[1] http://crypto.stanford.edu/pbc/
[2] https://www.openssl.org/
[3] https://snap.stanford.edu/data/

typically small.

3.6.2 Overview

For a graph $G = (V, E)$ with n nodes, we summarize in Table 3.2 our constructions' space, setup, and communication complexities as well as the complexities for both the server and client during the query phase. Note that the complexities for each scheme also depend on α, however, in practice, since setting σ for $\mathbf{DO}_1$ (ρ for $\mathbf{DO}_2$) to some small numbers resulted good approximations, therefore, it makes $\alpha = O(\log n)$. In our experiments, we test different σ and ρ's and the sketch size, $|\mathsf{Sk}_v|$, for each node is sublinear in the size of the graph, i.e. $O(\log n)$.

Scheme	$\mathsf{GraphEnc}_1$	$\mathsf{GraphEnc}_2$	$\mathsf{GraphEnc}_3$
Space	$O(n \log n)$	$O(n \log^2 n/\varepsilon)$	$O(n \log n)$
Setup Time	$O(n \log n)$	$O(n \log^2 n/\varepsilon)$	$O(n \log n)$
Communication	$O(\log n)$	$O(1)$	$O(1)$
Server Query Comp.	$O(1)$	$O(\log^2 n/\varepsilon)$	$O(\log n)$
Client Query Comp.	$O(\log n)$	$O(diameter)$	$O(diameter)$

Table 3.2: The space, setup, communication, and query complexities of our constructions (α is set to be in $O(\log n)$).

Table 3.3 summarizes our experimental results. Compared to existing schemes, such as [Aly et al., 2013], our experiments shows that the constructions are very efficient and scalable for large real dataset. For example, in [Aly et al., 2013], it takes several minutes to securely compute the shortest path distance for graph with only tens to hundreds of nodes, whereas it takes only seconds for our scheme to query the encrypted graph up to 1.6 million nodes.

3.6.3 Performance of $\mathsf{GraphEnc}_1$

We evaluate the performance of $\mathsf{GraphEnc}_1$ using both the Das Sarma *et al.* and Cohen *et al.* distance oracles. For the Das Sarma *et al.* oracle ($\mathbf{DO}_1$), we set the sampling parameter $\sigma = 3$ and for the Cohen *et al.* oracle ($\mathbf{DO}_2$) we set the rank parameter $\rho = 4$. We choose

Dataset	sketch size S	Graph Sketching Scheme	$\mathsf{GraphEnc}_1$ Comm. per query (in bytes)	$\mathsf{GraphEnc}_1$ Setup Time per node (in ms)	$\mathsf{GraphEnc}_1$ Size per node (in KBs)	$\mathsf{GraphEnc}_2$ T size	$\mathsf{GraphEnc}_2$ Comm. per query (in bytes)	$\mathsf{GraphEnc}_2$ Setup Time per node (in secs)	$\mathsf{GraphEnc}_2$ Size per node (in MBs)	$\mathsf{GraphEnc}_3$ Comm. per query (in bytes)	$\mathsf{GraphEnc}_3$ Setup Time per node (in ms)	$\mathsf{GraphEnc}_3$ Size per node (in KBs)
As-skitter	80	$\mathbf{DO}_1$	3,840	16.7	1.94	11K	34	7.3	1.1	34	20.1	1.91
	71	$\mathbf{DO}_2$	3,120	14	1.63	8.4K	34	6.59	0.76	34	16	1.83
Youtube	80	$\mathbf{DO}_1$	3,840	16.5	1.94	10K	34	8	1.1	34	18.2	1.91
	68	$\mathbf{DO}_2$	3,120	14.5	1.63	8.5K	34	6.57	0.76	34	17.3	1.7
Gowalla	70	$\mathbf{DO}_1$	3360	14.9	1.7	7.5K	34	7.4	0.82	34	15.6	1.71
	53	$\mathbf{DO}_2$	2544	12	1.29	7K	34	5	0.62	34	14.7	1.41
Enron	60	$\mathbf{DO}_1$	2880	12.5	1.44	7K	34	5.6	0.76	34	14	1.48
	45	$\mathbf{DO}_2$	2160	9.39	1.11	6.5K	34	4.81	0.53	34	10	1.25
CondMat	55	$\mathbf{DO}_1$	2640	11.8	1.34	5.5K	34	4.65	0.65	34	13.2	1.31
	42	$\mathbf{DO}_2$	2016	7.8	1.03	5K	34	3.8	0.49	34	8.2	1.21

Table 3.3: A full performance summary for $\mathsf{GraphEnc}_1$, $\mathsf{GraphEnc}_2$, and $\mathsf{GraphEnc}_3$

these parameters because they resulted in good approximation ratios and the maximum sketch sizes (i.e., S) of roughly the same amount. Note that, the approximation factor α in those then is in $O(\log n)$ for $\mathsf{GraphEnc}_1$, therefore, the communication complexity (see Table 3.2) in $\mathsf{GraphEnc}_1$ is $O(\log n)$. We can see from Table 3.3 that the time to setup an encrypted graph with $\mathsf{GraphEnc}_1$ is practical—even for large graphs. For example, it takes only 8 hours to setup an encryption of the *as-skitter* graph which includes 1.6 million nodes. Since the $\mathsf{GraphEnc}_1.\mathsf{Setup}$ is highly-parallelizable, we could speed setup time considerably by using a cluster. A cluster of 10 machines would be enough to bring the setup time down to less than an hour. Furthermore, the size of the encrypted sketches range from 1KB for *CondMat* to 1.94KB for *as-skitter* per node. The main limitation of this construction is that the communication is proportional to the size of the sketches. We tested for various sketch sizes, and the communication per query went up to 3.8KB for *as-skitter* when we set $S = 80$. This can become quite significant if the server is interacting with multiple clients.

3.6.4 Performance of $\mathsf{GraphEnc}_2$

The first column in Table 3.3 of the $\mathsf{GraphEnc}_2$ experiments gives the size the encrypted hash tables T_v constructed during $\mathsf{GraphEnc}_2.\mathsf{Setup}$. Table sizes range from 5K for *ca-CondMat* to $11K$ for *as-skitter*.

The Time column gives the time to create an encrypted hash-table/sketch per node. This includes generating the BGN encryptions of the distances and the 0-encryptions. Note

that this makes $\mathsf{GraphEnc}_2.\mathsf{Setup}$ quite costly, about 3 orders of magnitude more expensive than $\mathsf{GraphEnc}_1.\mathsf{Setup}$. This is mostly due to generating the 0-encryptions. Note, however, that similarly to $\mathsf{GraphEnc}_1$, we can use extensive parallelization to speed up the setup. For example, using a cluster of 100 machines, we can setup the encrypted graph on the order of hours, even for *as-skitter* which includes 1.6 million nodes. The space overhead per node is also large, but the encrypted graph itself can be distributed in a cluster since every encrypted sketch is independent of the other. Finally, as shown in Table 3.3, $\mathsf{GraphEnc}_2$ achieves a constant communication cost of 34B.

In Fig. 3.4, we report on the intra- and inter-collisions that we observed when executing over 10K different queries over our data sets. The collision probability ranges between 1% and 3.5%. As we can see from the results, the oracle $\mathbf{DO_2}$ has less collisions than $\mathbf{DO}_1$.

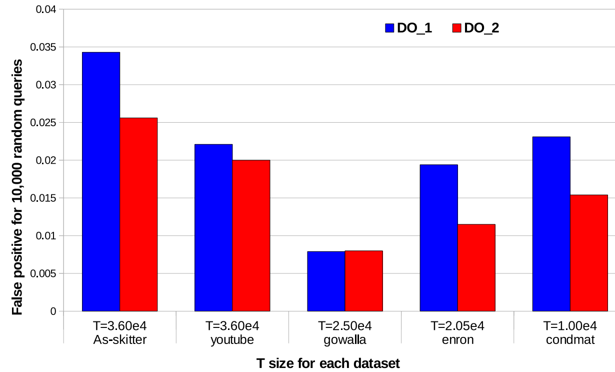


Figure 3.4: Collision probabilities for different datasets

3.6.5 Performance of $\mathsf{GraphEnc}_3$

The $\mathsf{GraphEnc}_3$ columns in Table 3.3 show that $\mathsf{GraphEnc}_3$ is as efficient as $\mathsf{GraphEnc}_1$ in terms of setup time and encrypted sketch size. Moreover, it achieves $O(1)$ communication of 34B like $\mathsf{GraphEnc}_2$. Using a single machine, $\mathsf{GraphEnc}_3.\mathsf{Setup}$ took less than 10 hours to encrypt *as-skitter*, but like the other schemes, it is highly parallelizable, and this could be brought down to an hour using 10 machines. We instantiated the hash function h using a cryptographic keyed hash function HMAC-SHA-256.

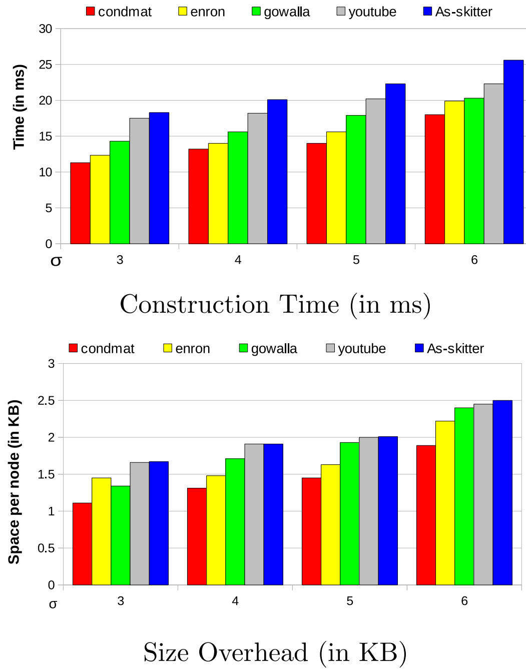
3.6.5.1 Construction time & encrypted sketch size

Since the performance of $\mathsf{GraphEnc}_3$ depends only on the size of the underlying sketches we investigate the relationship between the performance of $\mathsf{GraphEnc}_3.\mathsf{Setup}$ and the sampling and rank parameters of the Das Sarma *et al.* and Cohen *et al.* oracles, respectively. We use values of σ and ρ ranging from 3 to 6 in each case which resulted in maximum sketch sizes S ranging from 43 to 80. Figure 3.5 and Figure 3.6 give the construction time and size overhead of an encrypted sketch when using the Das Sarma *et al.* oracle and Cohen *et al.* oracle respectively.

In each case, the construction time scales linearly when σ and ρ increase. Also, unlike the previous schemes, $\mathsf{GraphEnc}_3$ produces encrypted sketches that are compact since it does not use 0-encryptions for padding purposes.

3.6.5.2 Query Time

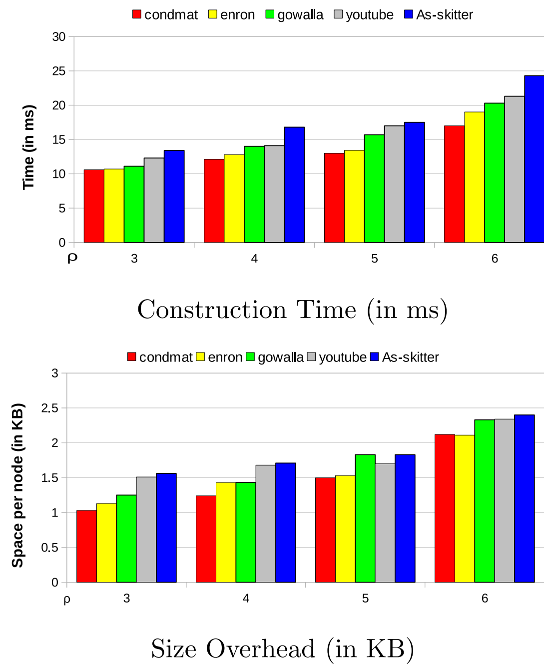
We measured the time to query an encrypted graph as a function of the oracle sampling/rank parameter. The average time at the server (taken over 10K random queries) is given in Figure 3.7 for all our graphs and using both distance oracles. The variance of the queries is within 0.5 ms. In general, the results show that query time is very fast and practical. For *as-skitter*, the query time ranges from 6.1 to 10 milliseconds with the Das Sarma *et al.* oracle and from 5.6 to 10 milliseconds with the Cohen *et al.* oracle. Query time is dominated by the homomorphic multiplication operation of the BGN scheme. But the number of multiplications only depends on the number of common seeds from the two encrypted sketches and, furthermore, these operations are independent so they can be parallelized. We note that we use mostly un-optimized implementations of all the underlying primitives and we believe that a more careful implementation (e.g., faster pairing library) would reduce the query time even further. We also measure the decryption time at the client. As pointed out previously, decryption time depends on N which itself is a function of the diameter of the graph. Since all our graphs have small diameter, client de-



Construction Time (in ms)

Size Overhead (in KB)

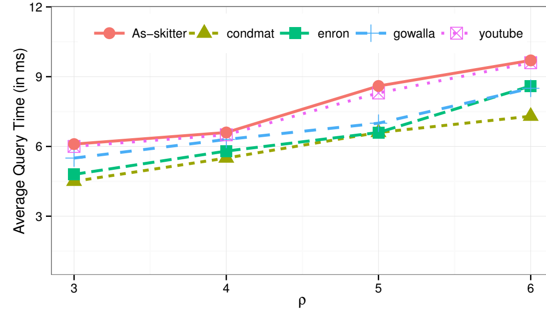
Figure 3.5: Construction time and size overhead ($\mathbf{DO}_1$)



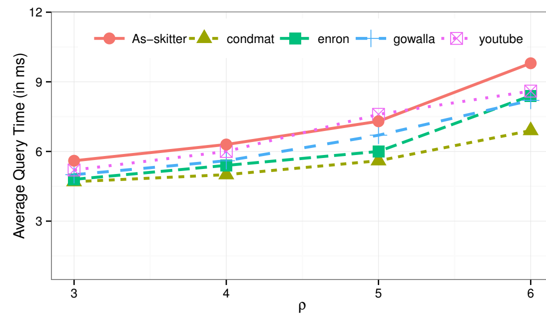
Construction Time (in ms)

Size Overhead (in KB)

Figure 3.6: Construction time and size overhead ($\mathbf{DO}_2$)



(a) Query Time (in ms) using $\mathbf{DO}_1$



(b) Query Time (in ms) $\mathbf{DO}_2$

Figure 3.7: Average Query time

cryption time—which itself consists of a BGN decryption— was performed very efficiently. In particular, the average decryption time was less than 4 seconds and in most cases the decryption ranged between 1 and 3 seconds.

Finally, we would like to mention that there is some additional information that is leaked. In our construction, we leak the parameter ρ and σ that are related to the size of the encrypted graph and this may leak some information about how "hard" it is to approximate the shortest distance values for the particular graph at hand. Also, the time that it takes to estimate the final result at the client may reveal the diameter of the graph, since it is related to the N and the max distance in the sketches.

3.6.6 Approximation errors

We investigate the approximation errors produced by our schemes. We generate 10K random queries and run the $\mathsf{Query}_{C,S}$ protocol. For client decryption, we recover $2N - \log m$

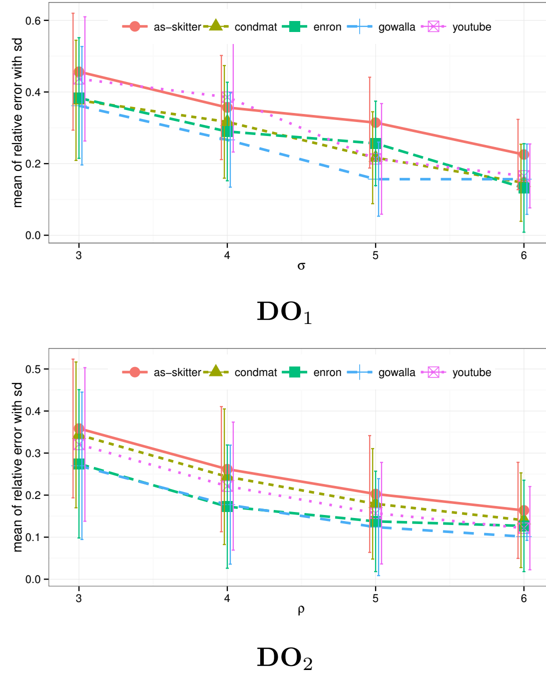


Figure 3.8: Mean of Estimated Error with Standard Deviation

and round it to its floor value. We used breadth-first search (BFS) to compute the exact distances between each pair of nodes and we compare the approximate distance returned by our construction to exact distances obtained with BFS. We report the mean and the standard deviation of the relative error for each dataset. We used both oracles to compute the sketches. We present our results in Figure 3.8, which shows that our approximations are quite good. Indeed, our experiments show that our constructions could report *better* approximations than the underlying oracles. This is due to the fact that both oracles overestimate the distance so subtracting $\log |\mathbf{I}|$ can improve the approximation. For the *Gowalla* dataset, the mean of the relative error ranges from 0.36 to 0.13 when using the Das Sarma *et al.* oracle $\mathbf{DO}_1$. For *as-skitter*, it ranges from 0.45 to 0.22. The mean error and the variance decreases as we increase the size of each sketch. In addition, we note that $\mathbf{DO}_2$ performs better in all datasets. Also, half of the distances returned are exact and most of the distances returned are at most 2 away from the real distance. Figure 3.9 shows the histogram for the absolute error when using $\mathbf{DO}_2$ with $\rho = 3$. All the other datasets

are very similar to them, so we omit them due to space limitations.

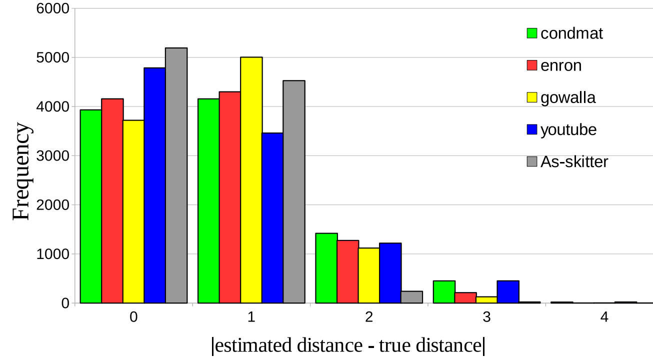


Figure 3.9: Absolute error histogram $\mathbf{DO}_2$ and $\rho = 3$

We note that a very small number of distances were negative and we removed them from the experiments. Negative distances result from the intersection size $|\mathbf{I}|$ being very large. Indeed, when the client decrypts the SWHE ciphertext returned by the server, it recovers $d \geq \mathsf{mindist} - \log |\mathbf{I}|$. If $|\mathbf{I}|$ is large and $\mathsf{mindist}$ is small (say, 1 or 2) then it is very likely that d is negative. However, in the experiments, the number of removed negative values were very small (i.e., 80 out of 10000 queries).

3.7 Application to Other Graph Queries

3.7.1 All-Distance Sketches

All-distances sketch (ADS) proposed by [Cohen et al., 2013, Cohen, 2014] can be used to estimate the shortest distance as well. As defined in [Cohen et al., 2013], for the weighted graph G, the *all-distance sketch* (ADS) of the node v, denote ADS(v), is the set of node ID and distance pairs. The included nodes are sample of the nodes reachable from v and with each included node $u \in \text{ADS}(v)$ the corresponding distance d_{uv} is also stored. Formally, followed by the notation from [Cohen, 2014, Cohen et al., 2013], let π_{vu} denote the *Dijkstra rank* of u with respect to v, defined as its position in the list of nodes ordered by increasing distance from v. For any two nodes u, v, let $\Phi_{<u}(v) = \{j | \pi_{vj} < \pi_{vu}\}$ for

the set of nodes that are closer to v than u is. For a numeric function $r : X \rightarrow [0, 1]$ over a set X, the function $k_r^{th}(X)$ returns the k-th smallest value in the range of r on X. If $|X| < k$ then we define $k_r^{th}(X) = 1$. Finally, *all-distance sketch*(ADS) labels are defined with respect to a random rank assignment to nodes such that for all v, $r(v) \sim U[0, 1]$, i.e, they are independently drawn from the uniform distribution on $[0, 1]$:

$$\text{ADS}(v) = \{(u, d_{vu}) | r(u) < k_r^{th}(\Phi_{<u}(v))\}$$

In other words, a node u belongs to $ADS(v)$ if u is among the k nodes with lowest rank r within the ball of radius d_{vu} around v. (For simplicity, we abuse notation and often interpret $ADS(v)$ as a set of nodes, even though it is actually a set of pairs , each consisting of a node and a distance.) Since the inclusion probability of a node is inversely proportional to its Dijkstra rank, the expected size of $ADS(v)$ is $\mathsf{E}|ADS(v)| \leq k \ln n$, where n is the number of nodes reachable from v.

It has been proved that one can use the ADS to estimate the distance between u and v. The estimated distance $\tilde{d}_{uv}$ have the approximation factor $(2\lceil \frac{\log n}{\log k} \rceil - 1)$ and the querying algorithm is very similar to the distance oracle (see [Cohen et al., 2013] for details).

3.7.2 Graph Similarity Queries using All-Distance Sketches

The closeness similarity measures the similarity of two nodes based on their views of the full graph. More precisely, we consider the distance from each of these two nodes to all other nodes in the graph and measure how much these two distance vectors differ. This is computationally expensive, but ADSs allow an efficient estimation of this measurement. It has also been shown that the ADS can be used to measure the closeness between two nodes. The following theorem show that it can be used to estimate the *Dijkstra Rank Closness* (denote by J^*). The Dijkstra rank of node v_j with respect to v_i is v_j's position in the nearest neighbors list of v_i when running the Dijkstra shortest path for the node v_i. Roughly speaking, closeness similarity is specified with respect to a distance function δ_{ij}

between two nodes, a *distance decay* function $\alpha'(d)$ (a monotone non-increasing function of distances), and *a weight function* $\beta(i)$ of node IDs. The basic expression for closeness similarity is

$$S_{\alpha',\beta}(u,v) = \sum_i \alpha'(\max\{\delta_{u,i}\delta_{v,i}\})\beta(i), \tag{3.2}$$

By setting k to be the parameter h in the decay function in the Dijkstra ranks, the following estimator can give a good estimation of J^*,

$$\hat{J}^*(u,v) = \frac{|\text{ADS}(u) \cap \text{ADS}(v)|}{|\text{ADS}(u) \cup \text{ADS}(v)|}$$

We refer the reader to [Cohen et al., 2013, Cohen, 2014] for details of how to set up the parameter of the ADS and its applications.

3.7.3 Graph Encryption based on ADS

We can easily apply the techniques of section 3.5.1 and 3.5.3 to the ADS data structure. For the approach in $\mathsf{GraphEnc}_1$, we generate the ADS for each node, the labels are computed by applying the PRF to each node, then apply similar approach as in $\mathsf{GraphEnc}_1$ using symmetric encryption. The Token and the Query algorithm are similarly defined. By retrieving the encryptions of the sketches, the client can use the ADS to measure the shortest distance and the estimation of the closeness. We will focus on the communication efficient construction based on $\mathsf{GraphEnc}_2$.

In order to to support the approximate shortest distance queries, we construct the graph encryption as $\mathsf{GraphEnc}_2$ except that during the $\mathsf{GraphEnc}_2.\mathsf{Setup}$ step (4) we generate the ADS for each node instead of using the distance oracle approach. The rest of the steps remains the same.

