MONASH University

Privacy-Preserving Social Search: Primitives and Realisation

Shangqi Lai

Doctor of Philosophy

A thesis submitted for the degree of *Doctor of Philosophy* at
Monash University in *2020*
Clayton School of Information Technology

Copyright notice

Abstract

Massive data breaches in online social networks (OSNs) raise severe privacy concerns. Although data encryption can effectively prevent data leakage, it also blocks the functionality of social search, a key enabler for quality OSN services.

This research investigates how to support efficient and versatile social search functionality while protecting data confidentiality. In this thesis, we consider to design and build a social search framework that supports the search functionality on the encrypted social graph with a strong privacy guarantee. As a result, we present a practical graph database system for privacy-preserving social search (GraphSE^2). The proposed system adopts a distributed graph model and customised cryptographic primitives to support a group of atomic operations efficiently. Those operations can be composed to support more complicated real-world query operators over the encrypted data. This thesis builds a real-world application to show that the system supports a wide range of social search applications over a large-scale encrypted social graph with low overheads.

In addition, this thesis reviews the security of the underlying cryptographic primitives in the system. We realise that the OXT protocol, which is adapted to support boolean queries in GraphSE^2, has an existing attack based on its leakage. To address this attack, this thesis presents a new cryptographic scheme (HXT) for the proposed framework that eliminates the vulnerable leakage and supports the conjunctive query. Furthermore, HXT retains the efficiency when processing queries; deploying the HXT protocol only introduces a moderate overhead comparing to OXT.

Publications during enrolment

Publication(s) included in this thesis:

1. S. Lai, X. Yuan, A. Sakzad, M. Salehi, J. K. Liu, D. Liu. Enabling Efficient Privacy-Assured Outlier Detection over Encrypted Incremental Datasets. In IEEE Internet of Things Journal, 7(4):2651–2662 2020. https://doi.org/10.1109/JIOT.2019.2949374 (Journal Impact Factor=9.5)

2. S. Lai, X. Yuan, S-F. Sun, J. K. Liu, Y. Liu and D. Liu. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, pages 41–54, 2019. (Core Rank B, Accepted Ratio: 17%)

3. S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S-F. Sun, D. Liu, and C. Zuo. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 745–762, 2018. (Core Rank A*, Accepted Ratio: 16.6%)

Other publication(s) not included in this thesis:

1. S. Kermanshahi, J. K. Liu, R. Steinfeld, S. Nepal, S. Lai, R. Loh, and C. Zuo. Multi-client Cloud-based symmetric searchable encryption. In IEEE Transactions on Dependable and Secure Computing, 2019. https://doi.org/10.1109/TDSC.2019.2950934 (In press, Journal Impact Factor=6.4)

2. Z. Sui, S. Lai, C. Zuo, X. Yuan, J. K. Liu, and H. Qian. An Encrypted Database Framework with Enforced Access Control and Blockchain Validation. In Information Security and Cryptology, pages 260–273, 2018. (Core Rank B, Accepted Ratio: 38.7%)

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 2 original papers published in top-tier conferences and 1 original papers published in a peer-reviewed journal. The core theme of the thesis is the modelling and realisation of privacy-preserving social search systems. The ideas, development and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the Clayton School of Information Technology under the supervision of Assoc. Prof. Joseph Liu, Dr. Xingliang Yuan, Dr. Ron Steinfeld and Dr. Dongxi Liu.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapters 3, 4 and 5, my contribution to the work involved the following:

Thesis Chapter	Publication Title	Status	Nature and % of student contribution	Co-author name(s) Nature and % of Co-author's contribution	Co-author(s), Monash student
3	GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search	published	60%. concept, system design, implementation, experiment, security analysis, writing the manuscript	1) Xingliang Yuan. 15%. concept, security analysis and revising the manuscript 2) Shi-Feng Sun. 10%. security analysis 3) Joseph. K. Liu. 5%. revising the manuscript 4) Yuhong Liu. 5%. domain knowledge 5) Dongxi Liu. 5%. revising the manuscript	1) No 2) No 3) No 4) No 5) No