On the other hand, to measure $J^*(u,v)$ between u and v, we will construct a graph encryption scheme slightly different than the $\mathsf{GraphEnc}_2$ in Section 3.5.3. In Figure 5, we present the construction for closeness queries. During the Setup in $\mathsf{GraphEnc}_2$, instead of generating the sketches using the distance oracle, we generate the ADS for all the nodes,

then apply the same method using SWHE. In addition, as stated in [Cohen et al., 2013], ADS can be used to evaluate the similarity and closeness queries as well. Here we briefly mention the method of measuring the Dijkstra Rank Closeness based all-distance sketch. In the Setup, we first generate the ADS for each node. Next, for each $v \in V$, we compute the label $\ell_v := P_K(v)$ and initialize the array T_v of size t. For each $(w_i, \delta_i) \in ADS$, the only changes is that we now set $\mathsf{T}_v[h(w_i)] \leftarrow \Pi.\mathsf{Enc}_{\mathsf{pk}}(1)$ and fill the remaining the cells of the T_v with encryption of 0. Next, in $\mathsf{DistQuery}(u, v)$, the client generates the token for u and v by compute $\mathsf{tk}_1 := P_K(u)$ and $\mathsf{tk}_2 := P_K(v)$. When the server retrieves $\mathsf{T}_1 := \mathsf{DX}[\mathsf{tk}_1]$ and $\mathsf{T}_2 := \mathsf{DX}[\mathsf{tk}_2]$, we can homomorphically evaluate the $|\mathrm{ADS}(u)|$ by computing the following:

$$\psi_1 \leftarrow \Pi.\mathsf{Eval}(+, \mathsf{T}_1[1], \ldots, \mathsf{T}_1[t]) \tag{3.3}$$

Similarly, for $|\mathrm{ADS}(v)|$, we can compute $\psi_2 \leftarrow \Pi.\mathsf{Eval}(+, \mathsf{T}_2[1], \ldots, \mathsf{T}_2[t])$. We can alsohomomorphically evaluate the $|\mathrm{ADS}(u) \cap \mathrm{ADS}(v)|$: for all $i \in [t]$, the server computes $c_i \leftarrow \Pi.\mathsf{Eval}(\times, \mathsf{T}_1[i], \mathsf{T}_2[i])$, then the server computes $\psi_3 \leftarrow \Pi.\mathsf{Eval}(+, c_1, \ldots, c_t)$. Then by sending all of those to the client, the client decrypts ψ_1, ψ_2, and ψ_3, furthermore, the client can get the $|\mathrm{ADS}(u) \cap \mathrm{ADS}(v)|$ by computing $\beta := \psi_1 + \psi_2 - \psi_3$ due to the fact $|\mathrm{ADS}(u) \cup \mathrm{ADS}(v)| = |\mathrm{ADS}(u)| + |\mathrm{ADS}(u)| - |\mathrm{ADS}(u) \cap \mathrm{ADS}(v)|$. Finally, the Dijkstra Rank Closeness $J^*(u, v)$ can be estimated by $\frac{\psi_3}{\beta}$.

The correctness of above simply follows the fact that the bit we place into the encryption in Setup indicates the existence of whether the particular node hashes into the table T for each node. The equation 3.3 homomorphically computes the size of the sketch. The inner product between the two Ts gives the intersection size between $\mathrm{ADS}(u)$ and $\mathrm{ADS}(v)$.

The security proof of the scheme above is quite similar to the proof of $\mathsf{GraphEnc}_2$. The simulator just has to simulate the appropriate size of list of the encryptions. The query simulation is just like the simulation in $\mathsf{GraphEnc}_2$.

Our method 'hash-and-encrypt' techniques can be useful for many hop-based graph queries. We want to point out that the estimation for particular queries under different

Algorithm 5: Graph Encryption with Dijkstra Closeness Query with $O(1)$ communication complexity

1 Let $\mathsf{SWHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a homomorphic encryption scheme, $P : V \times \{0,1\}^k \rightarrow \{0,1\}^{\log|V|}$ be a pseudo-random permutation and H be a family of universal hash functions. Consider the graph encryption scheme $\mathsf{GraphEnc}_3 = (\mathsf{Setup}, \mathsf{DistQuery})$ that works as follows:

- $\mathsf{Setup}(1^k, G, \alpha, \varepsilon)$:
  1. sample $K \xleftarrow{\$} \{0,1\}^k$ and generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SWHE.Gen}(1^k)$;
  2. set $r = \widetilde{\Theta}\big(n^{2/(\alpha+1)}\big)$ and $t = 2 \cdot (r \log n)^2 \cdot \varepsilon^{-1}$;
  3. sample a hash function $h : V \rightarrow t$ from H;
  4. **Construct the ADS:**
     (a) compute ADS(v) for each $v \in V$.
  5. **Produce hash tables:**
     (a) For each node $v \in V$,
        i. compute $\ell_v := P_K(v)$;
        ii. initialize an array T_v of size t;
        iii. for each $(w_i, \delta_i) \in \mathrm{ADS}(v)$, set $\mathsf{T}_v[h(w_i)] \leftarrow \mathsf{SWHE.Enc}_{\mathsf{pk}}(1)$;
        iv. fill remaining cells of T_v with encryptions of 0;
        v. set $\mathsf{DX}[\ell_v] := \mathsf{T}_v$;
  6. output $\mathsf{EO} = \mathsf{DX}$.

- $\mathsf{DistQuery}\big((K, q), \mathsf{EO}\big)$:
  1. the client parses q as (u, v) and sends a token $\mathsf{tk} = (\mathsf{tk}_1, \mathsf{tk}_2) = (P_K(u), P_K(v))$ to the server;
  2. the server retrieves $\mathsf{T}_1 := \mathsf{DX}[\mathsf{tk}_1]$ and $\mathsf{T}_2 := \mathsf{DX}[\mathsf{tk}_2]$;
  3. $R_1 \leftarrow \mathsf{SWHE.Eval}(+, \mathsf{T}_1[1], ..., \mathsf{T}_1[N])$.
  4. $R_2 \leftarrow \mathsf{SWHE.Eval}(+, \mathsf{T}_2[1], ..., \mathsf{T}_2[N])$
  5. for all $i \in [t]$, the server computes $c_i \leftarrow \mathsf{SWHE.Eval}(\times, \mathsf{T}_1[i], \mathsf{T}_2[i])$;
  6. the server computes and sends to the client $R_3 \leftarrow \mathsf{SWHE.Eval}(+, c_1, \ldots, c_t)$;
  7. the client computes $m \leftarrow \mathsf{SWHE.Dec}_{\mathsf{sk}}(c)$ and computes $\hat{J} = \frac{R_3}{R_1 + R_2 - R_3}$. Finally, output $\hat{J}$.

graph sketches mainly depends on the underlying sketch structure. Our method provide a generic framework of encrypting the sketches. As shown in the previous section, our

constructions, which incorporate the hash and SWHE, do not affect the bound estimation much (in the case of measuring the closeness, the construction does not affect it at all).

Chapter 4

Top-k Query Processing on Encrypted Relational Databases

4.1 Introduction

Although top-k queries are important query types in many database applications [Ilyas et al., 2008], to the best of our knowledge, none of the existing works handle the top-k queries securely and efficiently. Vaiyda et. al. [Vaidya and Clifton, 2005] studied privacy-preserving top-k queries in which the data are vertically partitioned instead of encrypting the data. Wong et. al. [Wong et al., 2009] proposed an encryption scheme for knn queries and mentioned a method of transforming their scheme to solve top-k queries, however, as shown in [Yao et al., 2013], their encryption scheme is not secure and is vulnerable to chosen plaintext attacks. Vaiyda et. al. [Vaidya and Clifton, 2005] also studied privacy-preserving top-k queries in which the data are vertically partitioned instead of encrypting the data.

We assume that the data owner and the clients are trusted, but not the cloud server. Therefore, the data owner encrypts each database relation R using some probabilistic encryption scheme before outsourcing it to the cloud. An authorized user specifies a query q and generates a *token* to query the server. Our objective is to allow the cloud to securely compute the top-k results based on a user-defined ranking function over R, and, more importantly, the cloud should not learn anything about R or q. Consider a real world example for a health medical database below:

Example 4.1.1. *An authorized doctor, Alice, wants to get the top-k results based on some ranking criteria from the* encrypted *electronic health record database* **`patients`** *(see Table 4.1). The encrypted* **`patients`** *database may contain several attributes; here we only list a few in Table 4.1:* patient name, age, id number, trestbps [1], chol[2], thalach[3].

patient name	age	id	trestbps	chol	thalach
$E(Bob)$	$E(38)$	$E(121)$	$E(110)$	$E(196)$	$E(166)$
$E(Celvin)$	$E(43)$	$E(222)$	$E(120)$	$E(201)$	$E(160)$
$E(David)$	$E(60)$	$E(285)$	$E(100)$	$E(248)$	$E(142)$
$E(Emma)$	$E(36)$	$E(956)$	$E(120)$	$E(267)$	$E(112)$
$E(Flora)$	$E(43)$	$E(756)$	$E(100)$	$E(223)$	$E(127)$

Table 4.1: Encrypted `patients` Heart-Disease Data

One example of a top-k query (in the form of a SQL query) can be: ***`SELECT * FROM patients ORDERED BY chol+thalach STOP AFTER k`****. That is, the doctor wants to get the top-2 results based the score chol+thalach from all the patient records. However, since this table contains very sensitive information about the patients, the data owner first encrypts the table and then delegates it to the cloud. So, Alice requests a key from the data owner and generates a query* token *based on the query. Then the cloud searches and computes on the encrypted table to find out the top-k results. In this case, the top-2 results are the records of patients David and Emma.*

Our protocol extends the No-Random-Access (NRA) [Fagin et al., 2001] algorithm for computing top-k queries over a probabilistically encrypted relational database. Moreover, our query processing model assumes that two non-colluding semi-honest clouds, which is the model that has been showed working well (see [Elmehdwi et al., 2014, Bugiel et al., 2011, Liu et al., 2015a, Baldimtsi and Ohrimenko, 2014, Bost et al., 2015]). We encrypt the database in such a way that the server can obliviously execute NRA over the encrypted database without learning the underlying data. This is accomplished with the help of a secondary independent cloud server (or Crypto Cloud). However, the encrypted database

[1]trestbps: resting blood pressure (in mm Hg)
[2]chol: serum cholestoral in mg/dl
[3]maximum heart rate achieved

resides only in the primary cloud. We adopt two efficient state-of-art secure protocols, EncSort [Baldimtsi and Ohrimenko, 2014] and EncCompare [Bost et al., 2015], which are the two essential building block we need in our top-k secure construction. We choose these two building blocks mainly because of their efficiency.

During the query processing, we propose several novel sub-routines that can securely compute the best/worst score and de-duplicate replicated data items over the encrypted database. Notice that our proposed sub-protocols can also be used as stand-alone building blocks for other applications as well. We also would like to point out that during the querying phase the computation performed by the client is very small. The client only needs to compute a simple token for the server and all of the relatively heavier computations are performed by the cloud side. Moreover, we also explore the problem of top-k join queries over multiple encrypted relations.

We also design a *secure top-k join operator*, denote as $\bowtie_{\mathsf{sec}}$, to securely join the tables based on *equi-join* condition. The cloud homomorphically computes the top-k join on the top of joined results and reports the encrypted top-k results. Below we summarize our main contributions:

- We propose a new practical protocol designed to answer top-k queries over encrypted relational databases.
- We propose two encrypted data structures called EHL and EHL$^+$ which allow the servers to homomorphically evaluate the equality relations between two objects.
- We propose several independent sub-protocols such that the clouds can securely compute the best/worst scores and de-duplicate replicated encrypted objects with the use of another non-colluding server.
- We also extend our techniques to answer top-k join queries over multiple encrypted relations.
- The scheme is experimentally evaluated using real-world datasets and result shows that our scheme is efficient and practical.

4.2 Related Works and Background

The problem of processing queries over the outsourced encrypted databases is not new. The work [Hacigümüs et al., 2002] proposed executing SQL queries over encrypted data in the database-service-provider model using bucketization. Since then, a number of works have appeared on executing various queries over encrypted data. One of the relevant problem related to top-k queries is the kNN (k Nearest Neighbor) queries. Note that top-k queries should not be confused with similarity search, such as kNN queries. For the kNN queries, one is interested in retrieving the k most similar objects over the database to a query object, where the similarity between two objects is measured over some metric space, for example the L_2 metric. Many works have been proposed to specifically handle kNN queries on encrypted data, such as [Wong et al., 2009, Elmehdwi et al., 2014, Yao et al., 2013, Choi et al., 2014]. A significant amount of works have been done for privacy preserving keyword search queries or boolean queries, such as [Song et al., 2000, Curtmola et al., 2011, Cash et al., 2013b]. Recent work [Samanthula et al., 2014] proposed a general framework for boolean queries of disjunctive normal form queries on encrypted data. In addition, many works have been proposed for range queries [Shi et al., 2007, Hore et al., 2012, Li et al., 2014]. Other relevant works include privacy-preserving data mining [Lindell and Pinkas, 2000, Vaidya et al., 2008, Aggarwal and Yu, 2008, Jaideep Vaidya, 2008, Murat Kantarcioglu, 2004].

Recent works in the cryptography community have shown that it is possible to perform arbitrary computations over encrypted data, using fully homomorphic encryption (FHE) [Gentry, 2009a], or Oblivious RAM [Goldreich and Ostrovsky, 1996]. However, the performance overheads of such constructions are very high in practice, thus they're not suitable for practical database queries. Some recent advancements in ORAM schemes [Ren et al., 2015] show promise and can be potentially used in certain environments. As mentioned, [Vaidya and Clifton, 2005] is the only work that studied privacy preserving execution of top-k queries. However, their approach is mainly based on the

k-anonymity privacy policies, therefore, it cannot extended to encrypted databases. Recently, differential privacy [Dwork and Nissim, 2004] has emerged as a powerful model to protect against unknown adversaries with guaranteed probabilistic accuracy. However, here we consider *encrypted* data in the outsourced model; moreover, we do not want our query answer to be perturbed by noise, but we want our query result to be exact. Kuzu et. al. [Kuzu et al., 2014] proposed a scheme that leverages DP and leaks obfuscated access statistics to enable efficient searching. Another approach has been extensively studied is order-preserving encryption (OPE) [Agrawal et al., 2004, Popa et al., 2011, Boldyreva et al., 2011a, Aggarwal and Yu, 2008, Lindell and Pinkas, 2000], which preserves the order of the message. We note that, by definition, OPE directly reveals the order of the objects' ranks, thus does not satisfy our data privacy guarantee. Furthermore, [Hang et al., 2015] proposed a prototype for access control using deterministic proxy encryption, and other secure database systems have been proposed by using embedded secure hardware, such as TrustedDB [Bajaj and Sion, 2011] and Cipherbase [Arasu et al., 2015].

4.3 Preliminaries

4.3.1 Problem Definition

Consider a data owner that has a database relation R of n objects, denoted by $o_1, \dots, o_n$, and each object o_i has M attributes. For simplicity, we assume that all M attributes take numerical values. Thus, the relation R is an $n \times M$ matrix. The data owner would like to outsource R to a third-party cloud S_1 that is completely untrusted. Therefore, data owner encrypts R and sends the encrypted relation ER to the cloud. After that, any authorized client should be able to get the results of the top-k query over this encrypted relation directly from S_1, by specifying k and a score function over the M (encrypted) attributes. We consider the monotone scoring (ranking) functions that are weighted linear combinations over all attributes, that is $F_W(o) = \sum w_i \times x_i(o)$, where each $w_i \geq 0$ is a user-specified weight for the i-th attribute and $x_i(o)$ is the local score (value) of the i-th attribute

for object o. Note that we consider the monotone linear function mainly because it is the most important and widely used score function on top-k queries [Ilyas et al., 2008]. The results of a top-k query are the objects with the highest k scores of F_W values. For example, consider an authorized client, Alice, who wants to run a top-k query over the encrypted relation ER. Consider the following query: `q = SELECT * FROM ER ORDER BY` $F_W(\cdot)$ `STOP AFTER k;` That is, Alice wants to get the top-k results based on her scoring function F_W, for a specified set of weights. Alice first has to request the keys from the data owner, then generates a *query token* tk. Alice sends the tk to the cloud server. The cloud server storing the encrypted database ER processes the top-k query and sends the encrypted results back to Alice. In the real world scenarios, the authorized clients can locally store the keys for generating the token.

4.3.2 The Architecture

We consider the secure computation on the cloud under the *semi-honest* (or *honest-but-curious*) adversarial model. Furthermore, our model assumes the existence of two *different* non-colluding semi-honest cloud providers, S_1 and S_2, where S_1 stores the encrypted database ER and S_2 holds the secret keys and provides the crypto services. We refer to the server S_2 as the *Crypto Cloud* and assume S_2 resides in the cloud environment and is isolated from S_1. The two parties S_1 and S_2 do not trust each other, and therefore, they have to execute secure computations on encrypted data.

This model is not new and has already been widely used in related work, such as [Elmehdwi et al., 2014, Bugiel et al., 2011, Liu et al., 2015a, Baldimtsi and Ohrimenko, 2014, Bost et al., 2015]. As pointed out by these works, we emphasize that these cloud services are typically provided by some large companies, such as Amazon, Microsoft Azure, and Google, who have also commercial interests not to collude. The Crypto Cloud S_2 is equipped with a cryptographic processor, which stores the decryption key. The cryptographic processor has been built and used in real life (e.g.,

the IBM PCIe[4] or the Freescale C29x[5]). When the server S_1 receives the query token, S_1 initiates the secure computation protocol with the Crypto Cloud S_2. Figure 4.1 shows an overview of the architecture.

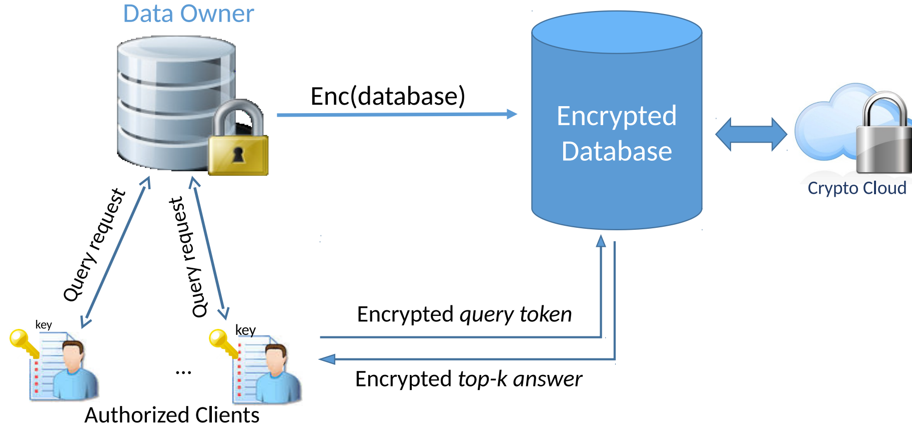


Figure 4.1: An overview of our model

4.3.3 Cryptographic Tools

In Table 4.2 we summarize the notation. In the following, we present the cryptographic primitives used in our construction.

Paillier Cryptosystem The Paillier cryptosystem [Paillier, 1999] is a semantically secure public key encryption scheme. We describe the algorithm in Chapter 2 The message space $\mathcal{M}$ for the encryption is $\mathbb{Z}_N$, where N is a product of two large prime numbers p and q. For a message $m \in \mathbb{Z}_N$, we denote $\mathsf{Enc}_{\mathsf{pk}}(m) \in \mathbb{Z}_{N^2}$ to be the encryption of m with the public key pk. When the key is clear in the text, we simply use $\mathsf{Enc}(m)$ to denote the encryption of m and $\mathsf{Dec}_{\mathsf{sk}}(c)$ to denote the decryption of a ciphertext c. The details of encryption and decryption algorithm can be found in [Paillier, 1999]. It has the following homomorphic properties:

- *Addition:* $\forall x, y \in \mathbb{Z}_N$, $\mathsf{Enc}(x) \cdot \mathsf{Enc}(y) = \mathsf{Enc}(x + y)$

[4] http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml
[5] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=C29x

- *Scalar Multiplication:* $\forall x, a \in \mathbb{Z}_N$, $\mathsf{Enc}(x)^a = \mathsf{Enc}(a \cdot x)$

Generalized Paillier Our construction also relies on Damgård-Jurik(DJ) cryptosystem introduced by Damgård and Jurik [Damgård and Jurik, 2001], which is a generalization of Paillier encryption. The message space $\mathcal{M}$ expands to $\mathbb{Z}_{N^s}$ for $s \geq 1$, and the ciphertext space is under the group $\mathbb{Z}_{N^{s+1}}$. As mentioned in [Adida and Wikström, 2007], this generalization allows one to doubly encrypt messages and use the additive homomorphism of the inner encryption layer under the same secret key. In particular, let $\mathsf{E}^2(x)$ denote an encryption of the DJ scheme for a message $x \in \mathbb{Z}_{N^2}$ (when $s = 2$) and $\mathsf{Enc}(x)$ be a normal Paillier encryption. This extension allows a ciphertext of the first layer to be treated as a plaintext in the second layer. Moreover, this nested encryption preserves the structure over inner ciphertexts and allows one to manipulate it as follows:

$$\mathsf{E}^2\big(\mathsf{Enc}(m_1)\big)^{\mathsf{Enc}(m_2)} = \mathsf{E}^2\big(\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2)\big) = \mathsf{E}^2\big(\mathsf{Enc}(m_1 + m_2)\big)$$

We note that this is the only homomorphic property that our construction relies on.

Throughout this paper, we use $\sim$ to denote that the underlying plaintext under encryption E are the same, i.e., $\mathsf{Enc}(x) \sim \mathsf{Enc}(y) \Rightarrow x = y$. We summarize the notation throughout this paper in Table 4.2. Note that in our application, we need one layered encryption; that is, given $\mathsf{E}^2\big(\mathsf{Enc}(x)\big)$, we want a normal Paillier encryption $\mathsf{Enc}(x)$. As introduced in [Baldimtsi and Ohrimenko, 2014], this could simply be done with the help of S_2. However, we need a protocol $\mathsf{RecoverEnc}$ to securely remove one layer of encryption.

4.3.4 No-Random-Access (NRA) Algorithm

The NRA algorithm [Fagin et al., 2001] finds the top-k answers by exploiting only sorted accesses to the relation R. The input to the NRA algorithm is a set of sorted lists S, each ranks the "same" set of objects based on different attributes. The output is a ranked list of these objects ordered on the aggregate input scores. We opted to use this algorithm

Notation	Definition
n	Size of the relation R, i.e. $\|R\| = n$
M	Total number of attributes in R
m	Total number of attributes for the query q
$\mathsf{Enc}(m)$	Paillier encryption of m
$\mathsf{Dec}(c)$	Paillier decryption of c
$\mathsf{E}^2(m)$	Damgård-Jurik (DJ) encryption of m
$\mathsf{Enc}(x) \sim \mathsf{Enc}(y)$	Denotes $x = y$, i.e. $\mathsf{Dec}(\mathsf{Enc}(x)) = \mathsf{Dec}(\mathsf{Enc}(y))$
$\mathsf{EHL}(o)$	Encrypted Hash List of the object o
$\mathsf{EHL}^+(o)$	Efficient Encrypted Hash List of the object o
$\ominus, \odot$	EHL and EHL^+ operations, see Section 4.5.
I_i^d	The data item in the ith sorted list L_i at depth d
$E(I_i^d)$	Encrypted data item I_i^d
$F_W(o)$	Cost function in the query token
$B^d(o)$	The best score (upper bound) of o at depth d
$W^d(o)$	The worst score (lower bound) of o at depth d

Table 4.2: Notation Summarization

Algorithm 6: NRA Algorithm [Fagin et al., 2001]

1 **def** *NRA* $(L_1, ..., L_M)$**:**
2   Do sorted access in parallel to each of the M sorted lists L_i. At each depth d:
  repeat
3   Maintain the bottom values $\underline{x}_1^d, \underline{x}_2^d, ..., \underline{x}_M^d$ encountered in the lists;
4   For every object o_i compute a lower bound $W^d(o_i)$ and upper bound $B^d(o_i)$;
5   Let T_k^d, the current top k list, contain the k objects with the largest $W^d(\cdot)$ values seen so far (and their grades), and let M_k^d be the kth largest lower bound value, $W^d(\cdot)$ in T_k^d;
6   Halt and return T_k^d when at least k distinct objects have been seen (so that in particular T_k^d contains k objects) and when $B^d(o_k) \leq M_k^d$ for all $o_k \notin T_k^d$, i.e the upper bound for every object who's not in T_k^d is no greater than M_k^d. Otherwise, go to next depth;
7   **until**;

because it provides a scheme that leaks minimal information to the cloud server (since during query processing there is no need to access intermediate objects). We assume that each column (attribute) is sorted independently to create a set of sorted lists S. The set of sorted lists is equivalent to the original relation, but the objects in each list L are sorted in ascending order according to their local score (attribute value). After sorting, R contains

M sorted lists, denoted as $S = \{L_1, L_2, \ldots, L_M\}$. Each sorted list consists of n data items, denoted as $L_i = \{I_i^1, I_i^2, \ldots, I_i^n\}$. Each data item is a object/value pair $I_i^d = (o_i^d, x_i^d)$, where o_i^d and x_i^d are the object id and local score at the depth d (when d objects have been accessed under sorted access in each list) in the ith sorted list respectively. Since it produces the top-k answers using bounds computed over their exact scores, NRA may not report the exact object scores. The score lower bound of some object o, $W(o)$, is obtained by applying the ranking function on o's known scores and the minimum possible values of o's unknown scores. The score upper bound of o, $B(o)$, is obtained by applying the ranking function on o's known scores and the maximum possible values of o's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. The algorithm reports a top-k object even if its score is not precisely known. Specifically, if the score lower bound of an object o is not below the score upper bounds of all other objects (including unseen objects), then o can be safely reported as the next top-k object. We give the details of the NRA in Algorithm 6.

4.4 Scheme Overview

In this section, we give an overview of our scheme. The two non-colluding semi-honest cloud servers are denoted by S_1 and S_2. Let $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$ be the secure top-k query scheme containing three algorithms Enc, Token and SecQuery. $\mathsf{Enc}(R)$ is the encryption algorithm that takes relation R as an input and outputs the encrypted relation ER. The idea of Enc is to encrypt and permute the set of sorted lists for R, so that the server can execute a variation of the NRA algorithm using only sequential accesses to the encrypted data. To do this encryption, we design a new encrypted data structure for the objects, called EHL. The Token algorithm takes a query q and produces a token for the query. The token serves as a trapdoor so that the cloud knows which list to access. Finally, SecQuery is the query processing algorithm that takes the token and securely computes top-k results based on the token. As mentioned earlier, our encryption scheme takes advantage

of the NRA top-k algorithm. In particular, S_1 scans the encrypted data depth by depth for each targeted list, maintaining a list of encrypted top-k object ids per depth until there are k encrypted object ids that satisfy the NRA halting condition. During this process, S_1 and S_2 learn nothing about the underlying scores and objects. At the end of the protocol, the object ids can be reported to the client. As we discuss next, there are two options after that. Either the encrypted records are retrieved and returned to the client, or the client retrieves the records using oblivious RAM [Goldreich and Ostrovsky, 1996] that does not even reveal the location of the actual encrypted records. In the first case, the server can get some additional information by observing the access patterns, i.e., the encrypted results of different queries. However, there are schemes that address this access leakage [Islam et al., 2012, Kuzu et al., 2014] and is beyond the scope of this paper. The second approach may be more expensive but is completely secure.

In the following sections, we first discuss the new encrypted data structures EHL and EHL$^+$. Then, we present the three algorithms Enc, Token and SecQuery in more details.

4.5 Encrypted Hash List (EHL)

In this paper, we propose a new data structure called encrypted hash list (EHL) to encrypt each object. The main purpose of this structure is to allow the cloud to homomorphically compute equality between the objects, whereas it is computationally hard for the server to figure out what the objects are. Intuitively, the idea is that given an object o we use s Pseudo-Random Function (PRF) to hash the object into a binary list of length H and then encrypt all the bits in the list to generate EHL. In partilar, we use the secure key-hash functions HMAC as the PRFs. Let $\mathsf{EHL}(o)$ be the encrypted list of an object o and let $\mathsf{EHL}(o)[i]$ denote the ith encryption in the list. In particular, we initialize an empty list EHL of length H and fill all the entries with 0. First, we generate s secure keys $\kappa_1, ..., \kappa_s$. The object o is hashed to a list as follows: 1) Set $\mathsf{EHL}[\mathtt{HMAC}(k_i, o) \mod H] = 1$ for $1 \leq i \leq s$. 2) Encrypt each bit using Paillier encryption: for $0 \leq j \leq H-1$, $\mathsf{Enc}(\mathsf{EHL}(o)[j])$. Fig. 4.2

shows how we obtain $\mathsf{EHL}(o)$ for the object o.

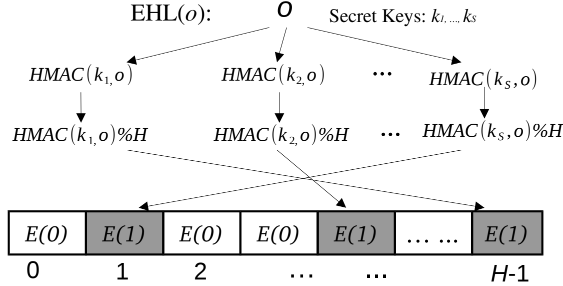


Figure 4.2: Encrypted Hash List for the object o.

Lemma 4.5.1. *Given two objects o_1 and o_2, their $\mathsf{EHL}(o_1)$ and $\mathsf{EHL}(o_2)$ are computationally indistinguishable.*

It is obvious to see that Lemma 4.5.1 holds since the bits in the EHL are encrypted by the semantically secure Paillier encryption scheme. Given $\mathsf{EHL}(x)$ and $\mathsf{EHL}(y)$, we define the *randomized operation* $\ominus$ between $\mathsf{EHL}(x)$ and $\mathsf{EHL}(y)$ as follows:

$$\mathsf{EHL}(x) \ominus \mathsf{EHL}(y) \stackrel{def}{=} \prod_{i=0}^{H-1} \big(\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1}\big)^{r_i} \tag{4.1}$$

where each r_i is some random value in $\mathbb{Z}_N$.

Lemma 4.5.2. *Let $\mathsf{Enc}(b) = \mathsf{EHL}(x) \ominus \mathsf{EHL}(y)$. Then the plaintext $b = 0$ if $x = y$ (two objects are the same), otherwise b is uniformly distributed in the group $\mathbb{Z}_N$ with high probability.*

Proof Sketch: Let $\mathsf{Enc}(x_i) = \mathsf{EHL}(x)[i]$ and $\mathsf{Enc}(y_i) = \mathsf{EHL}(y)[i]$. If $x = y$, i.e. they are the same objects, then for all $i \in [0, H-1]$, $x_i = y_i$. Therefore,

$$\prod_{i=0}^{H-1} (\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1})^{r_i} = \mathsf{E}\big(\sum_{i=0}^{H-1} (r_i(x_i - y_i))\big) = \mathsf{Enc}(0)$$

In the case of $x \neq y$, it must be true, with high probability, that there exists some $i \in [0, H-1]$ such that $\mathsf{Enc}(x_i) \nsim \mathsf{Enc}(y_i)$, i.e. the underlying bit at location i in $\mathsf{EHL}(x)$ is

different from the bit in $\mathsf{EHL}(y)$. Suppose $\mathsf{EHL}(x)[i] = \mathsf{Enc}(1)$ and $\mathsf{EHL}(y)[i] = \mathsf{Enc}(0)$. Therefore, the following holds:

$$\left(\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1}\right)^{r_i} = \mathsf{Enc}(r_i(1-0)) = \mathsf{Enc}(r_i)$$

Hence, based on the definition $\ominus$, it follows that b becomes random value uniformly distributed in the group $\mathbb{Z}_N$. ■

It is worth noting that one can also use BGN cryptosystem for the similar operations above, as the BGN scheme can homomorphically evaluate quadratic functions (see Chapter 2).

False Positive Rate. Note that the construction is indeed a probabilistically encrypted Bloom Filter except that we use one list for each object and encrypt each bit in the list. The construction of EHL may report some false positive results for its $\ominus$ operation, i.e. $\mathsf{Enc}(0) \leftarrow \mathsf{EHL}(x) \ominus \mathsf{EHL}(y)$ when $x \neq y$. This is due to the fact that x and y may be hashed to exactly the same locations using s many HMACs. Therefore, it is easy to see that the false positive rate (FPR) is the same as the FPR of the Bloom Filter, where we can choose the number of hash functions HMAC s to be $\frac{H}{n}\ln 2$ to minimize the false positive rate to be $(1-(1-\frac{1}{H}^{sn}))^s \approx (1-e^{-sn/H})^s \approx 0.62^{H/n}$. To reduce the false positive rate, we can increase the length of the list H. However, this will increase the cost of the structure both in terms of space overhead and number of operations for the randomization operation which is $O(H)$. In the next subsection, we introduce a more compact and space-efficient encrypted data structure EHL^+.

$\mathbf{EHL}^+$. We now present a computation- and space-efficient encrypted hash list EHL^+. The idea of the efficient EHL^+ is to first 'securely hash' the object o to a larger space s times and only encrypt those hash values. Therefore, for the operation $\ominus$, we only homomorphically subtract those hashed values. The complexity now reduces to $O(s)$ as opposed to $O(H)$, where s is the number of the secure hash functions used. We show that

one can get negligible false positive rate even using a very small s. To create an $\mathsf{EHL}^{+}(o)$ for an object o, we first generate s secure keys $k_1, ..., k_s$, then initialize a list EHL^{+} of size s. We first compute $o_i \leftarrow \mathtt{HMAC}(k_i, o) \mod N$ for $1 \leq i \leq s$. This step maps o to an element in the group $\mathbb{Z}_N$, i.e. the message space for Paillier encryption. Then set $\mathsf{EHL}^{+}[i] \leftarrow \mathsf{Enc}(o_i)$ for $1 \leq i \leq s$. The operation $\ominus$ between $\mathsf{EHL}^{+}(x)$ and $\mathsf{EHL}^{+}(y)$ are similar defined as in Equation(4.1), i.e. $\mathsf{EHL}^{+}(x) \ominus \mathsf{EHL}^{+}(y) \stackrel{def}{=} \prod_{i=0}^{s-1} \left(\mathsf{EHL}(x)[i] \cdot \mathsf{EHL}(y)[i]^{-1}\right)^{r_i}$, where each r_i is some randomly generated value in $\mathbb{Z}_N$. Similarly, EHL^{+} has the same properties as EHL. Let $\mathsf{Enc}(b) \leftarrow \mathsf{EHL}^{+}(x) \ominus \mathsf{EHL}^{+}(y)$, $b = 0$ if $x = y$ and otherwise b is random in $\mathbb{Z}_N$ with high probability.

We now analyze the false positive rate (FPR) for EHL^{+}. The false positive answer occurs when $x \neq y$ and $\mathsf{Enc}(0) \leftarrow \mathsf{EHL}^{+}(x) \ominus \mathsf{EHL}^{+}(y)$. That is $\mathtt{HMAC}(k_i, x)\%N = \mathtt{HMAC}(k_i, y)\%N$ for each $i \in [1, s]$. Assuming $\mathtt{HMAC}$ is a Pseudo-Random Function, the probability of this happens is at most $\frac{1}{N^s}$. Taking the union bound gives that the FPR is at most $\binom{n}{2}\frac{1}{N^s} \leq \frac{n^2}{N^s}$. Notice that $N \approx 2^{\lambda}$ is large number as N is the product of two large primes p and q in the Paillier encryption and λ is the security parameter. For instance, if we set N to be a 256 bit number (128-bit primes in Paillier) and set $s = 4$ or 5, then the FPR is negligible even for millions of records. In addition, the size of the EHL^{+} is much smaller than EHL as it stores only s encryptions. In the following section, we simply say EHL to denote the encrypted hash list using the EHL^{+} structure.

Notation. We introduce some notation that we use in our construction. Let $\mathbf{x} = (x_1, \ldots, x_s) \in \mathbb{Z}_N^s$ and let the encryption $\mathsf{Enc}(\mathbf{x})$ denotes the concatenation of the encryptions $\mathsf{Enc}(x_1)...\mathsf{Enc}(x_s)$. Also, we denote by $\odot$ the block-wise multiplication between $\mathsf{Enc}(\mathbf{x})$ and $\mathsf{EHL}(y)$; that is, $\mathbf{c} \leftarrow \mathsf{Enc}(\mathbf{x}) \odot \mathsf{EHL}(y)$, where $\mathbf{c}_i \leftarrow \mathsf{Enc}(x_i) \cdot \mathsf{EHL}(y)[i]$ for $i \in [1, s]$.

4.6 Database Encryption

We describe the database encryption procedure Enc in this section. Given a relation R with M attributes, the data owner first encrypts the relation using Algorithm 7.

Algorithm 7: $\mathsf{Enc}(R)$: Relation encryption

1. Given the relation R, sort each L_i based on the attribute's value for $1 \leq i \leq M$;
2. Generate a public/secret key $\mathsf{pk_p}, \mathsf{sk_p}$ for the Paillier encryption scheme and random secret keys $\kappa_1, \ldots, \kappa_s$ for EHL;
3. Do sorted access in parallel to each of the M sorted lists L_i;
4. **foreach** *data item* $I_i = \langle o_i^d, x_i^d \rangle \in L_i$ **do**
5.   **foreach** *depth* d **do**
6.   Compute $\mathsf{EHL}(o_i^d)$ using the keys $\kappa_1, \ldots, \kappa_s$;
7.   Compute $\mathsf{Enc}_{\mathsf{pk_p}}(x_i^d)$ using $\mathsf{pk_p}$;
8.   Store the item $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}_{\mathsf{pk_p}}(x_i^d \rangle)$ at depth d;
9. Generate a secret key K for a pseudorandom permutation P and permute all the list based on g. For $1 \leq i \leq M$, permute L_i as $L_{P_K(i)}$;
10. The data owner securely uploads the keys $\mathsf{pk_p}, \mathsf{sk_p}$ to the S_2, and only $\mathsf{pk_p}$ to S_1;
11. Finally, each permuted list contains a list of encrypted item of the form $E(I^d) = \langle \mathsf{EHL}(o^d), \mathsf{Enc}_{\mathsf{pk_p}}(x^d) \rangle$. Output all lists of encrypted items as the encrypted relation as ER;

In ER each data item $I_i^d = (o_i^d, x_i^d)$ at depth d in the sorted list L_i is encrypted as $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}_{\mathsf{pk_p}}(x_i^d) \rangle$. As all the score has been encrypted under the public key $\mathsf{pk_p}$, for the rest of the paper, we simply use $\mathsf{Enc}(x)$ to denote the encryption $\mathsf{Enc}_{\mathsf{pk_p}}(x)$ under the public key $\mathsf{pk_p}$. Besides the size of the database and M, the encrypted ER doesn't reveal anything. In Theorem 4.6.1, we demonstrate this by showing that two encrypted databases are indistinguishable if they have the same size and number of attributes. We denote $|R|$ by the size of a relation R.

Theorem 4.6.1. *Given two relations R_1 and R_2 with $|R_1| = |R_2|$ and same number of attributes. The encrypted ER_1 and ER_2 output by the algorithm Enc are indistinguishable.*

The proof is straight forward as it's easy to see that the theorem holds based on Lemma 4.5.1 and Paillier encryption scheme.

4.7 Query Token

Consider the SQL-like query `q = SELECT * FROM ER ORDERED BY` $F_W(\cdot)$ `STOP BY k`, where $F_W(\cdot)$ is a weighted linear combination of all attributes. In this paper, to simplify our presentation of the protocol, we consider binary weights and therefore the scoring function is just a sum of the values of a subset of attributes. However, notice that for non $\{0, 1\}$ weights the client should provide these weights to the server and the server can simply adapt the same techniques by using the scalar multiplication property of the Paillier encryption before it performs the rest of the protocol which we discuss next. The Token algorithm is quite simple and works as follows: the client specifies the scoring attribute set M of size m, i.e. $|\mathsf{M}| = m \leq M$, then requests the key K from the data owner, where K is the key corresponds the Pseudo Random Permutation P. Then the client computes the $P_{\mathsf{K}}(i)$ for each $i \in \mathsf{M}$ and sends the following query token to the cloud server S_1: $\mathsf{tk} =$ `SELECT * FROM` ER `ORDERED BY` $\{P_{\mathsf{K}}(i)\}_{i \in \mathsf{M}}$ `STOP BY k`.

4.8 Top-k Query Processing

As mentioned, our query processing protocol is based on the NRA algorithm. However, the technical difficulty is to execute the algorithm on the encrypted data while S_1 does not learn any object id or any score and attribute value of the data. We incorporate several cryptographic protocols to achieve this. Our query processing uses two state-of-the-art efficient and secure protocols: $\mathsf{EncSort}$ introduced by [Baldimtsi and Ohrimenko, 2014] and $\mathsf{EncCompare}$ introduced by [Bost et al., 2015] as building blocks. We skip the detailed description of these two protocols since they are not the focus of this paper. Here we only describe their functionalities: 1). $\mathsf{EncSort}$: S_1 has a list of encrypted keyed-value pairs $(\mathsf{Enc}(key_1), \mathsf{Enc}(a_1))...(\mathsf{Enc}(key_m), \mathsf{Enc}(a_m))$ and a public key pk, and S_2 has the secret key sk. At the end of the protocol, S_1 obtains a list *new* encryptions $(\mathsf{Enc}(key'_1), \mathsf{Enc}(a'_1))...(\mathsf{Enc}(key'_m), \mathsf{Enc}(a'_m))$, where the key/value list is sorted based on the order $a'_1 \leq a'_2 ... \leq a'_m$ and the set $\{(key_1, a_1), ..., (key_m, a_m)\}$ is the same as

$\{(key'_1, a'_1), ..., (key'_m, a'_m)\}$. 2). $\mathsf{EncCompare}(\mathsf{Enc}(a), \mathsf{Enc}(b))$: S_1 has a public key pk and two encrypted values $\mathsf{Enc}(a), \mathsf{Enc}(b)$, while S_2 has the secret key sk. At the end of the protocol, S_1 obtains the bit f such that $f := (a \leq b)$. Several protocols have been proposed for the functionality above. We choose the one from [Bost et al., 2015] mainly because it is efficient and perfectly suits our requirements.

4.8.1 Query Processing: SecQuery

We first give the overall description of the top-k query processing SecQuery at a high level. Then in Chapter 4.8.2, we describe in details the secure sub-routines that we use in the query processing: SecWorst, SecBest, SecDedup, and SecUpdate.

As mentioned, SecQuery makes use of the NRA algorithm but is different from the original NRA, because SecQuery cannot maintain the global worst/best scores in plaintext. Instead, SecQuery has to run secure protocols depth by depth and homomorphically compute the worst/best scores based on the items at each depth. It then has to update the complete list of encrypted items seen so far with their global worst/best scores. At the end, server S_1 reports k encrypted objects (or object ids) without learning any object or its scores.

Notations. In the encrypted database, we denote each *encrypted item* by $E(I) = \langle \mathsf{EHL}(o), \mathsf{Enc}(x) \rangle$, where I is the item with object id o and score x. During the query processing, the server S_1 needs to maintain the encrypted item with its current best/worst scores, and we denote by $\mathbf{E}(I) = (\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$ the *encrypted score item* I with object id o with best score B and worst score W.

In particular, upon receiving the query token tk = `SELECT * FROM ER ORDERED BY` $\{P_K(i)\}_{i \in \mathsf{M}}$ `STOP BY k`, the cloud server S_1 begins to process the query. The token tk contains $\{P_K(i)\}_{i \in \mathsf{M}}$ which informs S_1 to perform the sequential access to the lists $\{L_{P_K(i)}\}_{i \in \mathsf{M}}$. By maintaining an encrypted list T, which includes items with their encrypted global best and worst scores, S_1 updates the list T depth by depth. Let T^d be the state of the en-

Algorithm 8: Top-k Query Processing: $\mathsf{SecQuery}$

```
1  S_1 receives Token from the client;
2  Parses the Token and let L_i = L_{P_K(j)} for j ∈ M;
3  foreach depth d at each list do
4      foreach E(I_i^d) = ⟨EHL(o_i^d), Enc(x_i^d)⟩ ∈ L_i do
           /* Compute the worst score for object o_i^d at current depth d */
5          Compute Enc(W_i^d)) ← SecWorst(E(I_i^d), H, pk_p, sk_p), where H =
           {E(I_j^d)}_{j∈m,i≠j};
           /* Compute the best score for object o_i^d at current depth d */
6          Compute Enc(B_i^d) ← SecBest(E(I_i^d), {j}_{j≠i}, pk_p, sk_p);
       /* gets encrypted list Γ_d without duplicated objects */
7      Run Γ_d ← SecDedup({E(I_i^d)}, pk_p, sk_p) with S_2 and get the local encrypted list
       Γ_d;
8      Run T^d ← SecUpdate(T^{d-1}, Γ_d, pk_p, sk_p) with S_2 and get T^d;
9      If |T^d| < k elements, go to the next depth. Otherwise, run EncSort(T^d) by
       sorting on Enc(W_i), get first k items as T_k^d;
10     Let the kth and the (k+1)th item be E(I'_k) and E(I'_{k+1}), S_1 then runs
       f ← EncCompare(E(W'_k), E(B'_{k+1})) with S_2, where E(W'_k) is the worst score for
       E(I'_k), and E(B'_{k+1}) is the best score for E(I'_{k+1}) in T^d;
11     if f = 0 then
12         Halt and return the encrypted first k item in T_k^d
```

crypted list T after depth d. At depth d, S_1 first homomorphically computes the local encrypted worst/best scores for each item appearing at this depth by running $\mathsf{SecWorst}$ and $\mathsf{SecBest}$.