4	Enabling Efficient Privacy-Assured Outlier Detection over Encrypted Incremental Datasets	in press	60%. concept, system design, implementation, experiment, security analysis, writing the manuscript	1) Xingliang Yuan. 10%. concept and revising the manuscript 2) Amin Sakzad. 10%. concept and revising the manuscript 3) Mahsa Salehi. 10%. concept and domain knowledge 4) Joseph. K. Liu. 5%. revising the manuscript 5) Dongxi Liu. 5%. revising the manuscript	1) No 2) No 3) No 4) No 5) No
5	Result Pattern Hiding Searchable Encryption for Conjunctive Queries	published	60%. concept, system design, security analysis, implementation, experiment, writing the manuscript	1) Sikhar Patranabis. 10%. concept, security analysis and writing/revising the manuscript 2) Amin Sakzad. 10%. concept, security analysis and writing/revising the manuscript 3) Joseph. K. Liu. 2%. revising the manuscript 4) Debdeep Mukhopadhyay. 1%. supervision 5) Ron Steinfeld. 10%. security analysis and revising the manuscript 6) Shi-Feng Sun. 5%. concept, security analysis and writing/revising the manuscript 7) Dongxi Liu. 1%. supervision 8) Cong Zuo. 1%. implementation	1) No 2) No 3) No 4) No 5) No 6) No 7) No 8) Yes

I have renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

Student name: Shangqi Lai

Student signature: **Date:**

I hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Main Supervisor name: Joseph Liu

Main Supervisor signature: **Date:**

Acknowledgements

First of all, I would like to express my gratitude to my supervisors: Assoc. Prof. Joseph Liu, Dr. Xingliang Yuan, Dr. Ron Steinfeld and Dr. Dongxi Liu for their support and guidance throughout my PhD career. I learned a lot from them, ranging from the knowledge about cryptography and security to academic skills such as writing and presentation. I enjoyed the wonderful time I worked with them, and their experience and advice are invaluable for my study and my future career.

I am deeply grateful for the help from Dr. Shi-Feng Sun since he provided precious advice and feedback on the cryptographic aspect of my research. I also want to express my gratitude to my collaborators Dr. Yuhong Liu, Dr. Sikhar Patranabis, Dr. Amin Sakzad and Dr. Mahsa Salehi. They taught me the required domain knowledge on my research project and gave many constructive comments on my publications.

I also want to thank my panel members Assoc. Prof. Carsten Rudolph, Assoc. Prof. Chung-Hsing Yeh, Dr. Chunyang Chen and Dr. Jiangshan Yu for their insightful comments on my research. They have given me much useful feedback to help me finish my research project, as well as this thesis.

Meanwhile, I would take this chance to acknowledge Data61, CSIRO for the generous financial and resource support in the past three years.

Furthermore, I appreciate Ms Danette Deriane for her valuable time to help me arrange compulsory activities (commencement, course, milestone, etc..) and answer my enquiries before and after I started my study. I also thankful to Ms Julie Holden as her GSAS classes provide an excellent training opportunity for my academic communication skills. In addition, She also offers help when I am writing and revising this thesis. I am equally thankful to all the other graduation research staff from the Faculty of Information Technology for their kindly assistance throughout my PhD journey.

Finally, I would like to express my deepest appreciation to my parents and my partner Mingchen, who give unconditional love, encouragement and support not only in my three-year PhD career but also in many other things I have done in my life.

Contents

List of Figures

List of Tables

List of Acronyms

ABE Attribute-Based Encryption. 8

AWS Amazon Web Services. 1, 17, 60

BF Bloom Filter. 73, 79–81, 84–86, 88–90, 93, 97–99, 107

COT Correlated-Oblivious transfer. 24

DAUs Data Acquisition Units. 48–50, 53, 54

DBMS Database Management System. 10

DDH Decisional Diffie-Hellman. 67, 69, 70, 79, 84, 88, 93

EP Equality Pattern. 15, 83, 92

FHE Fully Homomorphic Encryption. 3, 9

GC Garbled Circuit. 13, 16, 19, 48

HVE Hidden Vector Encryption. 5, 64, 66–69, 73–75, 79, 80, 82, 84, 89, 90, 93, 94, 96–101, 104, 106, 107, 109

HXT Hidden Cross-Tags. II, 5, 6, 64, 66–69, 79, 81–84, 93, 97–100, 102–107, 110

IND-CPA Indistinguishability under Chosen-Plaintext Attack. 69, 70, 76, 84, 86, 93