In $\mathsf{SecWorst}$, S_1 takes the input of the current encrypted item $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}(x_i^d) \rangle$ and all of the encrypted items in other lists H at current depth, i.e., $H = \{E(I_j^d)\}_{j \neq i, j \in \mathsf{M}}$. S_1 runs the protocol $\mathsf{SecWorst}$ with S_2, and obtains the encrypted worst score for the object o_i^d. Similarly, in the protocol $\mathsf{SecBest}$, S_1 takes the input of the current encrypted item $E(I_i^d) = \langle \mathsf{EHL}(o_i^d), \mathsf{Enc}(x_i^d) \rangle$ and the list pointers $\{j\}_{j \neq i}$ that indicates all of the encrypted item seen so far. S_1 runs the protocol $\mathsf{SecBest}$ with S_2, and obtains the encrypted worst score for the object o_i^d. Then S_1 securely replaces the duplicated encrypted objects with large encrypted worst scores Z by running $\mathsf{SecDedup}$ with S_2. In the $\mathsf{SecDedup}$ protocol, S_1 inputs the current encrypted items, $\{E(I_i^d)\}$, seen

so far. After the execution of the protocol, S_1 gets list of encrypted items Γ_d such that there are no duplicated objects. Next, S_1 updates the encrypted global list from state T^{d-1} to state T^d by applying $\mathsf{SecUpdate}$. After that, S_1 utilizes $\mathsf{EncSort}$ to sort the distinct encrypted objects with their scores in T^d to obtain the first k encrypted objects which are essentially the top-k objects based on their worst scores so far. The protocol halts if at some depth, the encrypted best score of the (k+1)-th object, $\mathsf{Enc}(B_{k+1})$, is less than the k-th object's encrypted worst score $\mathsf{Enc}(W_k)$. This can be checked by calling the protocol $\mathsf{EncCompare}(\mathsf{Enc}(W_k), \mathsf{Enc}(B_{k+1}))$. Followed by underlying NRA algorithm, it is easy to see that S_1 can correctly reports the encrypted top-k objects. We describe the detailed query processing in Algorithm 8.

4.8.2 Building Blocks

In this section, we present the detailed description of the protocols SecWorst, SecBest, SecDedup, and SecUpdate.

4.8.2.1 Secure Worst Score

At each depth, for each encrypted data item, server S_1 should obtain the encryption $\mathsf{Enc}(W)$, which is the worst score based on the items at the current depth *only*. Note that this is different than the normal NRA algorithm as it computes the global worst possible score for each encountered objects until the current depth. We formally describe the protocol setup below:

Protocol 4.8.1. *Server S_1 has the input $E(I) = \langle \mathsf{EHL}(o), \mathsf{Enc}(x) \rangle$, a set of encrypted items H, i.e. $H = \{E(I_i)\}_{i=[|H|]}$, where $E(I_i) = \langle \mathsf{EHL}(o_i), \mathsf{Enc}(x_i) \rangle$, and the public key $\mathsf{pk_p}$. Server S_2's inputs are $\mathsf{pk_p}$ and $\mathsf{sk_p}$.* SecWorst *securely computes the encrypted worst ranking score based on L, i.e., S_1 outputs $\mathsf{Enc}(W(o))$, where $W(o)$ is the worst score based on the list H.*

The technical challenge here is to homomorphically evaluate the encrypted score only

based on the objects' equality relation. That is, if the object is the same as another o from L, then we add the score to $\mathsf{Enc}(W(o))$, otherwise, we don't. However, we want to prevent the servers from knowing the relations between the objects at any depth. We overcome this problem using the protocol $\mathsf{SecWorst}(E(I), L)$ between the two servers S_1 and S_2. We present the detailed protocol description of $\mathsf{SecWorst}$ in Algorithm 9.

Algorithm 9: $\mathsf{SecWorst}\big(E(I), H = \{E(I_i)\}_{i\in[|H|]}, \mathsf{pk_p}, \mathsf{sk_p}\big)$: Worst Score Protocol

S_1**'s input**: $E(I)$, $H = \{E(I_j)\}$, $\mathsf{pk_p}$
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1. **Server** S_1**:**
2. Let $|H| = m$. Generate a random permutation $\pi : [m] \rightarrow [m]$;
3. For the set of encrypted items $H = \{E(I_j)\}$, permute each $E(I_j)$ in H as $E(I_{\pi(j)}) = \mathsf{EHL}(o_{\pi(j)}), \mathsf{Enc}(x_{\pi(j)})$.;
4. **for** *each permuted item in* $E(I_{\pi(j)})$ **do**
5. compute $\mathsf{Enc}(b_j) \leftarrow \mathsf{EHL}(o) \ominus \mathsf{EHL}(o_{\pi(j)})$, send $\mathsf{Enc}(b_j)$ to S_2
6. Receive $\mathsf{E}^2\big(t_i\big)$ from S_2 and evaluate: $\mathsf{E}^2\big(\mathsf{Enc}(x'_i)\big) := \mathsf{E}^2\big(t_i\big)^{\mathsf{Enc}(x_i)} \cdot \Big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_i\big)^{-1}\Big)^{\mathsf{Enc}(0)}$;
7. Run $\mathsf{Enc}(x'_i) \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(x'_i)\big), \mathsf{pk_p}, \mathsf{sk_p})$ with S_2;
8. Set the worst score $\mathsf{Enc}(W) \leftarrow (\prod_{i=1}^{m} \mathsf{Enc}(x'_i))$;
9. Output $\mathsf{Enc}(W)$.
10. **Server** S_2**:**
11. **for** *each* $\mathsf{Enc}(b_i)$ *received from* S_1 **do**
12. Decrypt to get b_i, set $t_i \leftarrow (b_i = 0\ ?\ 1 : 0)$;
13. Send $\mathsf{E}^2\big(t_i\big)$ to S_1.

Intuitively, the idea of $\mathsf{SecWorst}$ is that S_1 first generates a random permutation π and permutes the list of items in L. Then, it computes the $\mathsf{Enc}(b_i)$ between $E(I)$ and each permuted $E(I_{\pi(i)})$, and sends $\mathsf{Enc}(b_i)$ to S_2. The random permutation prevents S_2 from knowing the pair-wise relations between o and the rest of the objects o_i's. Then S_2 sends $\mathsf{E}^2(t_i)$ to S_1 (line 13). Based on Lemma 4.5.2, $t_i = 1$ if two objects are the same, otherwise $t_i = 0$. S_1 then computes $\mathsf{E}^2\big(\mathsf{Enc}(x'_i)\big) \leftarrow \mathsf{E}^2\big(t_i\big)^{\mathsf{Enc}(x_i)} \cdot \Big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_i\big)^{-1}\Big)^{\mathsf{Enc}(0)}$. Based on the properties of DJ Encryption,

$$\mathsf{E}^2\big(t_i\big)^{\mathsf{Enc}(x_i)} \cdot \Big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_i\big)^{-1}\Big)^{\mathsf{Enc}(0)} = \mathsf{E}^2\big(t_i \cdot \mathsf{Enc}(x_i) + (1-t_i)\cdot \mathsf{Enc}(0)\big) = \mathsf{E}^2\big(\mathsf{Enc}(x'_i)\big)$$

Therefore, it follows that $x'_i = 0$ if $t_i = 0$, otherwise $x'_i = x_i$. S_1 then runs $\mathsf{RecoverEnc}(\mathsf{E}^2(\mathsf{Enc}(x'_i)), \mathsf{pk_p}, \mathsf{sk_p})$ (describe in Algorithm 10) to get $\mathsf{Enc}(x'_i)$. Note that the protocol RecoverEnc is also used in other protocols. Finally, S_1 evaluates the following equation: $\mathsf{Enc}(W(o)) \leftarrow \prod_{i=1}^{m} \mathsf{Enc}(x'_i)$. S_1 can correctly evaluate the worst score, because that, when $t_i = 0$, the object o_i is not the same as o, otherwise, $t_i = 1$. The following formula gives the correct computation of the worst score:

$$\prod_{i=1}^{m} \mathsf{Enc}(x'_i) = \mathsf{Enc}(\sum_{i=1}^{m} x'_i), \text{ where } x'_i = \begin{cases} x_i & \text{if } o_i = o \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 10: $\mathsf{RecoverEnc}(\mathsf{E}^2(\mathsf{Enc}(c)), \mathsf{pk_p}, \mathsf{sk_p})$ Recover Encryption

S_1**'s input**: $\mathsf{E}^2(\mathsf{Enc}(c))$, $\mathsf{pk_p}$
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1 **Server** S_1**:**
2 Generate $r \xleftarrow{\$} \mathbb{Z}_N$, compute and send $\mathsf{E}^2(\mathsf{Enc}(c+r)) \leftarrow \mathsf{E}^2(\mathsf{Enc}(c))^{\mathsf{Enc}(r)}$ to S_2.
3 **Server** S_2**:**
4 Decrypt as $\mathsf{Enc}(c+r)$ and send back to S_1
5 **Server** S_1**:**
6 Receive $\mathsf{Enc}(c+r)$ and compute: $\mathsf{Enc}(c) = \mathsf{Enc}(c+r) \cdot \mathsf{Enc}(r)^{-1}$;
7 Output $\mathsf{Enc}(c)$.

Note that nothing has been leaked to S_1 at the end of the protocol. However, there is some leakage function revealed to S_2 at current depth, which we will describe it in detail in later section. However, even by learning this pattern, S_2 has still no idea on which particular item is the same as the other at this depth since S_1 randomly permutes the item before sending to S_2 and everything has been encrypted. Moreover, no information has been leaked on the objects' scores.

4.8.2.2 Secure Best Score

The secure computation for the best score is different from computing the worst score. Below we describe the protocol SecBest between S_1 and S_2:

Protocol 4.8.2. *Server S_1 takes the inputs of the public key $\mathsf{pk_p}$, $E(I) = \langle \mathsf{EHL}(o), \mathsf{Enc}(x) \rangle$ for the object o in list L_i, and a set of pointers $\mathcal{P} = \{j\}_{i \neq j, j \in \mathsf{M}}$ to the list in* ER*. Server S_2's inputs are $\mathsf{pk_p}$, $\mathsf{sk_p}$. The protocol* SecBest *securely computes the encrypted best score at the current depth d, i.e., S_1 finally outputs $\mathsf{Enc}(B(o))$, where $B(o)$ is the best score for the o at current depth.*

Algorithm 11: $\mathsf{SecBest}\big(E(I_i), \mathcal{P}, \mathsf{pk_p}, \mathsf{sk_p}\big)$ Secure Best Score.

S_1**'s input**: $E(I_i)$ in list L_i, $\mathcal{P} = \{j\}_{i \neq j}$, $\mathsf{pk_p}$
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1. **Server** S_1**:**
2. **foreach** *list L_i* **do**
3. maintain $\mathsf{Enc}(\underline{x}_i^d)$ for L_i, where $\mathsf{Enc}(\underline{x}_i^d)$ is the encrypted score at depth d.
4. Generate a random permutation $\pi : [l] \rightarrow [l]$;
5. Permute each L_i as $L_{\pi(i)} = \mathsf{EHL}(o_{\pi(i)}), \mathsf{Enc}(x_{\pi(i)})$;
6. **foreach** *permuted $E(I_{\pi(i)})$* **do**
7. compute $\mathsf{Enc}(b_i) \leftarrow \mathsf{EHL}(o) \ominus \mathsf{EHL}(o_i)$
8. send $\mathsf{Enc}(b_i)$ to S_2 receive $\mathsf{E}^2\big(t_i\big)$ and compute:
   $\mathsf{E}^2\big(\mathsf{Enc}(x_i')\big) := \mathsf{E}^2\big(t_i\big)^{\mathsf{Enc}(x_i)} \cdot \Big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_i\big)^{-1}\Big)^{\mathsf{Enc}(0)}$;
9. run $\mathsf{Enc}(x_i') \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(x_i')\big), \mathsf{pk_p}, \mathsf{sk_p})$ with S_2;
10. compute $\mathsf{E}^2\big(\mathsf{Enc}(\underline{x}_i'^d)\big) \leftarrow \big(1 - \prod_{i=1}^{d} \mathsf{E}^2\big(t_i\big)\big)^{\mathsf{Enc}(\underline{x}_i^d)}$;
11. run $\mathsf{Enc}(\underline{x}_i'^d) \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(\underline{x}_i'^d)\big), \mathsf{pk_p}, \mathsf{sk_p})$ with S_2;
12. set $\mathsf{Enc}(B_i) \leftarrow \mathsf{Enc}(\underline{x}_i'^d) \cdot (\prod_{i=1}^{l} \mathsf{Enc}(x_i'))$;
13. compute $\mathsf{Enc}(B) \leftarrow \prod_{i=1}^{m} \mathsf{Enc}(B_i)$ and output $\mathsf{Enc}(B)$;
14. **Server** S_2**:**
15. **for** $\mathsf{Enc}(b_i)$ *received from S_1* **do**
16. Decrypt to get b_i. If $b_i = 0$, set $t_i = 1$, otherwise, set $t_i = 0$;
17. Send $\mathsf{E}^2\big(t_i\big)$ to S_1.

At depth d, let $E(I)$ be the encrypted item in the list L_i, then its best score up to this depth is based on the whether this item has appeared in other lists $\{L_j\}_{j \neq i, j \in \mathsf{M}}$. The

detailed description for SecBest is described in Algorithm 11.

In SecBest, S_1 has to scan the encrypted items in the other lists to securely evaluate the current best score for the encrypted $E(I)$. The last seen encrypted item in each sorted list contains the encryption of the *best possible values* (or *bottom scores*). If the same object o appears in the previous depth then homomorphically adds the object's score to the encrypted best score $\mathsf{Enc}(B)$, otherwise adds the bottom scores seen so far to $\mathsf{Enc}(B)$. In particular, S_1 can homomorphically evaluate (at line 9): $\mathsf{E}^2\big(x'_i\big) = \mathsf{E}^2\big(t_i \cdot \mathsf{Enc}(x_i) + (1 - t_i) \cdot \mathsf{Enc}(0)\big)$. That is, if $t_i = 0$ which means item I appeared in the previous depth, x'_i will be assigned the corresponding score x_i, otherwise, $x'_i = 0$. Similarly, S_1 homomorphically evaluates the following: $\mathsf{Enc}(\underline{x}'^d_i)) = \mathsf{Enc}((1 - \sum_i^d t_i) \cdot \underline{x}^d_i)$. If the item I does not appear in the previous depth, then $(1 - \sum_i^d t_i) = 1$ since each $t_i = 0$, therefore, $\underline{x}'^d_i$ will be assigned to the bottom value $\underline{x}^d_i$. Finally, S_1 homomorphically add up all the encrypted scores and get the encrypted best scores (line 12).

4.8.2.3 Secure Deduplication

At each depth, some of the objects might be repeatedly computed since the same objects may appear in different sorted list at the same depth. S_1 cannot identify duplicates since the items and their scores are probabilistically encrypted. We now present a protocol that deduplicates the encrypted objects in the following.

Protocol 4.8.3. *Let the* $\boldsymbol{E}(I)$ *be an encrypted scored item such that* $\boldsymbol{E}(I) = (\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$*, i.e. the* $\boldsymbol{E}(I)$ *is associated with* $\mathsf{EHL}(o_i)$*, its encrypted worst and best score* $\mathsf{Enc}(W_i)$*,* $\mathsf{Enc}(B_i)$*. Assuming that* S_1*'s inputs are the public key* $\mathsf{pk_p}$*, a set of encrypted scored items* $Q = \{\boldsymbol{E}(I_i)\}_{i\in[|Q|]}\}$*. Server* S_2 *has the public key* $\mathsf{pk_p}$ *and the secret key* $\mathsf{sk_p}$*. The execution of the protocol* SecDedup *between* S_1 *and* S_2 *enables* S_1 *to get a new list of encrypted distinct objects and their scores, that is, at the end of the protocol,* S_1 *outputs a new list of items* $\boldsymbol{E}(I'_1), ..., \boldsymbol{E}(I'_l)$*, and there does not exist* $i, j \in [l]$ *with* $i \neq j$ *such that* $o_i = o_j$*. Moreover, the new encrypted list should not affect the final top-k results.*

Algorithm 12: $\mathsf{SecDedup}\big(Q = \{\mathbf{E}(I_i)\}_{i\in[|Q|]}, \mathsf{pk_p}, \mathsf{sk_p}\big)$: De-duplication Protocol

S_1**'s input**: $\mathbf{E}(I_1), \ldots, \mathbf{E}(I_l)$, $\mathsf{pk_p}$
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$
S_1**'s ouput**: Output $\mathbf{E}(I'_1) \ldots \mathbf{E}(I'_l)$ without duplicated objects

1 **Server** S_1**:**
2 Let $|Q| = l$;
3 **for** $i = 1 \ldots l$ **do**
4 **for** $j = i+1, \ldots, l$ **do**
5 Compute $\mathsf{Enc}(b_{ij}) \leftarrow \big(\mathsf{EHL}(o_i) \ominus \mathsf{EHL}(o_j)\big)$;
6 Set the symmetric matrix $\mathbf{B}$ such that $\mathbf{B}_{ij} = \mathsf{Enc}(b_{ij})$;
7 S_1 generate it own public/private key $(\mathsf{pk}', \mathsf{sk}')$;
8 **for** *each* $\boldsymbol{E}(I_i)$ **do**
9 Generate random $\boldsymbol{\alpha}_i \in \mathbb{Z}_N^k$, β_i, $\gamma_i \in \mathbb{Z}_N$;
10 Compute $\mathbf{E}(\widetilde{I}_i) = (\mathsf{EHL}(\tilde{o}_i), \mathsf{Enc}(\widetilde{W}_i), \mathsf{Enc}(\widetilde{B}_i)) \leftarrow \mathsf{Rand}(E(I_i), \boldsymbol{\alpha}_i, \beta_i, \gamma_i)$;
11 Compute $H_i = \mathsf{Enc}_{\mathsf{pk}'}(\boldsymbol{\alpha}_i)||\mathsf{Enc}_{\mathsf{pk}'}(\beta_i)||\mathsf{Enc}_{\mathsf{pk}'}(\gamma_i)$ using pk';
12 Generate a random permutation $\pi : [l] \rightarrow [l]$;
13 Permute $\pi(\mathbf{B})$, i.e. permute $\mathbf{B}_{\pi(i)\pi(j)}$ for each B_{ij};
14 Permute $\mathbf{E}(\widetilde{I}_{\pi(i)})$ and $H_{\pi(i)}$ for $i \in [1, l]$;
15 Send $\pi(\mathbf{B})$, $\{\mathbf{E}(\widetilde{I}_{\pi(i)})\}_{i=1}^l, \{H_{\pi(i)}\}_{i=1}^l$, pk' to S_2;
16 **Server** S_2**:**
17 Receive $\pi(\mathbf{B})$, $\{\mathbf{E}(\widetilde{I}_{\pi(i)})\}_{i=1}^l$, $\{H_{\pi(i)}\}_{i=1}^l$, and pk' from S_1;
18 **for** *upper triangle of* $\pi(\boldsymbol{B})$ **do**
19 decrypt $b_{\pi(i)\pi(j)} := \mathsf{Dec}_{\mathsf{sk_p}}(\mathbf{B}_{\pi(i)\pi(j)})$;
20 **if** $b_{\pi(i)\pi(j)} = 0$ **then**
21 remove $\mathbf{E}(\widetilde{I}_{\pi(i)}), H_{\pi(i)}$;
`/* Deduplicate items */`
22 randomly generate o_i, and $\boldsymbol{\alpha}_i \in \mathbb{Z}_N^k$, $\beta_i, \gamma_i \in \mathbb{Z}_N$;
23 set $W_i = Z + \beta_i$ and $B_I = Z + \gamma_i$, where $Z = N - 1$;
24 Set $\mathbf{E}(I'_{\pi(i)}) := (\mathsf{EHL}(o) \odot \mathsf{Enc}(\boldsymbol{\alpha}_i), \mathsf{Enc}(W_i), \mathsf{Enc}(B_i))$;
25 Compute $H'_{\pi(i)} \leftarrow \mathsf{Enc}_{\mathsf{pk}'}(\boldsymbol{\alpha}_i)||\mathsf{Enc}_{\mathsf{pk}'}(\beta_i)||\mathsf{Enc}_{\mathsf{pk}'}(\gamma_i)$ using pk';
26 **for** *remaining* $\mathsf{Enc}(\widetilde{I}_{\pi(j)})$, $H_{\pi(j)}$ **do**
27 generate random $\boldsymbol{\alpha}'_i \in \mathbb{Z}_N^k$, β'_i, and $\gamma'_i \in \mathbb{Z}_N$;
28 $\mathsf{Enc}(I'_i) = (\mathsf{EHL}(o'_i), \mathsf{Enc}(W'_i), \mathsf{Enc}(B'_i)) \leftarrow \mathsf{Rand}(\mathsf{Enc}(\widetilde{I}_{\pi(j)}), \boldsymbol{\alpha}'_i, \beta'_i, \gamma'_i)$;
29 $H_{\pi(j)} = \mathsf{Enc}_{\mathsf{pk}'}(\boldsymbol{\alpha}_{\pi(j)}), \mathsf{Enc}_{\mathsf{pk}'}(\beta_{\pi(j)}), \mathsf{Enc}_{\mathsf{pk}'}(\gamma_{\pi(j)})$;
30 set $H'_i = \mathsf{Enc}_{\mathsf{pk}'}(\boldsymbol{\alpha}_{\pi(j)}) \cdot \mathsf{Enc}_{\mathsf{pk}'}(\boldsymbol{\alpha}'_i)||\mathsf{Enc}_{\mathsf{pk}'}(\beta_{\pi(j)}) \cdot \mathsf{Enc}_{\mathsf{pk}'}(\beta'_i)||\mathsf{Enc}_{\mathsf{pk}'}(\gamma_{\pi(j)}) \cdot \mathsf{Enc}_{\mathsf{pk}'}(\gamma'_i)$;
31 Generate a random permutation $\pi' : [l] \rightarrow [l]$. Permute new list $\mathbf{E}(I_{\pi'(i)})$ and $H'_{\pi'(i)}$, then send them back to S_1;
32 **Server** S_1**:**
33 Decrypt each $H'_{\pi'(i)}$ as $\boldsymbol{\alpha}'_{\pi'(i)}$, $\beta'_{\pi'(i)}$, $\gamma'_{\pi'(i)}$ using sk';
34 **foreach** $\boldsymbol{E}(I'_{\pi'(i)}) = \mathsf{EHL}(o'_{\pi'(i)}), \mathsf{Enc}(W'_{\pi'(i)}), \mathsf{Enc}(B'_{\pi'(i)})$ **do**
35 Run and get $\mathsf{Enc}(\hat{I}_i) = (\mathsf{EHL}(\hat{o}_i), \mathsf{Enc}(\widehat{W}_i), \mathsf{Enc}(\widehat{B}_i)) \leftarrow \mathsf{Rand}(\mathsf{Enc}(I'_{\pi'(i)}), -\boldsymbol{\alpha}'_{\pi'(i)}, -\beta'_{\pi'(i)}, -\gamma'_{\pi'(i)})$;
36 Output the encrypted list $\mathbf{E}(\hat{I}_1)...\mathbf{E}(\hat{I}_l)$;

Algorithm 13: $\mathsf{Rand}\big(\mathbf{E}(I), \boldsymbol{\alpha}, \beta, \gamma\big)$: Blinding the randomness

1 Let $\mathbf{E}(I) = (\mathsf{EHL}(o), \mathsf{Enc}(B), \mathsf{Enc}(W))$;
2 Compute $\mathbf{E}(\boldsymbol{\alpha}), \mathsf{Enc}(\beta), \mathsf{Enc}(\gamma)$;
3 Compute $\mathsf{EHL}(o) \leftarrow \mathsf{EHL}(o) \odot \mathsf{Enc}(\boldsymbol{\alpha})$, $\mathsf{Enc}(W) \leftarrow \mathsf{Enc}(W) \cdot \mathsf{Enc}(\beta)$, and $\mathsf{Enc}(B) \leftarrow \mathsf{Enc}(B) \cdot \mathsf{Enc}(\gamma)$;
4 Output $\mathbf{E}(I') = (\mathsf{EHL}(o), \mathsf{Enc}(W), \mathsf{Enc}(B))$;

Algorithm 14: A Secure Update Protocol $\mathsf{SecUpdate}\big(T^{d-1}, \Gamma^d, \mathsf{pk_p}, \mathsf{sk_p}\big)$