IoT Internet of Things. 43, 44

IP Intersection Pattern. 16, 66, 83, 91, 92

kNN k-Nearest Neighbours. 44, 51–53, 55, 57–62

KPRP Keyword Pair Result Pattern. 65–68, 72, 81, 83, 84, 110

ORAM Oblivious RAM. 3, 13, 66

OSN Online Social Network. II, 1–4, 7–9, 11, 19, 43, 108, 110

OT Oblivious Transfer. 16, 24, 26, 27, 33–37, 44, 48, 58–60

OXT Oblivious Cross-Tags. II, 4, 5, 15, 19–23, 30–36, 38, 42, 64–68, 72, 79–83, 93, 97–100, 102–107, 110

P2P Peer-to-Peer. 7

PIR Private Information Retrieval. 66

PPE Property-Preserving Encryption. 13

PPOD Privacy-Preserving Outlier Detection. 5, 43–45, 47–51, 56, 58–63

PPT Probabilistic Polynomial-Time. 32, 70, 72–75, 79, 82

PRF Pseudorandom Function. 15, 37, 67, 69–71, 76, 79, 82, 84, 86, 93, 94, 97, 98

RP Result Pattern. 16, 65, 66, 72, 83, 84

SHVE Symmetric Hidden Vector Encryption. 75–77, 79, 93, 94, 96, 97, 100, 101, 107, 109

SNS Social Network Services. 1, 7

SP Size Pattern. 16, 66, 83, 92

SSE Searchable Symmetric Encryption. 4–6, 9, 13, 15, 17, 22, 30, 31, 35, 64–66, 68–73, 77, 79, 82, 107–109

WRP whole result pattern. 66, 67, 72, 83, 84, 92, 93, 107

XP X-terms Size Pattern. 16

Chapter 1

Introduction

1.1 Motivation

Online Social Network (OSN) is highly popular for many years, and numerous Internet users have started to use OSN as their medium of communication. In addition to providing a convenient tool for communication, OSN providers also design diverse Social Network Services (SNS) for users to interact virtually: it includes updating activities and location information, sharing multimedia (e.g. photo, video) during some events, getting updates and comments on activities by friends. As a result, OSN attracts a huge amount of users to form an enormous community: In 2017, the monthly active user of Facebook reached 2 billion, which means over one half Internet users are using the services from Facebook [2, 3], and this is just an epitome of the today's OSN dependency.

However, OSN also becomes an attractive and convenient way for attackers to harvest sensitive data, while OSN users enjoy using OSN services. In 2005, two researchers from Carnegie Mellon University (CMU) conducted a study in the online behaviour of more than 4,000 students in the university who had signed up on Facebook [4]. In their study, they found that the majority of Facebook users at CMU provided an astonishing amount of sensitive information in their Facebook profiles. For example, users reveal their real names (over 90%), dates of birth (87%) and personal images (80%) without any protection, which makes these users searchable and identifiable to the vast majority of users in the network. Recently, massive data breaches have become a common case in OSN [5, 6]. Furthermore, driven by the demands on huge storage and computation resources, OSN service providers utilise public commercial clouds as their backend data storage [7–9], which further broadens the attack plane [10]. For instance, LocalBox is reported to store its data in AWS S3 without any protection (unencrypted, public accessible). As a result, more than 48 million users' private data is exposing to the public [5].

Due to the richness and variety of sensitive information in OSN, the privacy of these data is of critical importance to OSN users: the leakage of sensitive information will put OSN users at risk for many attacks from the cyberspace or even the real world. For instance, the precise personal information collecting from OSN helps potential adversaries to construct more deceitful fraud messages (e.g. Context-Aware Spam [11]). It also can be used to re-identify some anonymous datasets like hospital medical information by matching the common attributes [12]. Additionally, OSN users may be easily stalked as they revealed their location or timetable to others in OSN.

One potential approach to preventing data leakage is encryption. It can effectively stop

the unauthorised access since the unauthorised (no encryption key) attacker who gains full control on the OSN server only can see the encrypted data. However, this approach impairs the functionality of *social search*, a key enabler for quality OSN services. In particular, social search enables users to search the content of interests created by their friends, and it is widely deployed by OSN providers such as Facebook [13] and LinkedIn [14]. Compared with traditional web search, it produces personalised search results and serves for a wide range of OSN services such as friend discovering and user targeting.

In a nutshell, to enable privacy-preserving social search over the encrypted data, it is urged to design a system that can 1) access the encrypted data; 2) compute on the retrieved data; 3) support various social search-related services. On the other hand, most of existing encrypted search systems are either dedicated to the generic encrypted databases [15–18] or a specific query protocol, e.g., boolean queries [19, 20], shortest-distance queries [21, 22] and sub-graph queries [23]. None of the existing studies considered the privacy-preserving social search, despite the importance of it in the context of OSN.

1.2 Research Questions

The question for this research is:

> **How can we design a secure search system to protect the user's privacy while enabling the functionality of social search?**

In particular, enabling privacy-preserving social search via cryptographic techniques are investigated. This investigation includes designing a secure model and cryptographic primitives for secure social search as well as optimising the performance on real-world social search applications. Specifically, this thesis aims to make contributions to knowledge by addressing the following key questions:

- **Q1**: Social search queries in plaintext systems involve complex set-operations to retrieve data, and the retrieved contents need to be further processed. **How to design a proper model to enable rich types of social search queries in a privacy-preserving manner?**
- **Q2**: Existing systems adopt cryptographic primitives that can be time-consuming [24] or have a weak security guarantee [15]. **How to achieve strong security while preserving system efficiency in social search?**
- **Q3**: It is crucial to deploy the system over large-scale datasets in the real world and handle real-world problems. **How can the proposed system be applied to support social search and OSN services over large-scale encrypted OSN datasets?**

The above research questions comprise three requirements that need to be fulfilled in this research. First, the data confidentiality needs to be ensured even if the attacker gets full

access to the OSN server. Meanwhile, the social search functionality on the server should be functioning normally. Furthermore, social search queries need to be processed efficiently towards large datasets, which is essential for a practical solution in real-world OSN.

This thesis resorts to cryptographic solutions to satisfy the first two requirements since they can process and compute on the encrypted data without the key. However, some existing cryptographic solutions for securely data accessing and computing cannot offer a practical performance. For example, Oblivious RAM (ORAM) [25] and Fully Homomorphic Encryption (FHE) [26] can be easily adapted to provide a strong security guarantee to the social search functionality. Nonetheless, these schemes incur a prohibitive high computation and communication overhead, which makes them significantly slower than operating on the unencrypted data:

- ORAM [25] ensures that the data access will not reveal any pattern about data. But it consumes enormous memory space. For instance, if the social network dataset has N records, it takes $20N$ [27] to store it, as ORAM needs to pad many dummy blocks. Also, the bandwidth of loading/storing one record is logarithm in the size of dataset (i.e., $10 \log N$ [27]). It makes ORAM impractical to store the large-volume of OSN data.
- FHE enables arbitrary computations on the ciphertext, which allows any desirable social search functionality running on the encrypted input and outputting the encrypted result. The reason why FHE is not widely used is that the computation cost is prohibitively high. The recent FHE [26] connect the encrypted boolean gates to support more complex functions. The analysis shows it incurs a sub-cubic computation cost regarding the complexity of the target function and the length of the input. Specifically, for a given boolean circuit with depth L and input dimension n, FHE [26] takes $\mathcal{O}((nL)^{2.3})$ to evaluate one gate.

Hence, this thesis not only focuses on the system design facet to realise a secure social search system but also dives into cryptography to create customised cryptographic primitives for an efficient system.

1.3 Contributions

This thesis makes three major contributions. First, it presents a practical system (GraphSE^2 [28]) to support privacy-preserving social search queries over the encrypted data. In addition, this thesis introduces an application that can detect outliers based on the proposed system to further demonstrate the usability of our system in the real-world scenario. Finally, to enhance the security level of our system, it also proposes a cryptographic scheme to support secure conjunctive queries with a moderate extra cost. Note that this thesis focuses on the confidentiality of data and does not consider protecting the integrity of data.

The following sections outline the system, cryptographic primitives and application built.

1.3.1 GraphSE^2: Privacy-Preserving Social Search System

Chapter 3 introduces GraphSE^2, an encrypted graph database for online social network services to address massive data breaches: The system can encrypt the social graph and answer social search queries without decrypting it.