S_1**'s input**: $\mathsf{pk_p}$, T^{d-1}, Γ^d (encrypted list without duplicated objects)
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$
1 **Server** S_1**:**
2 Permute $\mathbf{E}(I_i) \in \Gamma^d$ as $\mathbf{E}(I_{\pi(i)})$ based on random permutation π;
3 **foreach** *each permute* $\boldsymbol{E}(I_{\pi(i)})$ **do**
4 **foreach** *each* $\boldsymbol{E}(I_j) \in T^{d-1}$ **do**
5 Let $\mathsf{Enc}(W_i)$, $\mathsf{Enc}(B_i)$ be encrypted worst/best score in $\mathbf{E}(I_{\pi(i)})$, and let $\mathsf{Enc}(W_j), \mathsf{Enc}(B_j)$ be encrypted worst/best score in $\mathbf{E}(I_j)$;
6 Compute $\mathsf{Enc}(b_{ij}) \leftarrow \mathsf{EHL}(I_{\pi(i)}) \ominus \mathsf{EHL}(I_j)$, send $\mathsf{Enc}(b_{ij})$ to S_2 and get $\mathsf{E}^2\big(t_{ij}\big)$;
7 Compute $\mathsf{E}^2\big(\mathsf{Enc}(W_i')\big) \leftarrow \mathsf{E}^2\big(t_{ij}\big)^{\mathsf{Enc}(W_i)}$, $\mathsf{Enc}(W_i') \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(W_i')\big), \mathsf{pk_p}, \mathsf{sk_p})$, $\mathsf{Enc}(W_j') \leftarrow \mathsf{Enc}(W_j)\mathsf{Enc}(W_i')$;
8 Compute $\mathsf{E}^2\big(\mathsf{Enc}(B_j')\big) \leftarrow \mathsf{E}^2\big(t_{ij}\big)^{\mathsf{Enc}(B_i)} \big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_{ij}\big)^{-1}\big)^{\mathsf{Enc}(B_j)}$ $\mathsf{Enc}(B_j') \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(B_j')\big), \mathsf{pk_p}, \mathsf{sk_p})$;
9 Set $\mathsf{Enc}(W_j'), \mathsf{Enc}(B_j')$ as the updated score for $\mathsf{Enc}(I_j)$;
10 compute $\mathsf{E}^2\big(\mathsf{Enc}(W_i')\big) \leftarrow \mathsf{E}^2\big(t_{ij}\big)^{\mathsf{Enc}(W_i)}\big(\mathsf{E}^2\big(1\big)\mathsf{E}^2\big(t_{ij}\big)^{-1}\big)^{\mathsf{Enc}(W_j')}$, run $\mathsf{Enc}(W_i') \leftarrow \mathsf{RecoverEnc}(\mathsf{E}^2\big(\mathsf{Enc}(W_i')\big), \mathsf{pk_p}, \mathsf{sk_p})$ and maintain $\mathsf{Enc}(W_i')$ for each $E(I_{\pi(i)})$
11 Update the encrypted worst score to $\mathsf{Enc}(W_i')$ for each $\mathsf{Enc}(I_{\pi(i)})$ and keep the original best score $\mathsf{Enc}(B_i)$;
12 Append the updated $\mathsf{Enc}(I_{\pi(i)})$ to T^{d-1} and get T^d;
13 S_1 and S_2 execute $\mathsf{SecDedup}(T^d, \mathsf{pk_p}, \mathsf{sk_p})$ and get the updated list T^d;
14 S_1 finally outputs T^d.
15 **Server** S_2**:**
16 **foreach** $\mathsf{Enc}(b_i)$ *received from* S_1 **do**
17 Decrypt to get b_i;
18 If $b_i = 0$, set $t_i = 1$, otherwise, set $t_i = 0$. Send $\mathsf{E}^2\big(t_i\big)$ to S_1.

Intuitively, at a high level, SecDedup let S_2 obliviously find the duplicated objects and its scores, and replaces the object id with a random value and its score with a large enough value $Z = N - 1 \in \mathbb{Z}_N$ (the largest value in the message space) such that, after sorting the worst scores, it will definitely not appear in the top-k list.

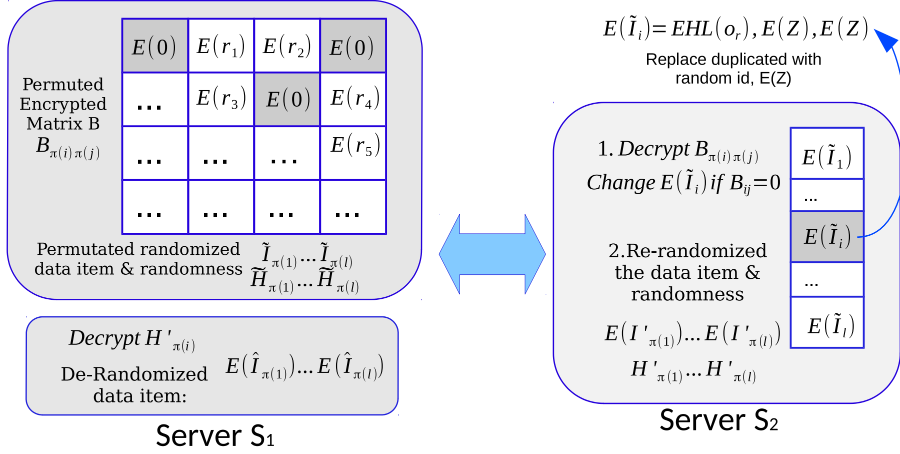


Figure 4.3: Overview of the SecDedup protocol

Figure 4.3 gives the overview of our approach. The technical challenge here is to allow S_2 to find the duplicated objects without letting S_1 know which objects have been changed. The idea is to let the server S_1 send a encrypted permuted matrix $\mathbf{B}$, which describes the pairwise equality relations between the objects in the list. S_1 then use the same permutation to permute the list of blinded encrypted items before sending it to S_2. This prevents S_2 from knowing the original data. For the duplicated objects, S_2 replace the scores with a large enough encrypted worst score. On the other hand, after deduplication, S_2 also has to blind the data items as well to prevent S_1 from knowing which items are the duplicated ones. S_1 finally gets the encrypted items without duplication. Algorithm 12 describes the detailed protocol.

We briefly discuss the execution of the protocol as follows: S_1 first fill the entry $\mathbf{B}_{ij}$ by computing $\mathsf{EHL}(o_i) \ominus \mathsf{EHL}(o_j)$. Note that, since the encrypted $\mathbf{B}$ is symmetric matrix indicating the equality relations for the list, therefore, S_1 only need fill the upper triangular for $\mathbf{B}$. and lower triangular can be filled by the fact that $\mathbf{B}_{ij} = \mathbf{B}_{ji}$. In addition, S_1 blinds

the encrypted item $\mathsf{Enc}(I_i)$ by homomorphically adding random values and get $\mathsf{Enc}(\widetilde{I}_i)$. This prevents S_2 from knowing the values of the item since S_2 has the secret key. Moreover, S_1 encrypts the randomnesses using his own public key pk' and get H_i. To hide the relation pattern between the objects in the list, S_1 applies a random permutation π to the matrix $\mathbf{B}_{\pi(i)\pi(j)}$, as well as $\mathsf{Enc}(I_{\pi(i)})$ and $H_{\pi(i)}$. Receiving the ciphertext, S_2 only needs to decrypt the upper triangular of the matrix, S_2 only keeps one copy of the $\mathsf{Enc}(\widetilde{I}_{\pi(i)})$, $H_{\pi(i)}$ and $\mathsf{Enc}(\widetilde{I}_{\pi(j)})$, $H_{\pi(j)}$ if $b_{\pi(i)\pi(j)} = 0$. Without loss of generality, we keep $\mathsf{Enc}(\widetilde{I}_{\pi(j)})$, $H_{\pi(j)}$ and replace $\mathsf{Enc}(\widetilde{I}_{\pi(i)}), H_{\pi(i)}$ as line 22-25. For the unchanged item, S_2 blinds them using as well (see line 28-30). It worth noting that the randomnesses added by S_2 are to prevent S_1 from discovering which item has been changed or not. S_2 also randomly permute the list as well (line 31). S_1 homomorphically recovers the original values by decrypting the received $H'_{\pi'(i)}$ using his sk' (see line 35). S_1 eventually the new permuted list of encrypted items.

For the duplicated objects, the protocol replaces their object id with a random value, and its worst score with a large number Z. For the new encrypted items that S_2 replaced (line 22), $\mathsf{Enc}(\hat{I}_i) = (\mathsf{EHL}(\hat{o}_i), \mathsf{Enc}(\widehat{W}_i), \mathsf{Enc}(\widehat{B}_i))$, we show in the following that $\mathsf{Enc}(\widehat{W}_i)$ is indeed a new encryption of the permuted $\mathsf{Enc}(W_{\pi'(\pi(j))})$ for some $j \in [l]$. As we can see, the $\mathsf{Enc}(\widehat{W}_i)$ is permuted by S_2's random π', i.e. $\mathsf{Enc}(\widehat{W}_{\pi'(i)})$ (see line 31). Hence, it follows that:

$$\mathsf{Enc}(\widehat{W}_{\pi'(i)}) \sim \mathsf{Enc}\big(W'_{\pi'(i)} - \beta'_{\pi'(i)}\big) \tag{4.2}$$

$$\sim \mathsf{Enc}\big(W'_{\pi'(i)} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})\big) \tag{4.3}$$

$$\sim \mathsf{Enc}\big(\widetilde{W}_{\pi'(\pi(j))} + \beta_{\pi'(\pi(j))} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})\big) \tag{4.4}$$

$$\sim \mathsf{Enc}\big(W_{\pi'(\pi(j))}\big) + \beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)} - (\beta_{\pi'(\pi(j))} + \beta'_{\pi'(i)})\big) \tag{4.5}$$

$$\sim \mathsf{Enc}\big(W_{\pi'(\pi(j))}\big) \tag{4.6}$$

In particular, from Algorithm 12, we can see that Equation (4.2) holds due to line 35, Equation (4.3) holds since line 30 and 33, Equation (4.4) holds due to line 28, and Equation (4.5) holds because of line 10. On the other hand, for the duplicated items that S_1

has changed from line 22 to 25, by the homomorphic operations of S_1 at line 35, we have

$$\mathsf{Enc}(\widehat{W}_{\pi'(k)}) \sim \mathsf{Enc}(W'_{\pi'(k)} - \beta'_{\pi'(k)}) \sim \mathsf{Enc}(Z + \beta'_{\pi'(k)} - \beta'_{\pi'(k)}) \sim \mathsf{Enc}(Z)$$

Since Z is a very large enough number, this randomly generated objects definitely do not appear in the top-k list after sorting.

4.8.2.4 Secure Update

At each depth d, we need to update the current list of objects with the latest global worst/best scores. At a high level, S_1 has to update the encrypted list Γ^d from the state T^{d-1} (previous depth) to T^d, and appends the new encrypted items at this depth. Let Γ^d be the list of encrypted items with the encrypted worst/best scores S_1 get at depth d. Specifically, for each encrypted item $E(I_i) \in T^{d-1}$ and each $E(I_j) \in L_d$ at depth d, we update I_i's worst score by adding the worst from I_j and replace its best score with I_j's best score if $I_i = I_j$ since the worst score for I_j is the in-depth worst score and best score for I_j is the most updated best score. If $I_i \neq I_j$, we then simply append $E(I_j)$ with its scores to the list. Finally, we get the fresh T^d after depth d. We describe the SecUpdate protocol in Algorithm 14.

4.9 Security Discussion

In this thesis, we do not provide a proof of security for $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$ and leave it as future work. In particular, since our non-colluding two-server model is different than the standard model considered from searchable and structured encryption, a provable security treatment of our protocol requires new definitions against which to prove security. Intuitively, however, the security guarantee our protocol should achieve is that the encrypted database and query processing protocol should reveal no information about the plaintext database and queries to either server (S_1 or S_2) beyond some well-specified and reasonable leakage. It remains an interesting and important open question

to mathematically formalize such a notion and prove that our protocol satisfies it.

4.10 Query Optimization

In this section, we present some optimizations that improve the performance of our protocol. The optimizations are two-fold: 1) we optimize the efficiency of the protocol SecDedup at the expense of some additional privacy leakage, and 2) we propose batch processing of SecDupElim and EncSort to further improve the SecQuery.

4.10.1 Efficient SecDupElim

We now introduce the efficient protocol SecDupElim that provides similar functionality as SecDedup. Recall that, at each depth, S_1 runs SecDedup to deduplicate m encrypted objects, then after the execution of SecDedup S_1 still receives m items but without duplication, and add these m objects to the list T^d when running SecUpdate. Therefore, when we execute the costly sorting algorithm EncSort the size of list to sort has md elements at depth d.

The idea for SecDupElim is that instead of keeping the same number encrypted items m, SecDupElim *eliminates* the duplicated objects. In this way, the number of encrypted objects gets reduced, especially if there are many duplicated objects. The SecDupElim can be obtained by simply changing the SecDedup as follows: in Algorithm 12 at line 20, when S_2 observes that there exist duplicated objects, S_2 only keeps one copy of them. The algorithm works exactly the same as before but without performing the line 22-25. We also run SecDupElim instead of SecDedup at line 13 in the SecUpdate. That is, after secure update, we only keep the distinct objects with updated scores. Thus, the number of items to be sorted also decrease. Now by adapting SecDupElim, if there are many duplicated objects appear in the list, we have much fewer encrypted items to sort.

Remark on security. The SecDupElim leaks additional information to the server S_1. S_1 learns the *uniqueness pattern* $\mathsf{UP}^d(q_i)$ at depth d, where $\mathsf{UP}^d(q_i)$ denotes the number

of the unique objects that appear at current depth d. The distinct encrypted values at depth d are independent from all other depths, therefore, this protocol still protects the distribution of the original ER. In addition, due to the 're-encryptions' during the execution of the protocol, all the encryptions are fresh ones, i.e., there are not as the same as the encryptions from ER. Finally, we emphasize that nothing on the objects and their values have been revealed since they are all encrypted.

4.10.2 Batch Processing for SecQuery

In the query processing SecQuery, we observe that we do not need to run the protocols SecDupElim and EncSort for every depth. Since SecDupElim and EncSort are the most costly protocols in SecQuery, we can perform *batch processing* and execute them after a few depths and not at each depth. Our observation is that there is no need to deduplicate repeated objects at each scanned depth. If we perform the SecDupElim after certain depths of scanning, then the repeated objects will be eliminated, and those distinct encrypted objects with updated worst and best scores will be sorted by running EncSort. The protocol will remain correct. We introduce a parameter p such that $p \geq k$. The parameter p specifies where we need to run the SecDupElim and EncSort in the SecQuery protocol. That is, the server S_1 runs the SecQuery with S_2 the same as in Algorithm 8, except that every p depths we run line 9-12 in Algorithm 8 to check if the algorithm could halt. In addition, we can replace the SecDupElim with the original SecDedup in the batch processing for better privacy but at the cost of some efficiency.

Security. Compared to the optimization from SecDupElim, we show that the batching strategy provides more privacy than just running the SecDupElim alone. For query q, assuming that we compute the scores over m attributes. Recall that the $\mathsf{UP}^p(q)$ at depth p has been revealed to S_1 while running SecDupElim, therefore, after the first depth, in the worst case, S_1 learns that the objects at the first depth is the same object. To prevent this worst case leakage, we perform SecDupElim every p depth. Then S_1 learns there are p

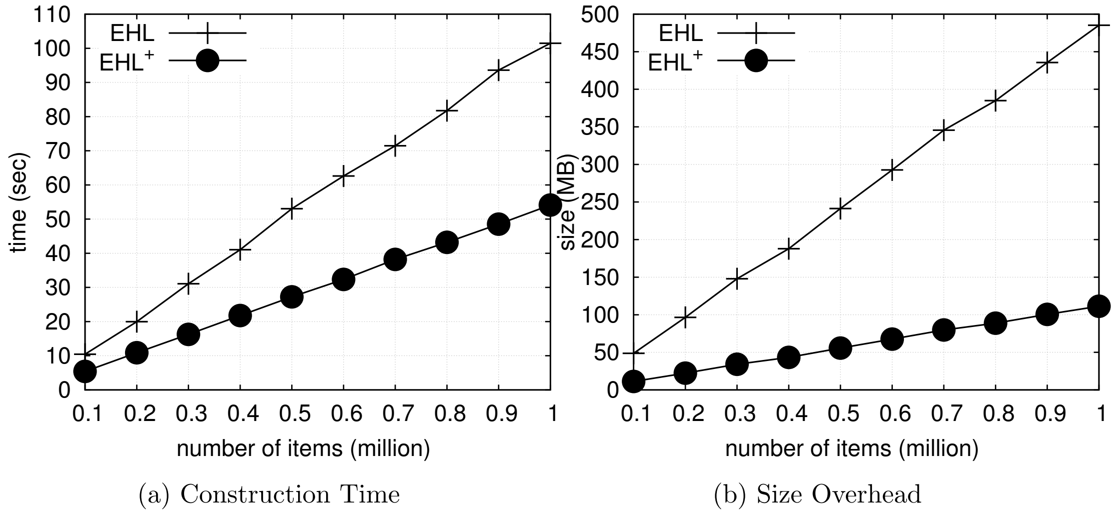
distinct objects in the worst case. After depth p, the probability that S_1 can correctly locate those distinct encrypted objects' positions in the table is at most $\frac{1}{(p!)^m}$. This decreases fast for bigger p. However, in practice this leakage is very small as many distinct objects appear every p depth. Similar to all our protocols, the encryptions are fresh due to the 're-encryption' by the server. Even though S_1 has some probability of guessing the distinct objects' location, the object id and their scores have not been revealed since they are all encrypted.

4.10.3 Efficiency

We analyze the efficiency of query execution. Suppose the client chooses m attributes for the query, therefore at each depth there are m objects. At depth d, it takes S_1 $O(m)$ for executing $\mathsf{SecWorst}$, $O(md)$ for executing $\mathsf{SecBest}$, $O(m^2)$ for $\mathsf{SecDedup}$, and $O(m^2d)$ for the $\mathsf{SecUpdate}$. The complexities for S_2 are similar. In addition, the $\mathsf{EncSort}$ has time overhead $O(m \log^2 m)$; however, we can further reduce to $O(\log^2 m)$ by adapting parallelism (see [Baldimtsi and Ohrimenko, 2014]). On the other hand, the $\mathsf{SecDupElim}$ only takes $O(u^2)$, where u is the number of distinct objects at this depth. Notice that most of the computations are multiplication (homomorphic addition), therefore, the cost of query processing is relatively small.

4.11 Experiments

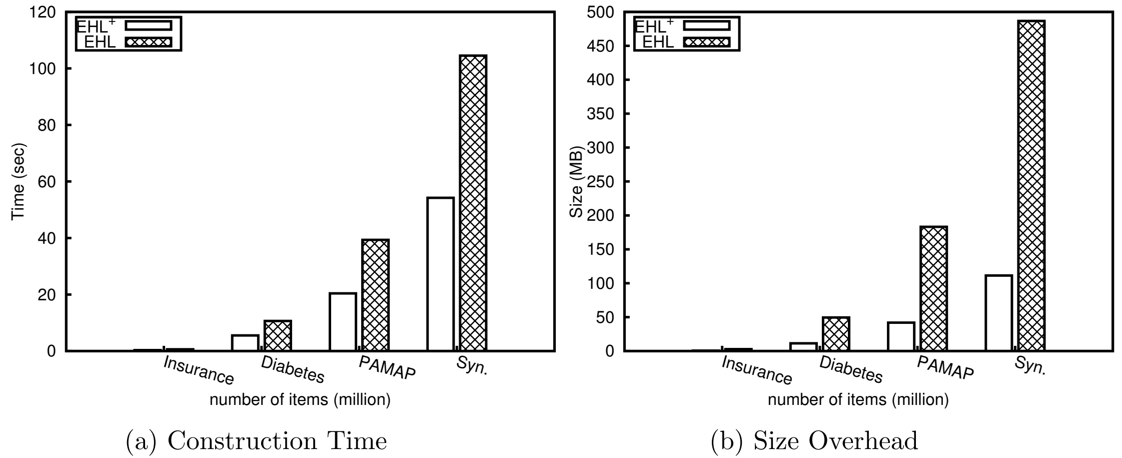
To evaluate the performance of our protocols, we conducted a set of experiments using real and synthetic datasets. We used HMAC-SHA-256 as the pseudo-random function (PRF) for the EHL and EHL^+ encoding, 512-bit security for the Paillier and DJ encryption schemes, and all experiments are implemented using C++. We implement the scheme $\mathsf{SecTopK} = (\mathsf{Enc}, \mathsf{Token}, \mathsf{SecQuery})$, including all the protocols $\mathsf{SecWorst}$, $\mathsf{SecBest}$, $\mathsf{EncSort}$, and $\mathsf{EncCompare}$ and their optimizations. For the server S_1, we run our experiments on a 24 core machine, that serves as the cloud, running Scientific Linux with 128GB memory



(a) Construction Time

(b) Size Overhead

Figure 4.4: Encryption using EHL vs. EHL$^+$.



(a) Construction Time

(b) Size Overhead

Figure 4.5: Encryption EHL vs. EHL$^+$ on real data

and 2.9GHz Intel Xeon. For the server S_2, we used a 9 core machine, running Scientific Linux with 64GB RAM and 3.4GHz Intel Xeon.

Data Sets We use the following real world datasets from UCI Machine Learning Repository [Lichman, 2013]. `insurance`: a benchmark dataset that contains 5822 customers' information from an insurance company and we extracted 13 attributes from the original dataset. `diabetes`: a patients dataset containing 101767 patients' records (i.e. data objects), where we extracted 10 attributes. `PAMAP`: a physical activity monitoring dataset that

contains 376416 objects, from which we extracted 15 attributes. We also generated synthetic datasets `synthetic` with 10 attributes that take values from a Gaussian distribution and the number of records are varied between 5 thousand to 1 million.

4.11.1 Evaluation of the Encryption Setup

We implemented both the EHL and the efficient EHL^+. For EHL, to minimize the false positives, we set the parameters as $H = 23$ and $s = 5$, where L is the size of the EHL and s is the number of secure hash functions. For EHL^+, we choose the number of secure hash function `HMAC` in EHL^+ to be $s = 5$, and, as discussed in the previous section, we obtained negligible false positive rate in practice. The encryption Enc is independent of the characteristics of the dataset and depends only on the size. Thus, we generated datasets such that the number of the objects range from 0.1 to 1 million. We compare the encryptions using EHL and EHL^+. After sorting the scores for each attribute, the encryption for each item can be fully parallelized. Therefore, when encrypting each dataset, we used 64 threads on the machine that we discussed before. Figure 4.4 shows that, both in terms of time and space, the cost of database encryption Enc is reasonable and scales linearly with the size of the database. Clearly, EHL^+ has less time and space overhead. For example, it only takes 54 seconds to encrypt 1 million records using EHL^+. The size is also reasonable, as the encrypted database only takes 111 MB using EHL^+. Figure 4.5 also shows the encryption time and size overhead for the real dataset that we used. Finally, we emphasize that the encryption only incurs a one-time off-line construction overhead.

4.11.2 Query Processing Performance

4.11.2.1 Query Performance and Methodology

We evaluate the performance of the secure query processing and their optimizations that we discussed before. In particular, we use the query algorithm without any optimization but with full privacy, denoted as Qry_F; the query algorithm running SecDupElim instead

of SecDedup at every depth, denoted as Qry_E; and the one using the batching strategies, denoted as Qry_Ba. We evaluate the query processing performance using all the datasets and use EHL^+ to encrypt all of the object ids.