To address **Q1**, GraphSE^2 provides an encrypted and distributed structural data model to facilitate parallel and encrypted graph data access. Also, as observed in the plaintext social search system [29], the social search queries can be decomposed into several recognised essential operations, dubbed *atomic operations*. To achieve strong security and efficiency at the same time (ref. **R2**), the atomic operators are realised via the carefully chosen cryptographic primitives. In particular, the atomic operators are categorised into two types: The first is to perform set operations over the encrypted social graph. GraphSE^2 realises an encrypted graph model via the well-known Searchable Symmetric Encryption (SSE) scheme [30]. Specifically, it adopts an SSE scheme with boolean queries (OXT) [19], so as to support common complex set operations in social search; the second type of atomic operations enables GraphSE^2 to analyse user contents further. GraphSE^2 implements these operations via interchangeable secure computation protocols [31], which allows GraphSE^2 to execute different computations (i.e., arithmetic and comparison) securely and efficiently. In order to combine the computation with the secure graph index, Chapter 3 presents customised secure computation protocols that run under the two non-colluded server model [32]. The protocol utilises secret shares to represent the importance (score) of user-generated contents, and the score is colocated with the encrypted graph index. Doing so allows GraphSE^2 to further compute over the retrieved content; it also ensures that the underlying scores are hidden against servers from either of the two parties. The security of GraphSE^2 is formalised under the two semi-honest but non-colluded server model [32], and the corresponding security proof is given to show that GraphSE^2 offers demanded security guarantees to users.

GraphSE^2 has been implemented as a full-fledged prototype with various query operators supported in the Facebook graph search engine. The evaluation is performed with a real-world dataset collected from YouTube [33] (a social graph with 1,000,000 anonymised users). The result shows that the proposed distributed graph model and atomic operations can successfully address **Q3**: Most of the queries for an average user (130 friends) are processed within 1 s, and throughput is reduced at most 49% compared to the plaintext queries. Additionally, the system is high-scalable to support more users under the distributed model. This thesis further designs a real-world application based on GraphSE^2 to demonstrates its practicality in Chapter 4.

1.3.2 PPOD: Securely Detect Outlier over Incremental Data

Outlier detection is widely used in OSN services. For instance, it can be adopted in OSN to support services such as contextual filtering [34] and spam detection [35].

To illustrate the practicality of GraphSE^2 on real-world applications (ref. **Q3**), in Chapter 4, a Privacy-Preserving Outlier Detection (PPOD) protocol for incremental datasets is presented. PPOD uses the same two-server model as in GraphSE^2 and decomposes the outlier detection algorithm for incremental datasets into several phases. Each phase is then implemented via the atomic operators defined by GraphSE^2. To support efficient updates, PPOD integrates the sliding window model to periodically evict the expired data in order to maintain a constant update time. The PPOD protocol can track the outliers in an efficient and timeliness manner, and it only reveals the outlier information while preserving users' privacy regarding their sensitive data.

Chapter 4 provides systematic evaluations on the cryptographic modules and the overall protocols under various outlier detection parameters and sliding window settings. The results show that PPOD can efficiently handle incremental datasets: For a dataset with 16-dimensional data, PPOD can process outlier queries within 217 ms and updates the outlier model within 9 s after receiving a new data point.

1.3.3 HXT: Reducing Leakage for Conjunctive Queries

GraphSE^2 deploys the OXT protocol to encrypt the social graph and support set operations upon it. While the OXT protocol offers high performance by adopting a number of specialised data-structures, it also trades-off security by leaking 'partial' database information to the server. In particular, when querying a conjunctive query $w_1 \wedge \cdots \wedge w_n$, the OXT protocol reveals sub-intersections (i.e., $w_1 \wedge w_2$, $w_1 \wedge w_3$, ..., $w_1 \wedge w_n$) to the server. However, the file injection attack [36] have exploited the above partial information leakage to recover the query keywords with 100% accuracy.

Chapter 5 further studies **Q2** and proposes a new SSE protocol, called Hidden Cross-Tags (HXT), that removes the above leakage for conjunctive keyword search with a moderate overhead. HXT avoids this leakage by adopting two additional cryptographic primitives - Hidden Vector Encryption (HVE) and probabilistic (Bloom filter) indexing into the HXT protocol. By integrating the HVE scheme, the SSE scheme is still able to retrieve the result of a conjunctive query, but the partial information leakage (sub-intersections) is hidden from the server. Nevertheless, another issue about the existing HVE schemes [1, 37] is that they are typically built via the public-key setting. Hence, applying such schemes will result in a huge compromise in the efficiency of the SSE scheme, and it also makes the social search system impractical. Chapter 5 introduces a 'lightweight' HVE scheme that only uses efficient symmetric-key building blocks and entirely avoids public-key operations. At the same time, it affords selective simulation-security against an unbounded number of secret-key queries. Adopting this efficient HVE scheme, the overall practical storage and computational overheads of HXT over OXT are relatively small (no more than 10% for two keywords query, and 21% for six keywords query), while providing a higher level of security.