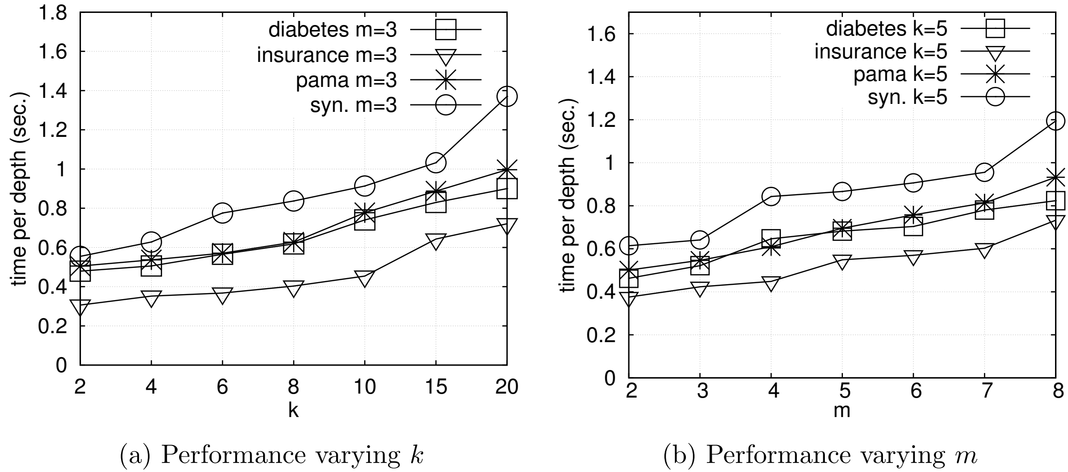
Notice that the performance of the NRA algorithm depends on the distribution of the dataset among other things. Therefore, to present a clear and simple comparison of the different methods, we measure the average time per depth for the query processing, i.e. $\frac{T}{D}$, where T is the total time that the program spends on executing a query and D is the total number of depths the program scanned before halting. In most of our experiments the value of D ranges between a few hundred and a few thousands. For each query, we randomly choose the number of attributes m that are used for the ranking function ranging from 2 to 8, and we also vary k between 2 and 20. The ranking function F that we use is the sum function.

4.11.2.2 Qry_F evaluation

We report the query processing performance without any query optimization. Figure 4.6 shows Qry_F query performance. The results are very promising considering that the query is executed completely on encrypted data. For a fixed number of attributes $m = 3$, the average time is about 1.30 seconds for the largest dataset `synthetic` running top-20 queries. When fixing $k = 5$, the average time per depth for all the dataset is below 1.20 seconds. As we can see, for fixed m, the performance scales linearly as k increases. Similarly, the query time also linearly increases as m gets larger for fixed k.

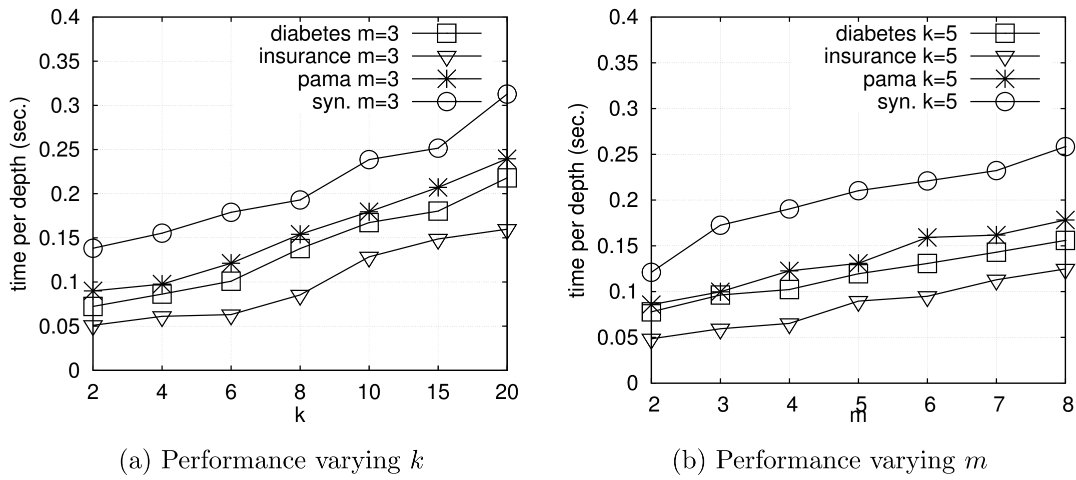
4.11.2.3 Qry_E evaluation

The experiments show that the SecDupElim improves the efficiency of the query processing. Figure 4.7 shows the querying overhead for exactly the same setting as before. Since Qry_E eliminates all the duplicated the items for each depth, Qry_E has been improved compared to the Qry_F above. As k increases, the performance for Qry_E executes up to 5 times faster than Qry_F when k increase to 20. On the other hand, fixing $k = 5$, the performance of



(a) Performance varying k (b) Performance varying m

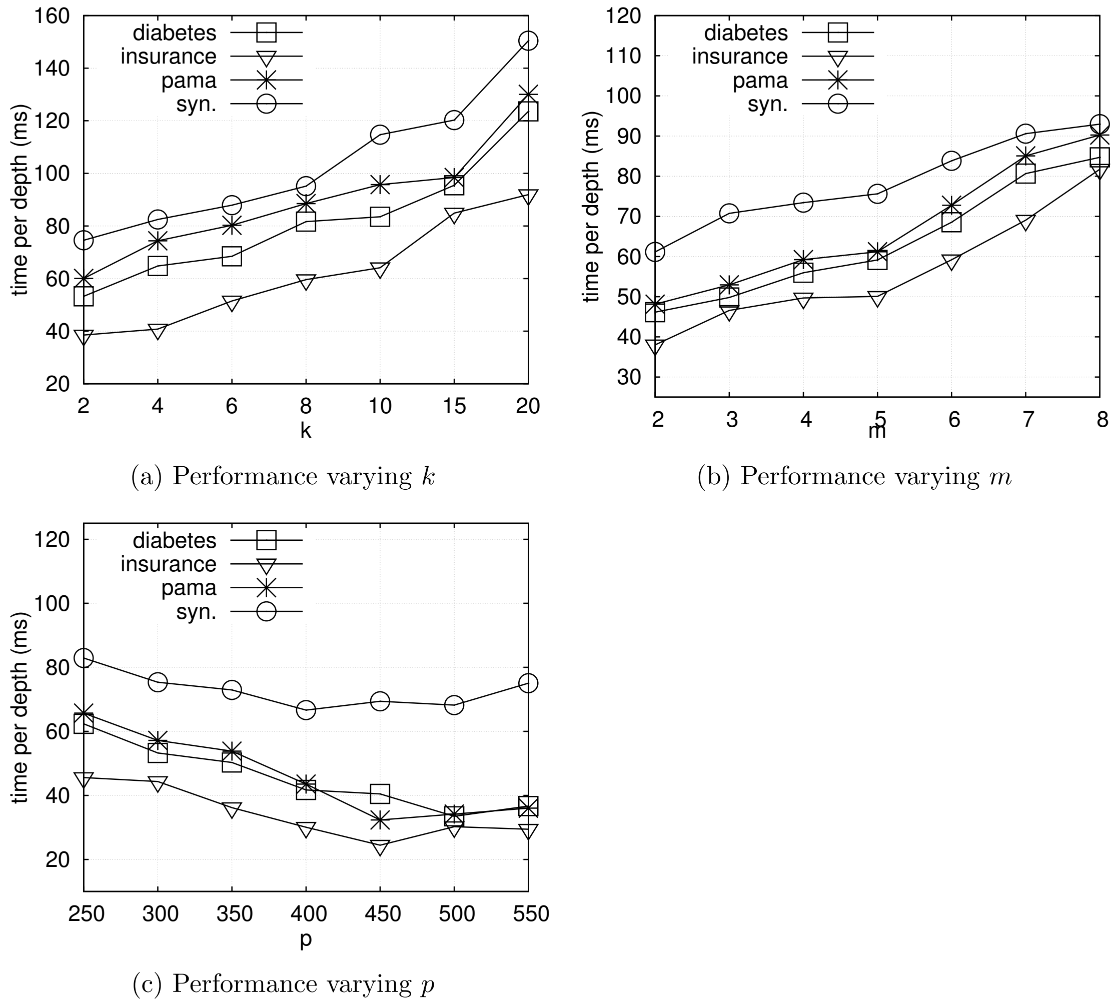
Figure 4.6: Qry_F query performance



(a) Performance varying k (b) Performance varying m

Figure 4.7: Qry_E query optimization performance

Qry_E can execute up to around 7 times faster than Qry_F as m grows to 20. In general, the experiments show that Qry_E effectively speeds up the query time 5 to 7 times over the basic approach.



(a) Performance varying k

(b) Performance varying m

(c) Performance varying p

Figure 4.8: Qry_Ba query optimization performance

4.11.2.4 Qry_Ba evaluation

We evaluate the effectiveness of batching optimization for the Qry_Ba queries. Figure 4.8 shows the query performance of the Qry_Ba for the same settings as the previous experiments. The experiments show that the batching technique further improves the perfor-

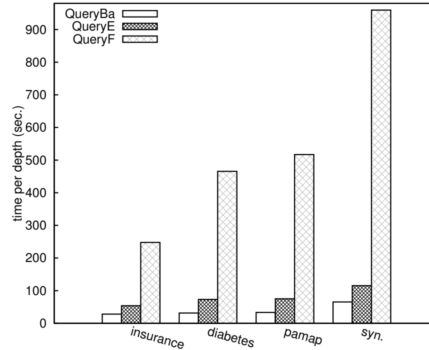


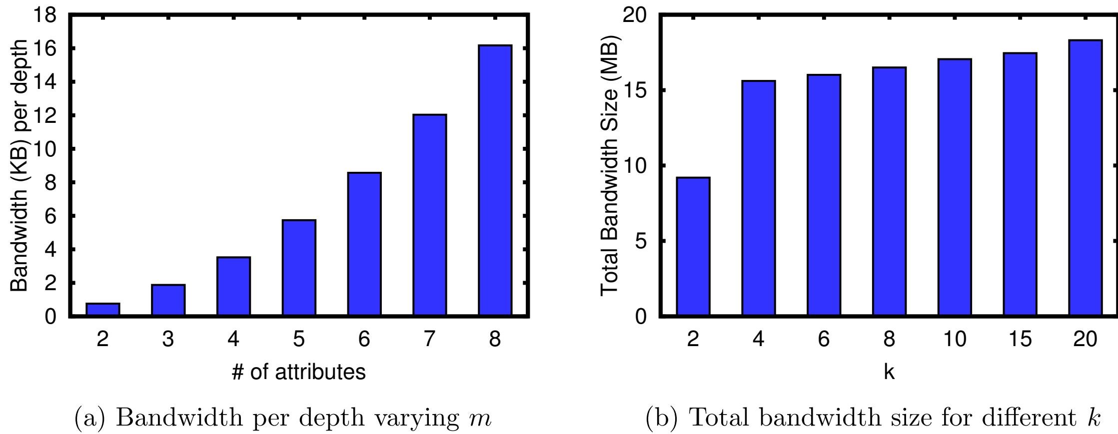
Figure 4.9: Comparisons ($k = 5$, $m = 2$, and $p = 500$)

mance. In particular, for fixed batching parameter $p = 150$, i.e. every 150 depths we perform SecDupElim and EncSort in the SecQuery, and we vary our k from 2 to 20. Compared to the Qry_E, the average time per depth for all of the datasets have been further improved. For example, when $k = 2$, the average time for the largest dataset `synthetic` is reduced to 74.5 milliseconds, while for Qry_F it takes more than 500 milliseconds . For `diabetes`, the average time is reduced to 53 milliseconds when $k = 2$ and 123.5 milliseconds when k increases to 20. As shown in figure 4.8a, the average time linearly increases as k gets larger. Similarly, when fixing the $k = 5$ and $p = 150$, for `synthetic` the performance per depth reduce to 61.1 milliseconds and 92.5 milliseconds when $m = 2$ and 8 separately. In Figure 4.8c, We further evaluate the parameter p. Ranging p from 200 to 550, the experiments show that the proper p can be chosen for better query performance. For example, the performance for `diabetes` achieves the best when $p = 450$. In general, for different dataset, there are different p's that can achieve the best query performance. When p gets larger, the number of calls for EncSort and SecDupElim are reduced, however, the performance for these two protocols also slow down as there're more encrypted items.

We finally compare the three queries' performance. Figure 4.9 shows the query performance when fixing $k = 5$, $m = 3$, and $p = 500$. Clearly, as we can see, Qry_Ba significantly improves the performance compared to Qry_F. For example, compared to Qry_F, the average running time is roughly 15 times faster for `PAMAP`.

4.11.2.5 Communication Bandwidth

Finally we show the experiments on the communication bandwidth and latency. *Our experiments show that the network latency is significantly less than the query computation cost.* We evaluate the communication on the fully secure and un-optimized Qry_Ba queries. Note that, for each depth, the bandwidth is the same since the duplicated encrypted objects are filled with encryptions of random values.



(a) Bandwidth per depth varying m

(b) Total bandwidth size for different k

Figure 4.10: Communication bandwidth evaluation

We evaluate the bandwidth on the largest dataset `synthetic`. Note that, the bandwidth per depth is independent of k since each depth this communication size only depends on m. As mentioned the bandwidth is $O(m^2)$, by varying m we show in Figure 4.10a the bandwidth per depth. In Figure 4.10b, we show the total bandwidth when executing the top-20 by fixing $m = 4$. As we can see, the total size of the bandwidth is very small, therefore, the total latency could be very small for a high-speed connection between the two clouds. By assuming a standard 50 Mbps LAN setting, we show in the table 4.3 below the total network latency between the servers S_1 and S_2 when $k = 20$ and $m = 4$.

dataset	total bandwidth (MB)	total network latency (in seconds)
`insurance`	8.87	1.41
`diabetes`	12.45	1.99
`PAMAP`	15.72	2.5152
`synthetic`	17.3	2.768

Table 4.3: Total Communication Network Latency for each dataset when $k = 20$, $m = 4$

The top-20 can be reported for all the dataset after a few thousands depth. Therefore, the average latency for each depth on every dataset is less than 1ms, which is significantly less than the query processing cost. Similar conclusions can be drawn for other parameter settings.

4.12 Top-k Join

We would like to briefly mention that our technique can be also extended to compute top-k join queries over multiple encrypted relations. Given a set of relations, $R_1, \ldots, R_L$, each tuple in R_i is associated with some score that gives it a rank within R_i. The *top-k join query* joins R_1 to R_L and produces the results ranked on a total score. The total score is computed according to some function, F, that combines individual scores. We consider only (i.e.*equi-join*) conditions in this paper. Similarly, the score function F we consider in this paper is also a linear combination over the attributes from the joining relations. A possible SQL-like join query example is as follows: `Q1 = SELECT * FROM A,B,C WHERE A.1=B.1 and B.2=C.3 ORDER BY A.1+B.2+C.4 STOP AFTER k;` where `A, B` and `C` are three relations and `A.1, B.1, B.2, C.3, C.4` are attributes on these relations. Our idea is to design a secure join operator, denoted as $\bowtie_{\mathsf{sec}}$, such that the server S_1 obliviously joins the relations based on the received token. S_1 has to invoke a protocol with S_2 to get the resulting joined results that meet the join condition.

4.12.1 Secure Top-k Join

We provide a description of the secure top-k join in this section. Since a join operator is implemented in most system as a dyadic (2-way) operator, we describe the secure top-k operator as a *binary* join operator between two relations R_1 and R_2. Consider an authorized client that wants to join two encrypted relations and get the top-k based on a join condition. Assume that each tuple in R_i has m_i many attributes and each R_i have n_i many tuples for $i = \{1, 2\}$. Furthermore, denote o^i_j be the jth objects in R_i and let $o^i_j.x_k$ be the kth attribute value.

4.12.2 Encryption Setup for Multiple databases

Algorithm 15: $\mathsf{Enc}(R_1, R_2)$: database encryption

1. Generate public/secret key $\mathsf{pk}_p, \mathsf{sk}_p$ for the pailliar encryption, generate random secret keys $\kappa_1, \ldots, \kappa_s$ for the EHL;
2. **foreach** *each* $o^i_j \in R_i$ **do**
3.   **foreach** *each attribute* $o^i_j.x_k$ **do**
4.   set $E(s_k) \leftarrow \langle \mathsf{EHL}(o^i_j.x_k), \mathsf{Enc}(o^i_j.x_k) \rangle$;
5.   set $E(o^i_j) = \big(E(s_1), ..., E(s_{m_i})\big)$
6. Generate a key K for the PRP P;
7. Permutes the encrypted attributes based on P, i.e. set $E(o^i_j) = \big(E(s_{P_{\mathsf{K}}(1)}), ..., E(s_{P_{\mathsf{K}}(m_i)})\big)$;
8. Output permuted encrypted databases as $\mathsf{ER}_1 = \{E(o^1_1)...E(o^1_{n_1})\}$ and $\mathsf{ER}_2 = \{E(o^2_1)...E(o^2_{n_2})\}$;

Consider a set of relations R_1 and R_2. The encryption setup is similar as the top-k for one relation. The difference is that since we have multiple relations on different data we cannot assign a global object identifier for each the objects in different relations. The difference here is that, in addition to encrypting an object id with EHL, we encrypt the attribute value using EHL since the join condition generated from the client is to join the relations based on the attribute values. Therefore, we can compare the equality between different records based on their attributes. The encryption $\mathsf{Enc}(R_1, R_2)$ is given in Algorithm 15.

The encrypted relations $\mathsf{ER}_1, \mathsf{ER}_2$ do not reveal anything besides the size. The proof is similar to the proof in Theorem 4.6.1.

4.12.3 Query Token

Consider a client that wants to run query a SQL-like top-k join as follows: `Q = SELECT * FROM R1, R2 WHERE R1.A = R2.B ORDER BY R1.C + R1.D STOP AFTER k;` where `A, C` are attributes in R_1 and `B, D` are attributes in R_2. The client first requests the key K for the P, then computes $(t_1, t_2, t_3, t_4) \leftarrow (P_{\mathsf{K}}(R_1.A), P_{\mathsf{K}}(R_2.B), P_{\mathsf{K}}(R_1.C), P_{\mathsf{K}}(R_2.D))$. Finally, the client generates the SQL-like query token as follows: t_Q = `SELECT * FROM` $\mathsf{ER}_1, \mathsf{ER}_2$ `WHERE` $\mathsf{ER}_1.t_1 = \mathsf{ER}_2.t_2$ `ORDERED BY` $\mathsf{ER}_1.t_3 + \mathsf{ER}_2.t_4$ `STOP AFTER` k. Then, the client sends the token t_Q to the server S_1.

4.12.4 Query Processing for top-k join

In this section, we introduce the secure top-k join operator $\bowtie_{\mathsf{sec}}$. We first introduce some notation that we use in the query processing algorithm. For a receiving token t_Q that is described in Section 4.12.4, let the join condition be $\mathsf{JC} \stackrel{\textbf{def}}{:=} \big(\mathsf{ER}_1.t_1 = \mathsf{ER}_2.t_2\big)$, and the score function $\mathsf{Score} = \mathsf{ER}_1.t_3 + \mathsf{ER}_4.t_4$. Moreover, for each $E(o_i^1) \in \mathsf{ER}_1$, let $E(x_{it_1})$ and $E(x_{it_3})$ be the t_1-th and t_3-th encrypted attribute. Similarly, let $E(x_{jt_2})$ and $E(x_{jt_4})$ be the t_2-th and t_4-th encrypted attribute for each $E(o_j^2)$ in ER_2. In addition, let $\mathbf{E}(X)$ be a vector of encryptions, i.e. $\mathbf{E}(X) = \langle \mathsf{Enc}(x_1), ..., \mathsf{Enc}(x_s) \rangle$, and let $\mathbf{E}(R) = \langle (\mathsf{Enc}(r_1), ..., \mathsf{Enc}(r_s)) \rangle$, where $R \in \mathbb{Z}_N^s$ with each $r_i \xleftarrow{\$} \mathbb{Z}_N$. Denote the randomization function Rand as below:

$$\begin{aligned}\mathsf{Rand}(\mathbf{E}(X), \mathbf{E}(R)) &= (\mathsf{Enc}(x_1) \cdot \mathsf{Enc}(r_1), ..., \mathsf{Enc}(x_n) \cdot \mathsf{Enc}(r_n)) \\ &= (\mathsf{Enc}(x_1 + r_1), ..., \mathsf{Enc}(x_n + r_n))\end{aligned}$$

. This function is similar to Rand in Algorithm 12 and is used to homomorphically blind the original value.

In general, the procedure for query processing includes the following steps:

Algorithm 16: $\mathsf{SecJoin}(\mathsf{tk}, \mathsf{pk_p}, \mathsf{sk_p})$: $\bowtie_{\mathsf{sec}}$ with $\mathsf{JC} = (\mathsf{ER}_1.t_1, \mathsf{ER}_2.t_2)$ and $\mathsf{Score} = \mathsf{ER}_1.t_3 + \mathsf{ER}_2.t_4$

S_1**'s input**: $\mathsf{pk_p}$, tk
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1 **Server** S_1**:**
2 Parse tk, let $\mathsf{JC} = (\mathsf{ER}_1.t_1, \mathsf{ER}_2.t_2)$ and $\mathsf{Score} = \mathsf{ER}_1.t_3 + \mathsf{ER}_2.t_4$
3 **foreach** $E(o_i^1) \in \mathsf{ER}_1$, $E(o_j^2) \in \mathsf{ER}_2$ *in random order* **do**
4 Let $E(o_i^1) = \big(E(x_{i1}), ...E(x_{im_1})\big)$ and $E(o_j^2) = \big(E(x_{j1}), ...E(x_{jm_2})\big)$;
5 Compute $\mathsf{Enc}(b_{ij}) \leftarrow \mathsf{EHL}(x_{it_1}) \ominus \mathsf{EHL}(x_{jt_2})$, where $E(x_{it_1})$, $E(x_{jt_2})$ are the t_1-th, t_2-th attributes in $E(o_i^1)$, $E(o_j^2)$;
/* evaluate $s_{ij} = b_{ij}(x_{it_3} + x_{jt_4})$ */
6 Send $\mathsf{Enc}(b_{ij})$ to S_2 and receive $\mathsf{E}^2\big(t_{ij}\big)$ from S_2;
7 Compute Score as: $s_{ij} \leftarrow \mathsf{E}^2\big(b_{ij}\big)^{\mathsf{Enc}(x_{it_3})\mathsf{Enc}(x_{jt_4})}$;
8 run $\mathsf{Enc}(s_{ij}) \leftarrow \mathsf{RecoverEnc}(S_{ij}, \mathsf{pk_p}, \mathsf{sk_p})$.
9 Combine rest of the attributes for $E(o_{ij})$ as follows: $x_l \leftarrow \mathsf{E}^2\big(b_{ij}\big)^{\mathsf{Enc}(x_l)}$, where $\mathsf{Enc}(x_l) \in \{\mathsf{Enc}(x_{i1})...\mathsf{Enc}(x_{jm_2})\}$ in $E(O_i^1)$, $E(O_j^2)$;
10 Run $\mathsf{Enc}(x_l) \leftarrow \mathsf{RecoverEnc}(x_l, \mathsf{pk_p}, \mathsf{sk_p})$.
11 **Server** S_2**:**
12 For each received t_{ij}, decrypts it. If it is 0, then compute $b_{ij} \leftarrow \mathsf{E}^2\big(1\big)$. Otherwise, $b_{ij} \leftarrow \mathsf{E}^2\big(0\big)$. Sends b_{ij} to S_1.
13 **Server** S_1**:**
14 Finally holds joined encrypted tuples $E(o_{ij}) = \mathsf{Enc}(s_{ij})$, $\{\mathsf{Enc}(x_l)\}$, where $\mathsf{Enc}(s_{ij})$ is the encrypted Score, $\{\mathsf{Enc}(x_l)\}$ are the joined attributes from $\mathsf{ER}_1, \mathsf{ER}_2$.
15 Run $L \leftarrow \mathsf{SecFilter}(\{E(o_{ij}), \mathsf{pk_p}, \mathsf{sk_p}\})$ and get the encrypted list L.
16 Run $\mathsf{EncSort}$ to conduct encrypted sort on encrypted Score $\mathsf{Enc}(s_{ij})$, and return top-k encrypted items.

- Perform the join on ER_1 and ER_2.
  - Receiving the token, S_1 runs the protocol with S_2 to generate all possible joined tuples from two relations and homomorphically computes the encrypted scores.
  - After getting all the joined tuples, S_1 runs $\mathsf{SecFilter}(\{E(o_i)\}, \mathsf{pk_p}, \mathsf{sk_p})$ (see Algorithm 17), which is a protocol with S_2 to eliminate the tuples that do not meet the join condition. S_1 and S_2 then runs the protocol $\mathsf{SecJoin}$. S_1 finally produce the encrypted join tuples together with their scores.

Algorithm 17: $\mathsf{SecFilter}\big(\{E(o_i)\}, \mathsf{pk_p}, \mathsf{sk_p}\big)$

S_1**'s input**: $\{E(o_i)\}$, $\mathsf{pk_p}$
S_2**'s input**: $\mathsf{pk_p}$, $\mathsf{sk_p}$

1 **Server** S_1**:**
2 Let $E(o_i) = \big(\mathsf{Enc}(s_i), \mathbf{E}(X_i)\big)$ where $\mathbf{E}(X_i) = \langle \mathsf{Enc}(x_{i1}), ..., \mathsf{Enc}(x_{is})\rangle$;
3 Generate a key pair $(\mathsf{pk}_s, \mathsf{sk}_s)$;
4 **foreach** $E(o_i)$ **do**
5 Generate random $r_i \in \mathbb{Z}_N^*$, and $R_i \in \mathbb{Z}_N^m$;
6 $\mathsf{Enc}(s_i') \leftarrow \mathsf{Enc}(s_i)^{r_i}$ and $\mathbf{E}(X_i') \leftarrow \mathsf{Rand}(\mathbf{E}(X_i), \mathbf{E}(R_i))$;
7 Set $E(o_i') = \big(\mathsf{Enc}(s_i'), \mathbf{E}(X_i')\big)$;
8 Compute the following: $\mathsf{Enc}_{\mathsf{pk}_s}(r_1^{-1}), \mathsf{Enc}_{\mathsf{pk}_s}(R_1), \ldots, \mathsf{Enc}_{\mathsf{pk}_s}(r_n^{-1}), \mathsf{Enc}_{\mathsf{pk}_s}(R_n)$;
9 Generate random permutation π, permute $E(o'_{\pi(i)})$, $\mathsf{Enc}_{\mathsf{pk}_s}(r^{-1}_{\pi(i)})$, and $\mathsf{Enc}_{\mathsf{pk}_s}(R_{\pi(i)})$;
10 Sends $E(o'_{\pi(i)})$, $\mathsf{Enc}_{\mathsf{pk}_s}(r^{-1}_{\pi(i)}), \mathsf{Enc}_{\mathsf{pk}_s}(R_{\pi(i)})$, and pk_s to S_2;

11 **Server** S_2**:**
12 Receiving the list $E(o'_{\pi(i)})$ and $\mathsf{Enc}_{\mathsf{pk}_s}(r_{\pi(i)}), \mathsf{Enc}_{\mathsf{pk}_s}(R_{\pi(i)})$;
13 **foreach** $E(o'_{\pi(i)}) \in \big(\mathsf{Enc}(s'_{\pi(i)}), \boldsymbol{E}(X'_{\pi(i)})\big)$ **do**
14 decrypt $b \leftarrow \mathsf{Enc}(s'_{\pi(i)})$;
15 **if** $b = 0$ **then**
16 Remove this entry $E(T'_{\pi(i)})$ and $\mathsf{Enc}_{\mathsf{pk}_s}(r^{-1}_{\pi(i)}), \mathsf{Enc}_{\mathsf{pk}_s}(R_{\pi(i)})$
17 **foreach** *remaining items* **do**
18 Generate random $\gamma_i \in \mathbb{Z}_N^*$, and $\Gamma_i \in \mathbb{Z}_N^m$;
19 $\mathsf{Enc}(\tilde{s}_i) \leftarrow \mathsf{Enc}(s'_{\pi(i)})^{\gamma_i}$ and $\mathbf{E}(\widetilde{X}_i) \leftarrow \mathsf{Rand}\big(\mathbf{E}(X'_{\pi(i)}), \mathbf{E}(\Gamma_i)\big)$;
20 Set $E(\tilde{o}_i) = \big(\mathsf{Enc}(\tilde{s}_i), \mathbf{E}(\widetilde{X}_i)\big)$;
`/* evaluate` $\widetilde{r}_i = r^{-1}_{\pi(i)}\gamma_i^{-1}$ `*/`
21 compute the following using pk_s: $\mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{r}_i) \leftarrow \mathsf{Enc}_{\mathsf{pk}_s}(r^{-1}_{\pi(i)})^{\gamma_i^{-1}}$;
`/* evaluate` $\widetilde{R}_i = R_{\pi(i)} + \Gamma_i^{-1}$ `*/`
22 $\mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{R}_i) \leftarrow \mathsf{Enc}_{\mathsf{pk}_s}(R_{\pi(i)}) \cdot \mathsf{Enc}_{\mathsf{pk}_s}(\Gamma_i)$
23 Sends the $E(\tilde{o}_i)$ and $\mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{r}_i), \mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{R}_i)$ to S_1

24 **Server** S_1**:**
25 **foreach** $E(\tilde{o}_i) = (\mathsf{Enc}(\tilde{s}), \boldsymbol{E}(\widetilde{X}_i))$ *and* $\mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{r}_i), \mathsf{Enc}_{\mathsf{pk}_s}(\widetilde{R}_i)$ **do**
26 use sk_s to decrypt $\mathsf{Enc}(\widetilde{r}_i)$ as $\widetilde{r}_i$, $\mathsf{Enc}(\widetilde{R}_i)$ as $\widetilde{R}_i$;
`/* homomorphically de-blind */`
27 compute $\mathsf{Enc}(\overline{s}_i) \leftarrow \mathsf{Enc}(\tilde{s}_i)^{\widetilde{r}_i}$ and $\mathbf{E}(\overline{X}_i) \leftarrow \mathsf{Rand}(\mathbf{E}(\widetilde{X}_i), \mathbf{E}(-\widetilde{R}_i))$;
28 Set $E(o_i') = (\mathsf{Enc}(\overline{s}_i), \mathbf{E}(\overline{X}_i))$;
`/* Suppose there're` l `tuples left */`
29 Output the list $E(o_1')$ ... $E(o_l')$.

- $\mathsf{EncSort}$: after securely joining all the databases, S_1 then runs the encrypted sorting protocol to get the top-k results.

The main $\bowtie_{\mathsf{sec}}$ is fully described in Algorithm 16. As mentioned earlier, since all the

attributes are encrypted, we cannot simply use the traditional join strategy. The merge-sort or hash based join cannot be applied here since all the tuples have been encrypted by a probabilistic encryption. Our idea for S_1 to securely produce the joined result is as follows: S_1 first combines all the tuples from two databases (say, using nested loop) by initiating the protocol $\mathsf{SecJoin}$. After that, S_1 holds all the combined tuples together with the scores. The joined tuple have $m_1 + m_2$ many attributes (or user selected attributes). Those tuples that meet the equi-join condition JC are successfully joined together with the encrypted scores that satisfy the Score function. However, for those tuples that do not meet the JC, their encrypted scores are homomorphically computed as $\mathsf{Enc}(0)$ and their joined attributes are all $\mathsf{Enc}(0)$ as well. S_1 holds all the possible combined tuples. Next, the $\mathsf{SecFilter}$ eliminates all of those tuples that do not satisfy JC. It is easy to see that similar techniques from $\mathsf{SecDupElim}$ can be applied here. At the end of the protocol, both S_1 and S_2 only learn the final number of the joined tuples that meet JC.

Below we describe the $\mathsf{SecJoin}$ and $\mathsf{SecFilter}$ protocols in detail. Receiving the Token, S_1 first parses it as the join condition $\mathsf{JC} = (\mathsf{ER}_1.t_1, \mathsf{ER}_2.t_2)$, and the score function $\mathsf{Score} = \mathsf{ER}_1.t_3 + \mathsf{ER}_2.t_4$. Then for each encrypted objects $E(o_{1i}) \in \mathsf{ER}_1$ and $E(o_{2i}) \in \mathsf{ER}_2$ in random order S_1 computes $t_{ij} \leftarrow \big(\mathsf{EHL}(x_{it_1}) \ominus \mathsf{EHL}(x_{jt_2})\big)^{r_{ij}}$, where x_{it_1} and x_{jt_2} are the value for the t_1th and t_2th attribute for $E(o_{1i})$ and $E(o_{2j})$ separately. r_{ij} is randomly generated value in $\mathbb{Z}_N^*$, then S_1 sends t_{ij} to S_2. Having the decryption key, S_2 decrypts it to b_{ij}, which indicates whether the encrypted value x_{it_1} and x_{jt_2} are equal or not. If $b_{ij} = 0$, then we have $x_{it_1} = x_{jt_2}$ which meets the join condition JC. Otherwise, b_{ij} is a random value. S_2 then encrypts the bit $b_i j$ using a double layered encryption and sends it to S_1, where $b_{ij} = 0$ if $x_{it_1} \neq x_{jt_2}$ otherwise $b_{ij} = 1$. Receiving the encryption, S_1 computes $S_{ij} \leftarrow \mathsf{E}^2\big(b_{ij}\big)^{\mathsf{Enc}(x_{1t_3})\mathsf{Enc}(x_{2t_4})}$, where x_{1t_3} is the t_3-th attribute for $E(o_{1i})$ and x_{2t_4} is the t_4-th attribute for $E(o_{2j})$. Finally, S_1 runs the $\mathsf{StripEnc}$ to get the normal encryption $\mathsf{Enc}(s_{ij})$.

Based on the construction,

$$\mathsf{Enc}(s_{ij}) \sim \mathsf{Enc}\big(b_{ij}(x_{1t_3} + x_{2t_4})\big), \text{ where } b_{ij} = \begin{cases} 1 & \text{if } x_{1t_1} = x_{2t_2} \\ 0 & \text{otherwise} \end{cases}$$

Finally, after fully combining the encrypted tuples, S_1 holds the joined encrypted tuple as well as the encrypted scores, i.e. $E(T) = (\mathsf{Enc}(s_{ij}),\ \mathsf{Enc}(x_{11})...\mathsf{Enc}(x_{1m_1}),$ $\mathsf{Enc}(x_{21}), ..., \mathsf{Enc}(x_{2m_2}))$. During the execution above, nothing has been revealed to S_1, S_2 only learns the number of tuples meets the join condition JC but does not which pairs since the S_1 sends out the encrypted values in random order. Also, notice that S_1 can only select interested attributes from ER_1 and ER_2 when combining the encrypted tuples. Here we describe the protocol in general.

After $\mathsf{SecJoin}$, assume S_1 holds n combined the tuples with each tuple has m combined attributes, then for each of tuple $E(T_i) = (\mathsf{Enc}(s_i), \mathbf{E}(X_i))$, where $\mathbf{E}(X_i)$ is the combined encrypted attributes $\mathbf{E}(X_i) = \langle \mathsf{Enc}(x_{i1}), ...\mathsf{Enc}(x_{im})\rangle$. Next, S_1 tries to blind encryptions in order to prevent S_2 from knowing the actual value. For each $E(T_i)$, S_1 generates random $r_i \in \mathbb{Z}_N^*$ and $R_i \in {\mathbb{Z}_N}^m$, and blinds the encryption by computing following: $\mathsf{Enc}(s'_i) \leftarrow$ $\mathsf{Enc}(s_i)^{r_i}$ and $\mathbf{E}(X'_i) \leftarrow \mathsf{Rand}(\mathbf{E}(X_i), \mathbf{E}(R_i))$. Then S_1 sets $E(T'_i) = \big(\mathsf{Enc}(s'_i), \mathsf{Enc}(X'_i)\big)$. Furthermore, S_1 generates a new key pair for the paillier encryption scheme $(\mathsf{pk}_s, \mathsf{sk}_s)$ and encrypts the following: $L = \mathsf{Enc}(r_1^{-1})_{\mathsf{pk}_s}, \mathsf{Enc}(R_1)_{\mathsf{pk}_s}, \ldots, \mathsf{Enc}(r_n^{-1})_{\mathsf{pk}_s}, \mathsf{Enc}(R_n)_{\mathsf{pk}_s}$, where each r_i^{-1} is the inverse of r_i in the group $\mathbb{Z}_N$. S_1 needs to encrypt the randomnesses in order to recover the original values, and we will explain this later. Moreover, S_1 generates a random permutation π, then permutes $E(T_{\pi(i)})$ and $\mathsf{Enc}(r_{\pi(i)}^{-1})_{\mathsf{pk}_s}, \mathsf{Enc}(R_{\pi(i)})_{\mathsf{pk}_s}$ for $i =$ $[1, n]$. S_1 sends the permuted encryptions to S_2.

S_2 receives all the encryptions. For each received $E(T'_{\pi(i)}) = (\mathsf{Enc}(s'_{\pi(i)}), \mathbf{E}(X'_{\pi(i)}))$, S_2 decrypts $\mathsf{Enc}(s'_{\pi(i)})$, if $s'_{\pi(i)}$ is 0 then S_2 removes tuple $E(T'_{\pi(i)})$ and corresponding $\mathsf{Enc}(r_{\pi(i)}^{-1})_{\mathsf{pk}_s}$, $\mathsf{Enc}(R_{\pi(i)})_{\mathsf{pk}_s}$. For the remaining tuples $E(T_{\pi(i)})$, S_2 generates random $r'_i \in \mathbb{Z}_N^*$, and $R'_i \in \mathbb{Z}_N^m$, and compute the following $\mathsf{Enc}(\tilde{s}_i) \leftarrow \mathsf{Enc}(s'_{\pi(i)})^{r'_i}$, $\mathsf{Enc}(\widetilde{X}_i) \leftarrow$

$\mathsf{Rand}\big(\mathbf{E}(X'_{\pi(i)}), \mathbf{E}(R'_i)\big)$ (see Algorithm 16 line 19). Then set $E(\widetilde{T}_i) = \big(\mathsf{Enc}(\tilde{s}_i), \mathsf{Enc}(\widetilde{X}_i)\big)$ Note that, this step prevents the S_1 from knowing which tuples have been removed. Also, in order to let S_1 recover the original values, S_2 encrypts and compute the following using pk_s, $\mathsf{Enc}(\widetilde{r}_i) \leftarrow \mathsf{Enc}(r^{-1}_{\pi(i)})^{r_i'^{-1}}_{\mathsf{pk}_s}$ and $\mathsf{Enc}(\widetilde{R}_i) \leftarrow \mathsf{Enc}(R'_{\pi(i)})_{\mathsf{pk}_s} \cdot \mathsf{Enc}(R'_i)_{\mathsf{pk}_s}$. Finally, S_1 sends the $E(\widetilde{T}_i)$ and $\mathsf{Enc}(\widetilde{r}_i), \mathsf{Enc}(\widetilde{R}_i)$ to S_1. Assuming there're n' joined tuples left. On the other side, S_1 receives the encrypted tuples, for each $E(\widetilde{T}_i) = \mathsf{Enc}(\tilde{s}), \mathsf{Enc}(\widetilde{X}_i)$ and $\mathsf{Enc}(\widetilde{r}_i), \mathsf{Enc}(\widetilde{R}_i)$, S_1 recovers the original values by computing the following: compute $\mathsf{Enc}(s'_i) \leftarrow \mathsf{Enc}(\tilde{s}_i)^{\widetilde{r}_i}$ and $\mathbf{E}(X'_i) \leftarrow \mathsf{Rand}(\mathbf{E}(\widetilde{X}_i), \mathbf{E}(-\widetilde{R}_i))$. Notice that, for the remaining encrypted tuples and their encrypted scores, S_1 can successfully recover the original value, we show below that the encrypted scores $\mathsf{Enc}(\overline{s}_j)$ is indeed some permuted $\mathsf{Enc}(s_{\pi(i)})$:

$$
\begin{aligned}
\mathsf{Enc}(\overline{s}_j) &\sim \mathsf{Enc}(\widetilde{r}_j \cdot \tilde{s}_j) && \text{(see Alg. 16 line 28)} \\
&\sim \mathsf{Enc}(r^{-1}_{\pi(j)} r_j'^{-1} \cdot \tilde{s}_j) && \text{(see Alg. 16 line 22)} \\
&\sim \mathsf{Enc}(r^{-1}_{\pi(j)} r_j'^{-1} \cdot s'_{\pi(j)} \cdot r'_j) && \text{(see Alg. 16 line 19)} \\
&\sim \mathsf{Enc}(r^{-1}_{\pi(j)} r_j'^{-1} \cdot s_{\pi(j)} r_{\pi(j)} r'_j) && \text{(see Alg. 16 line 6,10)} \\
&\sim \mathsf{Enc}(s_{\pi(j)})
\end{aligned}
$$

If we don't want to leak the number of tuples that meet JC, we can use a similar technique from SecDedup, that is, S_2 generates some random tuples and large enough random scores for the tuples to not satisfy JC. In this way, nothing else has been leaked to the servers. It is worth noting that the technique sketched above not only can be used for top-k join, but for any equality join can be applied here.

4.12.4.1 Performance Evaluation

We conduct the experiments under the same environment as in Section 4.11. We use synthetic datasets to evaluate our sec-join operator $\bowtie_{\mathsf{sec}}$: we uniformly generate R_1 with

$5K$ tuples and 10 attributes, and R_2 with $10K$ tuples and 15 attributes. Since the server runs the *oblivious join* that we discuss before over the encrypted databases, the performance of the $\bowtie_{\mathsf{sec}}$ does not depend on the parameter k. We test the effect of the joined attributes in the experiments. We vary the total number of the attribute m joined together from two tables. Figure 4.11 shows performance when m ranges from 5 to 20.

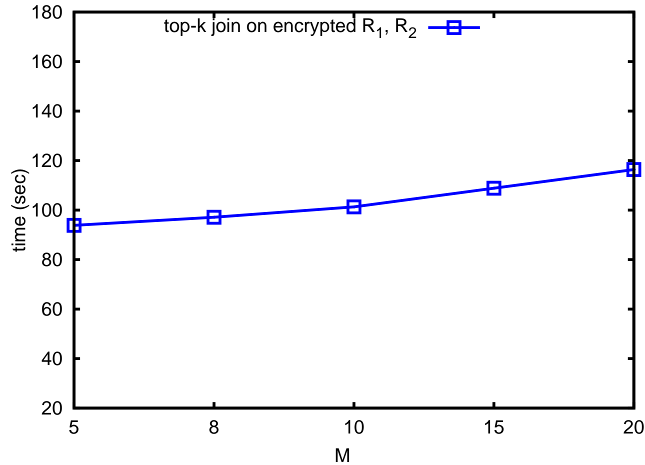


Figure 4.11: Top-k join: $\bowtie_{\mathsf{sec}}$

Our operator $\bowtie_{\mathsf{sec}}$ is generically designed for joining any attributes between two relations. In practice, one would be only interested in joining two tables using primary-key-to-foreign-key join or foreign-key-to-primary-key join. Our methods can be easily generalized to those joins. In addition, one can also pre-sort the attributes to be ranked and save computations in the $\bowtie_{\mathsf{sec}}$ processing. We leave this as the future work of this thesis.

4.12.5 Related works on Secure Join

Many works have proposed for executing equi-joins over encrypted data. One recent work [Pang and Ding, 2014] proposed a privacy-preserving join on encrypted data. Their work mainly designed for the private join operation, therefore cannot support the top-k join. In addition, in [Pang and Ding, 2014], although the actual values for the joined records are not revealed, the server learns some equality pattern on the attributes if

records are successfully joined. In addition, [Pang and Ding, 2014] uses bilinear pairing during their query processing, thus it might cause high computation overhead for large datasets. CryptDB [Popa et al., 2011] is a well-known system for processing queries on encrypted data. MONOMI [Tu et al., 2013] is based on CryptDB with a special focus on efficient analytical query processing. [Kerschbaum et al., 2013] adapts the deterministic proxy re-encryption to provide the data confidentiality. The approaches using deterministic encryption directly leak the duplicates and, as a result, the equality patterns to the adversarial servers. [Wong et al., 2014] propose a secure query system SDB that protects the data confidentiality by decomposing the sensitive data into shares and can perform secure joins on shares of the data. However, it is unclear whether the system can perform top-k queries over the shares of the data. Other solutions such as Order-preserving encryption (OPE) [Boldyreva et al., 2011b, Agrawal et al., 2004] can also be adapted to secure top-k join, however, it is commonly not considered very secure on protecting the ranks of the score as the adversarial server directly learns the order of the attributes.

4.13 Top-k Query Processing Conclusion

This paper proposes the first complete scheme that executes top-k ranking queries over encrypted databases in the cloud. First, we describe a secure probabilistic data structure called encrypted hash list (EHL) that allows a cloud server to homomorphically check equality between two objects without learning anything about the original objects. Then, by adapting the well-known NRA algorithm, we propose a number of secure protocols that allow efficient top-k queries execution over encrypted data. The protocols proposed can securely compute the best/worst ranking scores and de-duplication of the replicated objects. Moreover, the protocols in this paper are stand-alone which means the protocols can be used for other applications besides the secure top-k query problem. The scheme is experimentally evaluated using real-world data sets which show the scheme is efficient and practical.

Chapter 5

Conclusions

Ensuring security and privacy is becoming a significant challenge for cloud computing, especially for users with sensitive and valuable data. This thesis focus on the development of privacy-enhancing technologies that minimize the amount of data being revealed when outsourcing massive datasets in cloud-based environments. In particular, this thesis investigates several practical and provably secure encryption schemes that allow the data owner to encrypting large-scale databases without losing the ability to query and retrieve it efficiently for authorized clients.

In this dissertation, we have formalized and proposed the graph encryption framework for graph databases. The graph encryption is to encrypt graph data in such a way that they can be privately queried. In particular, in Chapter 3, this thesis proposes a graph encryption scheme for approximate shortest distance queries, called GRECS. Such scheme allows the client to query the shortest distance between two nodes in an encrypted graph. The GRECS system consists of three different schemes and can support approximated shortest distance queries. By leveraging distance oracle structures, we present several encryption schemes with different trade-offs. These encryption schemes do not affect the distance approximation of the underlying distance oracles, and, at the same time, provide strong security guarantees based on the formal security definition from graph encryption. The first scheme is computationally efficient, while the second one is communication-efficient by adopting specialized homomorphic encryption schemes. This thesis also propose a third scheme which is both computational and communication-efficient but leaks some small amount of controlled information. The experimental results of using the encryption scheme

on many real world graph datasets are very promising. It demonstrates that the constructions are extremely efficient and scalable compared to state-of-the-art solutions. In fact, in most cases, GRECS can report better approximations than the original distance oracle, which makes the accuracy of the shortest distance quite high. For example, for 10,000 randomly generated queries, roughly 50% of the distances returned are the true shortest distances. Moreover, this thesis also explores how the techniques can be applied to other graph queries.