1.4 Thesis Structure

This thesis is organised as follows. Chapter 2 discusses the literature related to the global approach of this thesis. Note that the related work for each specific contribution is given in the corresponding chapter. Chapter 3 introduces the system model, atomic operation design, as well as the secure operators of GraphSE^2. It also presents a security proof and experimental evaluations to demonstrate the performance of GraphSE^2 in terms of its security and efficiency. Chapter 4 presents an advanced application (i.e., outlier detection) based on GraphSE^2 to show its practicality further. Chapter 5 reviews the security of the SSE scheme in GraphSE^2 and presents HXT to enhance the security of conjunctive queries in SSE schemes with a moderate extra cost. Chapter 6 further discusses some possible future directions related to this thesis and reports our progress on those research directions. Finally, Chapter 7 concludes the thesis.

Chapter 2

Related Work

This chapter summarises the related work to this thesis. Note that we only discuss the work related to the global background of this thesis. The related work of a specific contribution is presented in the corresponding chapter.

2.1 Privacy-Preserving Online Social Network

Numerous works have proposed to protect the sensitive data privacy in OSN, and two strategies are mainly applied in these works: **data decentralisation** and **cryptography**.

2.1.1 Decentralisation

Decentralisation OSN attempts to provide social services through a collection of independent nodes, and users can control their data in a flexible way.

Decentralised OSN in Peer-to-Peer Overlays. PeerSoN [38] and Safebook [39] build a decentralised social network in Peer-to-Peer (P2P) overlays. Users can store their data in their local device and on the devices of their trusted friends. These systems consist of two parts: a distributed hash table which is used provide lookup service to find users and the data they store; a P2P network which offers direct data exchange between users' devices. On these social networks, users are granted full privilege to control their data as well as the communication channel over P2P network: they can choose to either establish communication via their real-life trust [39] or the encrypted channel [38]. Cachet [40] further integrates a cache mechanism to reduce the access latency and maintain the availability of contents when the original content provider is offline.

Decentralised OSN in Outsourced Storage. Diaspora [41] and Confidant [42] create the decentralised social network by decoupling users' data with OSN. In those systems, users are asked for keeping their data in a trusted server. When the user wants to access SNS from an existing OSN, the user should generate some data descriptors and use them instead of their data with SNS. Others can read and retrieve the data from the trusted server via a valid descriptor and an appropriate access privilege.

2.1.2 Cryptography

Cryptography provides an extra layer of protection over user's data upon the decentralisation strategy, i.e., even if the attacker compromises a server in OSN, the confidentiality of sensitive data still can be guaranteed since the attacker does not have the key to recover it.

Table 2.1: Summary of the advantages/disadvantages of the existing privacy-preserving OSN schemes.

Scheme	Access Control	Self-host Data	Social Search
Decentralisation [38–42]	√	√	×
Cryptography [43–46, 48–50]	√	×	×

Private Social Relationship: Recent studies try to achieve a privacy-preserving OSN by separating social relationship from OSN service providers. NOYB [43] stores users' data with untrusted service providers, but the index (relationship) of these data is obfuscated by the cryptographic technique. In this system, only the authorised users can recover the real index and personal information associated with specific users, while unauthorised users only get a mixture of arbitrary personal data from the crowd. Lockr [44] separates social relationship from OSN service providers by providing Social Attestation mechanism and Social Access Control Lists. Furthermore, Social Attestation mechanism makes it easier for key revocation as it has an explicit expiry date. Lockr also applies proof of knowledge protocol to access third party services without revealing the actual identifier of users under the protocol. Some other scheme also proposed private social relationship based on privacy homomorphisms [45] or Blind Signature [46]. Private Social Relationship only provides the security guarantee to the index, but it does not work for the actual user data. In this case, the attacker can efficiently recover an astonishing amount of data if he or she has enough background knowledge [47].

Attribute-Based Encryption: An alternative approach is to use cryptographic primitives (i.e. Attribute-Based Encryption (ABE)) to enforce access control over a group of users in OSN [48]. Also, ABE is capable of associating the attributes of the user with a key, which provides a convenient way for the group manager to grant different access privileges to different groups. Users within those groups use the key to access the secret key of group manager as well as his/her sensitive data, so each group's access to the sensitive data is properly restricted, which guarantees the privacy of group manager. In Persona [48], key revocation is not efficient because it needs to re-issue a new key to all remaining users in the group. To address this issue, Sun et al. [49] and Jahid et al. [50] achieve efficient key revocation by using broadcast encryption.

Table 2.1 summarises the advantages and disadvantages of the above two strategies. As shown in Table 2.1, the existing solutions allow the user to have fine-grained access control on his or her data, and they can effectively prevent the data leakage from OSN providers [5]. Moreover, the decentralisation approach enables the user to host private data in their device, which gives them more flexibility when sharing the data to OSN. However, it is hard to enable social search in decentralised systems as the relations in a decentralised system are distributed in different networks. In this case, the quality of social search is impaired as each social search query needs to be processed by all independent nodes. After the results return to the client from multiple nodes, the client still undertakes heavy aggregation tasks

to get the final result. Furthermore, the additional cryptographic protection could make the problem harder to cope with, because different groups of users use a different key to communicate. In this context, OSN service providers should maintain a complex key management system and involve an expensive communication cost to combine the information from different groups.

2.2 Secure Search

Secure search has been extensively studied in the past years, and it can be categorised into two classes. The first is to design cryptographic protocols for different functionality (e.g., Boolean queries, range queries, etc.). The second category is to design secure search systems that work for real-world applications.

2.2.1 Cryptographic Protocols

Keyword Matching. Keyword matching is the fundamental function of a search system. The most well-studied secure keyword matching protocol is known as Searchable Symmetric Encryption (SSE). Most of the state-of-art SSE schemes provide a secure index to guarantee its query efficiency and have a sounded security definition.

The first SSE scheme is proposed by Song et al. [51]: their construction only supports single keyword queries and does not have an index to accelerate keyword search. Indexes are introduced to achieve a better search performance in the following researches. In particular, [52] builds a forward index-based SSE to achieve a linear search complexity. Curtmola et al. [30] propose to use an inverted index data structure, which further reduces the search complexity to a sub-linear time.

Advanced Search. The consequent work extends SSE to support more complex keyword matching queries, such as Boolean queries [19, 20] and fuzzy keyword queries [53]. Moreover, to support queries that involve comparison operations, i.e., range queries and ranked queries, the cryptographic protocols need to integrate specialised data structures with advanced primitives. For instance, some approaches [54, 55] leverage Order-Preserving Encryption to support the above queries. Meanwhile, the binary tree and homomorphic encryption [56] are also possible candidates to support the secure range queries and ranked queries. However, neither protocol leads to a generic search system that can support different types of searches, while it is crucial to the social search system.

Generic Cryptographic Tools. Some cryptographic primitives directly enable arbitrary operations over the encrypted data, and the social search system can be readily supported by those primitives. Hence, it is possible to design a privacy-preserving social search system via FHE [26, 57] or multi-party computation [58, 59]. Since the cost of using those cryptographic tools are prohibitively expensive, we do not consider to use them as our aim is to be a practical privacy-preserving social search system.

2.2.2 Secure Search Systems

Commercialised Database. Most of the existing commercialised databases [60–62] can encrypt the sensitive data before putting them into the database. Those solutions are integrated with the database product, which means the user can easily deploy them. Also, the DBMS has a dedicated query planner to transparently query the encrypted data, which assist the user to reduce the unnecessary exposure of sensitive data. Nonetheless, the above approaches do not support search over the encrypted data, i.e., the DBMS has to decrypt the data before querying them. As a result, the data is still leaked to the attacker if he or she controls the DBMS, which is a common case under the data outsourcing context.

Encrypted Database. The encrypted database is a promising solution to mitigate the data breach issue in an outsourced database. In particular, CryptDB [15] and Arx [63] provide an encrypted search protocol over the encrypted relational database. In addition, they implement a SQL interface to transform the unencrypted SQL statement to the encrypted queries for the encrypted database. There are also some existing systems designed for No-SQLs database, such as key-value store [18], column-based store [17] and graph database [64]. However, the above encrypted databases are designed for the functionality other than social search, and it is not trivial to support social search over these systems.