In Chapter 4, this thesis presents a secure top-k query processing protocol on encrypted databases under the non-colluding semi-honest clouds model. The thesis also has formulated and presented several novel secure sub-protocols, such as secure best/worst score and secure de-duplication, which can be adapted as stand-alone building blocks for many other applications. Furthermore, we extend the techniques to support secure top-k join queries over multiple encrypted databases. The results show that the protocol is extremely efficient and has very low computational overhead. Moreover, this dissertation presents the secure protocol for handling general secure join queries. The proposed scheme also improves the security of other existing works which adopt some less secure property-preserving encryptions. This thesis has evaluated schemes by implementing the proposed protocols and running a set of experiments on a number of real-world databases.

5.1 Future Directions

This dissertation so far has taken preliminary steps to dealing with some of the security and privacy issues that arise in databases and data mining. Nevertheless, in the future there are a number of fascinating open problems in the area that need to be addressed on how to provide large-scale data security and privacy in cloud computing. Some compelling problems that come to mind include:

Privacy-preserving Machine Learning One goal is to design a practical privacy-preserving machine learning system that enables machine learning algorithms to run on

encrypted data. Recently, a number of applied security and privacy research problems have appeared in data mining and machine learning. For example, in the cloud-based environment, a user with sensitive data wants to make an inference using a machine learning predictive model that is held by the cloud, without compromising the user's private information. To protect data confidentiality, the user encrypts the data and sends the ciphertexts to the cloud who runs the machine learning algorithm over the encrypted data. Existing solutions are mainly focused on specific machine learning models and rely on some high-degree homomorphic encryptions with extremely high performance overhead. However, by taking advantage of the structures of the original dataset, one can have an efficient structured encryption scheme for some algorithms running on specific data structures. Having many such encryption schemes as building blocks can result in a more efficient and more secure system. The goal would be to have a generic and modular approach that can combine those structured encryption schemes for machine learning tasks.

Verifiable Computation on Encrypted Databases This thesis mainly addresses the problem of providing strong privacy under semi-honest adversarial model. In the future, it will be compelling to explore the possibility of bringing verifiable computation to both structured and graph encryptions. A number of works have shown some preliminary results of designing verifiable computation for some particular homomorphic encryption schemes. However, these theoretical results are very inefficient and are very hard to apply to database and data mining applications. A future direction is to combine the techniques of searchable and graph encryption with the techniques used in verifiable computation. A long-term goal would be to have a practical verifiable encryption scheme for massive datasets that provides both data privacy and integrity, without losing the capability to query the datasets.

Leakage Mitigation The protocols given in this thesis come with certain leakages. Another direction is to explore techniques to further reduce leakage in privacy-preserving database systems and mitigate inference attacks on those systems. In addition, we still

don't know if there are any better ways of expressing the semantic meanings of leakage and quantifying the amount of leakage when querying encrypted databases. We have seen many proposed works on supporting rich SQL queries on encrypted databases by leveraging property-preserving encryptions. Those approaches have much weaker security guarantees. Resolving this issue seems a promising future direction.

List of Journal Abbreviations

ACM Comput. Surv.	ACM Computing Surveys
ACM Trans. Database Syst.	ACM Transactions on Database Systems
ACNS	International Conference on Applied Cryptography and Network Security
ASIACCS	ACM Conference on Computer and Communications Security
CIKM	ACM International Conference on Information and Knowledge Management
COSN	International conference on Online social networks
CODASPY	ACM Conference on Data and Applications Security and Privacy
CRYPTO	International Cryptology Conference
DBSec	Conference on Data and Applications Security and Privacy
EDBT	International Conference on Extending Database Technology
ESORICS	European Symposium on Research in Computer Security
FC	International Conference on Financial Cryptography and Data Security

ICDE	IEEE International Conference on Data Engineering
ICDM	IEEE International Conference on Data Mining
Inf. Syst.	Information Systems
J. ACM	Journal of the ACM
J. Assoc. Inf. Sci. Technol.	Journal of the Association for Information Science and Technology
J. Math. Sociol.	Journal of Mathematical Sociology
J. Com. Sec.	Journal of Computer Security
KDD	ACM International Conference on Knowledge Discovery and Data Mining
NDSS	The Network and Distributed System Symposium
OSDI	USENIX Symposium on Operating Systems Design and Implementation
Oakland S & P	IEEE Symposium on Security and Privacy
PAKDD	Pacific Asia Conference on Knowledge Discovery and Data Mining
PKC	International Conference on Practice and Theory of Public-Key Cryptography
PVLDB	Proceedings of the Very Large DataBase Endowment
PODS	ACM Symposium on Principles of Database Systems
SDM	SIAM International Conference on Data Mining

SIAM Rev.	SIAM Review
SIGMOD	ACM International Conference on Management of Data
SODA	ACM-SIAM Symposium on Discrete Algorithms
SOSP	ACM Symposium on Operating Systems Principles
STOC	ACM Symposium on Theory of Computing
TCC	International Conference on Theory of Cryptography Conference
TKDD	ACM Transactions on Knowledge Discovery from Data
TKDE	IEEE Transactions on Knowledge and Data Engineering
USENIX Security	USENIX Security Symposium
VLDB J.	Very Large DataBase Journal
WSDM	ACM International Conference on Web Search and Data Mining

Bibliography

[Adida and Wikström, 2007] Adida, B. and Wikström, D. (2007). How to shuffle in public. In *TCC*, pages 555–574.

[Aggarwal and Yu, 2008] Aggarwal, C. C. and Yu, P. S., editors (2008). *Privacy-Preserving Data Mining - Models and Algorithms*, volume 34 of *Advances in Database Systems*.

[Agrawal et al., 2011] Agrawal, D., El Abbadi, A., Emekci, F., Metwally, A., and Wang, S. (2011). Secure data management service on cloud computing infrastructures. In *New Frontiers in Information and Software as Services*, volume 74, pages 57–80.

[Agrawal et al., 2004] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2004). Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574.

[Aly et al., 2013] Aly, A., Cuvelier, E., Mawet, S., Pereira, O., and Vyve, M. V. (2013). Securely solving simple combinatorial graph problems. In *Financial Cryptography*, pages 239–257.

[Arasu et al., 2015] Arasu, A., Eguro, K., Joglekar, M., Kaushik, R., Kossmann, D., and Ramamurthy, R. (2015). Transaction processing on confidential data using cipherbase. In *ICDE*, pages 435–446.

[Bajaj and Sion, 2011] Bajaj, S. and Sion, R. (2011). Trusteddb:a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216.

[Baldimtsi and Ohrimenko, 2014] Baldimtsi, F. and Ohrimenko, O. (2014). Sorting and searching behind the curtain. In *Financial Cryptography*, volume 2014, page 1017.

[Blanton et al., 2013] Blanton, M., Steele, A., and Aliasgari, M. (2013). Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS*, pages 207–218.

[Boldyreva et al., 2011a] Boldyreva, A., Chenette, N., and O'Neill, A. (2011a). Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 578–595.

[Boldyreva et al., 2011b] Boldyreva, A., Chenette, N., and O'Neill, A. (2011b). Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO '11*, pages 578–595.

[Boneh et al., 2005] Boneh, D., Goh, E.-J., and Nissim, K. (2005). Evaluating 2-dnf formulas on ciphertexts. In *TCC 2005*, pages 325–342.

[Bost et al., 2015] Bost, R., Popa, R. A., Tu, S., and Goldwasser, S. (2015). Machine learning classification over encrypted data. In *NDSS 2015*.

[Bugiel et al., 2011] Bugiel, S., Nürnberger, S., Sadeghi, A., and Schneider, T. (2011). Twin clouds: Secure cloud computing with low latency. In *CMS*, pages 32–44.

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

[Cash et al., 2013a] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., and Steiner, M. (2013a). Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO '13*, pages 353–373.

[Cash et al., 2013b] Cash, D., Jarecki, S., Jutla, C. S., Krawczyk, H., Rosu, M.-C., and Steiner, M. (2013b). Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, pages 353–373.

[Chang and Mitzenmacher, 2005] Chang, Y. and Mitzenmacher, M. (2005). Privacy preserving keyword searches on remote encrypted data. In *ACNS '05*, pages 442–455. Springer.

[Chase and Kamara, 2010] Chase, M. and Kamara, S. (2010). Structured encryption and controlled disclosure. In *ASIACRYPT '10*, volume 6477, pages 577–594.

[Chechik, 2014] Chechik, S. (2014). Approximate distance oracles with constant query time. In *STOC*, pages 654–663.

[Cheng et al., 2010] Cheng, J., Fu, A. W.-C., and Liu, J. (2010). K-isomorphism: privacy preserving network publication against structural attacks. In *SIGMOD Conference*, pages 459–470.

[Choi et al., 2014] Choi, S., Ghinita, G., Lim, H., and Bertino, E. (2014). Secure knn query processing in untrusted cloud environments. *TKDE*, 26:2818–2831.

[Cohen, 2014] Cohen, E. (2014). All-distances sketches, revisited: Hip estimators for massive graphs analysis. In *PODS*, pages 88–99.

[Cohen et al., 2013] Cohen, E., Delling, D., Fuchs, F., Goldberg, A. V., Goldszmidt, M., and Werneck, R. F. (2013). Scalable similarity estimation in social networks: closeness, node labels, and random edge lengths. In *COSN*, pages 131–142.

[Cohen et al., 2003] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. (2003). Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355.

[Curtmola et al., 2006] Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2006). Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS 2006*, pages 79–88. ACM.

[Curtmola et al., 2011] Curtmola, R., Garay, J. A., Kamara, S., and Ostrovsky, R. (2011). Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934.

[Damgård and Jurik, 2001] Damgård, I. and Jurik, M. (2001). A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC*, pages 119–136.

[Das Sarma et al., 2010] Das Sarma, A., Gollapudi, S., Najork, M., and Panigrahy, R. (2010). A sketch-based distance oracle for web-scale graphs. In *WSDM*, pages 401–410.

[Dwork et al., 2006] Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284.

[Dwork and Nissim, 2004] Dwork, C. and Nissim, K. (2004). Privacy-preserving datamining on vertically partitioned databases. In *CRYPTO 2004, 2004, Proceedings*, pages 528–544.

[Elmehdwi et al., 2014] Elmehdwi, Y., Samanthula, B. K., and Jiang, W. (2014). Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*, pages 664–675.

[Fagin et al., 2001] Fagin, R., Lotem, A., and Naor, M. (2001). Optimal aggregation algorithms for middleware. In *SIGACT-SIGMOD-SIGART*.

[Feigenbaum et al., 2006] Feigenbaum, J., Ishai, Y., Malkin, T., Nissim, K., Strauss, M. J., and Wright, R. N. (2006). Secure multiparty computation of approximations. *ACM Transactions on Algorithms*, 2(3):435–472.

[Gao et al., 2011] Gao, J., Yu, J. X., Jin, R., Zhou, J., Wang, T., and Yang, D. (2011). Neighborhood-privacy protected shortest distance computing in cloud. In *SIGMOD*, pages 409–420.

[Gentry, 2009a] Gentry, C. (2009a). *A fully homomorphic encryption scheme*. PhD thesis, Stanford University.

[Gentry, 2009b] Gentry, C. (2009b). Fully homomorphic encryption using ideal lattices. In *STOC '09*, pages 169–178. ACM Press.

[Gentry et al., 2010] Gentry, C., Halevi, S., and Vaikuntanathan, V. (2010). A simple bgn-type cryptosystem from lwe. In *Advances in Cryptology - EUROCRYPT '10*, pages 506–522. Springer.

[Goh, 2003] Goh, E.-J. (2003). Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive. See http://eprint.iacr.org/2003/216.

[Goldreich and Ostrovsky, 1996] Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473.

[Hacigümüs et al., 2002] Hacigümüs, H., Iyer, B. R., Li, C., and Mehrotra, S. (2002). Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227.

[Halevi et al., 2001] Halevi, S., Krauthgamer, R., Kushilevitz, E., and Nissim, K. (2001). Private approximation of np-hard functions. In *STOC*, pages 550–559. ACM.

[Han et al., 2013] Han, W., Lee, S., Park, K., Lee, J., Kim, M., Kim, J., and Yu, H. (2013). Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85.

[Hang et al., 2015] Hang, I., Kerschbaum, F., and Damiani, E. (2015). ENKI: access control for encrypted query processing. In *SIGMOD*, pages 183–196.

[Hore et al., 2012] Hore, B., Mehrotra, S., Canim, M., and Kantarcioglu, M. (2012). Secure multidimensional range queries over outsourced data. *VLDB J.*, 21(3):333–358.

[Ilyas et al., 2008] Ilyas, I. F., Beskales, G., and Soliman, M. A. (2008). A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4).

[Islam et al., 2012] Islam, M. S., Kuzu, M., and Kantarcioglu, M. (2012). Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*.

[Jaideep Vaidya, 2008] Jaideep Vaidya, Murat Kantarcioglu, C. C. (2008). Privacy-preserving naïve bayes classification. *VLDB J.*, 17(4):879–898.

[Kamara and Papamanthou, 2013] Kamara, S. and Papamanthou, C. (2013). Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*.

[Kamara et al., 2012] Kamara, S., Papamanthou, C., and Roeder, T. (2012). Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press.

[Kasiviswanathan et al., 2013] Kasiviswanathan, S. P., Nissim, K., Raskhodnikova, S., and Smith, A. (2013). Analyzing graphs with node differential privacy. In *TCC*, pages 457–476.

[Katz and Lindell, 2008] Katz, J. and Lindell, Y. (2008). *Introduction to Modern Cryptography*. Chapman & Hall/CRC.

[Kerschbaum et al., 2013] Kerschbaum, F., Härterich, M., Grofig, P., Kohler, M., Schaad, A., Schröpfer, A., and Tighzert, W. (2013). Optimal re-encryption strategy for joins in encrypted databases. In *DBSec*, pages 195–210.

[Kurosawa and Ohtaki, 2012] Kurosawa, K. and Ohtaki, Y. (2012). Uc-secure searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '12)*, Lecture Notes in Computer Science, pages 285–298. Springer.

[Kuzu et al., 2014] Kuzu, M., Islam, M. S., and Kantarcioglu, M. (2014). Efficient privacy-aware search over encrypted databases. CODASPY, pages 249–256.

[Kyrola and Guestrin, 2014] Kyrola, A. and Guestrin, C. (2014). Graphchi-db: Simple design for a scalable graph database system - on just a PC. *CoRR*, abs/1403.0701.

[Leskovec et al., 2005] Leskovec, J., Kleinberg, J. M., and Faloutsos, C. (2005). Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187.

[Li et al., 2014] Li, R., Liu, A. X., Wang, A. L., and Bruhadeshwar, B. (2014). Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964.

[Lichman, 2013] Lichman, M. (2013). UCI machine learning repository.

[Lindell and Pinkas, 2000] Lindell, Y. and Pinkas, B. (2000). Privacy preserving data mining. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '00)*, pages 36–54, London, UK. Springer-Verlag.

[Liu et al., 2015a] Liu, A., Zheng, K., Li, L., Liu, G., Zhao, L., and Zhou, X. (2015a). Efficient secure similarity computation on encrypted trajectory data. In *ICDE*, pages 66–77.

[Liu et al., 2014] Liu, C., Huang, Y., Shi, E., Katz, J., and Hicks, M. W. (2014). Automating efficient ram-model secure computation. In *IEEE SP*, pages 623–638.

[Liu et al., 2015b] Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. (2015b). Oblivm: A programming framework for secure computation. In *IEEE SP*, pages 359–376.

[Liu and Terzi, 2008] Liu, K. and Terzi, E. (2008). Towards identity anonymization on graphs. In *SIGMOD Conference*, pages 93–106.

[Low et al., 2010] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146.

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

[Mouratidis and Yiu, 2012] Mouratidis, K. and Yiu, M. L. (2012). Shortest path computation with no information leakage. *PVLDB*, pages 692–703.

[Murat Kantarcioglu, 2004] Murat Kantarcioglu, C. C. (2004). Privacy-preserving distributed mining of association rules on horizontally partitioned data. *TKDE*, 16:1026–1037.

[Naveed et al., 2014] Naveed, M., Prabhakaran, M., and Gunter, C. (2014). Dynamic searchable encryption via blind storage. In *Oakland S& P 2014*.

[Paillier, 1999] Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – Eurocrypt '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer-Verlag.

[Pang and Ding, 2014] Pang, H. and Ding, X. (2014). Privacy-preserving ad-hoc equi-join on outsourced data. *ACM Trans. Database Syst.*, 39(3):23:1–23:40.

[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 *SOSP*, pages 85–100.

[Potamias et al., 2009] Potamias, M., Bonchi, F., Castillo, C., and Gionis, A. (2009). Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876.

[Przulj et al., 2004] Przulj, N., Wigle, D. A., and Jurisica, I. (2004). Functional topology in a network of protein interactions. *Bioinformatics*, 20(3):340–348.

[Qi et al., 2013] Qi, Z., Xiao, Y., Shao, B., and Wang, H. (2013). Toward a distance oracle for billion-node graphs. In *VLDB*, pages 61–72.

[Ren et al., 2015] Ren, L., Fletcher, C. W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and Devadas, S. (2015). Constants count: Practical improvements to oblivious ram. In *USENIX Security*.

[Rivest et al., 1978] Rivest, R., Adleman, L., and Dertouzos, M. (1978). On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180.

[Samanthula et al., 2014] Samanthula, B. K., Jiang, W., and Bertino, E. (2014). Privacy-preserving complex query evaluation over semantically secure encrypted data. In *ES-ORICS*, pages 400–418.

[Sarwat et al., 2012] Sarwat, M., Elnikety, S., He, Y., and Kliot, G. (2012). Horton: Online query execution engine for large distributed graphs. In *ICDE*, pages 1289–1292.

[Shanks, 1971] Shanks, D. (1971). Class number, a theory of factorization, and genera. In *1969 Number Theory Institute*, pages 415–440. Providence, R.I.

[Shao et al., 2013] Shao, B., Wang, H., and Li, Y. (2013). Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516.

[Shen and Yu, 2013] Shen, E. and Yu, T. (2013). Mining frequent graph patterns with differential privacy. In *KDD 2013*, pages 545–553.

[Shi et al., 2007] Shi, E., Bethencourt, J., Chan, H. T., Song, D. X., and Perrig, A. (2007). Multi-dimensional range query over encrypted data. In *IEEES&P*, pages 350–364.

[Song et al., 2000] Song, D., Wagner, D., and Perrig, A. (2000). Practical techniques for searching on encrypted data. In *Oakland S & P*, pages 44–55.

[Stefanov et al., 2014] Stefanov, E., Papamanthou, C., and Shi, E. (2014). Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*.

[Thorup and Zwick, 2005] Thorup, M. and Zwick, U. (2005). Approximate distance oracles. *Journal of the ACM*, 52(1):1–24.

[Tu et al., 2013] Tu, S., Kaashoek, M. F., Madden, S., and Zeldovich, N. (2013). Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300.

[Vaidya and Clifton, 2005] Vaidya, J. and Clifton, C. (2005). Privacy-preserving top-k queries. In *ICDE*, pages 545–546.

[Vaidya et al., 2008] Vaidya, J., Clifton, C., Kantarcioglu, M., and Patterson, A. S. (2008). Privacy-preserving decision trees over vertically partitioned data. *TKDD*, 2(3).

[Wang et al., 2014] Wang, X. S., Nayak, K., Liu, C., Chan, T. H., Shi, E., Stefanov, E., and Huang, Y. (2014). Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226.

[Wong et al., 2009] Wong, W. K., Cheung, D. W.-l., Kao, B., and Mamoulis, N. (2009). Secure knn computation on encrypted databases. In *SIGMOD*, pages 139–152.

[Wong et al., 2014] Wong, W. K., Kao, B., Cheung, D. W., Li, R., and Yiu, S. (2014). Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*, pages 1395–1406.

[Yao et al., 2013] Yao, B., Li, F., and Xiao, X. (2013). Secure nearest neighbor revisited. In *ICDE*, pages 733–744.

Curriculum Vitae of Xianrui Meng

Address	MCS 207,111 Cummington Mall Department of Computer Science, Boston University Boston, 02215, MA, US
Email	xmeng@bu.edu
Website	http://cs-people.bu.edu/xmeng

Education

Ph.D in Computer Science
· Boston University, Sep. 2010 – July. 2016
· Advisor: Prof. George Kollios
· Thesis Topic: *Privacy-preserving queries on encrypted databases*

Master of Science in Computer Science
· Boston University, Sep. 2010 – Jan. 2013
· Advisor: Prof. Steven Homer

Bachelor of Science in Mathematics and Computer Science
· Bloomsburg University of Pennsylvania, Sep. 2006 – May. 2010
· Advisor: Prof. William Calhoun

Research Interests

Database Security and Privacy, Applied Cryptography
Graph Database Management, Query Optimization, Cloud computing

Research Experience

Research Assistant, Sep. 15 – Jul. 16
· NSF CNS-1414119

Research Assistant, Sep. 14 – May. 15
· NSF CISE IIS-1320542

Research Assistant, Sep. 13 – Aug. 14
· NSF 1012910

Research Assistant, Sep. 12 – Aug. 13
· NSF CNS-1017529

Teaching Experience

Teaching Fellow, 2014 Spring
· CS 330: Introduction to Algorithms

Teaching Fellow, 2012 Fall
· CS 235: Algebraic Algorithms

Teaching Fellow, 2011 Fall
· CS 101: Introduction to Computer Science

Teaching Fellow, 2011 Spring
· CS 111: Introduction to Computer Science I

Teaching Fellow, 2010 Fall
· CS 101: Introduction to Computer Science

Services

External Reviewer
· SIGMOD: 2016, 2015, 2014
· TKDE: 2015, 2014
· VLDB 2015, 2014
· ICDE: 2016, 2015, 2014
· EDBT: 2016, 2015
· EUROCRYPT: 2013
· ASIACRYPT: 2014

Publications

1. **Xianrui Meng**, Haohan Zhu, George Kollios. *Secure Top-k Query Processing on Encrypted Databases.* eprint arXiv: CoRR abs/1510.05175. 2015. (Under Review)
2. Haohan Zhu, **Xianrui Meng**, George Kollios. *NED: An Inter-Graph Node Metric Based On Edit Distance.* eprint arXiv: CoRR abs/1602.02358. 2016. (Under Review)
3. **Xianrui Meng**, Seny Kamara, Kobbi Nissim, George Kollios. *GRECS: Approximate Shortest Distance Queries On Encrypted Graphs.* 22nd ACM Conference on Computer and Communications Security (CCS), Denver, Colorado, USA, October 2015
4. Haohan Zhu, **Xianrui Meng**, George Kollios. *Privacy Preserving Similarity Evaluation of Time Series Data.* EDBT 2014: 499-510.
5. Benjamin Fuller, **Xianrui Meng**, Leonid Reyzin. *Computational Fuzzy Extractors.* ASIACRYPT 2013.
6. **Xianrui Meng** *Security and Privacy Aspects Of Mobile Application For Post-Surgical Care.* Boston University Technical Report.