Hardware-Assisted Secure Search. Recently, some modern CPU is equipped with a hardware-enforced trusted execution environment, such as Intel SGX [65] and ARM TrustZone [66]. Thus, several schemes [67–69] are proposed to support secure search with the help of the trusted execution environment. We note that designing a hardware-assisted privacy-preserving social search system could be a complementary work to this thesis, and we reserve this as an interesting future direction.

Chapter 3

Enabling Privacy-Preserving Social Search with GraphSE^2

In this chapter, we propose GraphSE^2, an encrypted graph database for online social network services to address massive data breaches. GraphSE^2 preserves the functionality of *social search*, a key enabler for quality social network services, where social search queries are conducted on a large-scale social graph and meanwhile perform set and computational operations on user-generated contents. To enable efficient privacy-preserving social search, GraphSE^2 provides an encrypted structural data model to facilitate parallel and encrypted graph data access. It is also designed to decompose complex social search queries into atomic operations and realise them via interchangeable protocols in a fast and scalable manner. We build GraphSE^2 with various queries supported in the Facebook graph search engine and implement a full-fledged prototype. Extensive evaluations on Azure Cloud demonstrate that GraphSE^2 is practical for querying a social graph with a million users.

3.1 Introduction

Data breaches in online social networks (OSNs) affect billions of individuals and raise critical privacy concerns across the entire society [70, 71]. Besides, driven by the demands on huge storage and computation resources, OSN service providers utilise public commercial clouds as their back-end data storage [7–9], which further broadens the attack plane [10]. Therefore, there is an urgent call to improve the control of data confidentiality for cloud providers [72–74], in particular for current OSN services. The prevailing consensus to prevent data leakage is encryption. However, this approach impairs the functionality of *s*ocial search, a key enabler for quality OSN services [75]. Social search allows users to search the content of interests created by their friends. Compared with traditional web search, it produces personalised search results and serves for a wide range of OSN services such as friend discovering and user targeting.

The first task to enable privacy-preserving social search is how to scalably query over very large encrypted social graphs. On the one hand, a typical social graph can contain millions or even billions of users. On the other hand, users may generate a large volume of contents which will be queried for social search-related services [29]. The second and more challenging task is how to realise complex social search queries in an efficient and secure manner. As developed in plaintext systems (e.g., Facebook's Unicorn [29]), queries of social search contain set operations on graph-structured data, and the retrieved contents from the graph need to further be analysed (e.g., aggregation and sorting) for advanced services such as friendship-based recommendation.

In the literature, some work [24, 76] leverages generic building blocks (e.g., garbled circuits and oblivious data structures) to devise secure computational frameworks for graph algorithms. However, those frameworks do not appear to be scalable for low latency queries over large graphs. For example, a recent garbled circuit-based framework [24] takes several minutes to complete a sorting algorithm over a graph with only tens of thousands of nodes. Other work focuses on dedicated privacy-preserving graph algorithms, e.g., neighbour search [23, 77], and shortest distance queries [21, 22, 78, 79]. Unfortunately, the above algorithms are limited for or different from the functionality of social search queries.

Contributions. To bridge the gap, in this chapter, we propose and implement GraphSE^2, the first encrypted graph database that supports privacy-preserving social search. Unlike prior work which either suffers from low scalability or limited functionality, GraphSE^2 enables scalable queries over very large encrypted social graphs and preserves the rich functionality of the plaintext social search systems. Our contributions can be summarised as follows:

- We propose an encrypted and distributed graph model built on social graph modelling, searchable encryption, and the data partition technique. It facilitates queries over encrypted graph partitions in parallel while maintains the locality of graph data and user-generated contents for low query latency.
- We devise mixed yet interchangeable protocols to enable complex social search functions. The way of doing this is to decompose queries into atomic operations (i.e., set, arithmetic, and sorting operations) and then adapt suitable cryptographic primitives for efficient realisation. All these operations are tailored to be executed in parallel.
- We realise query operators of Facebook's social search system Unicorn [29], i.e., **term**, **and**, **or**, **difference**, and **apply**. We also design a query planner that can parse a query to atomic operations and initiate the corresponding primitives.
- We formally prove the security of our proposed query protocols under the real and ideal paradigm. Queries, graph data, and results are protected throughout the query process.
- We show the practicality of GraphSE^2 by implementing a prototype which is readily deployable. It leverages Spark [80] for setup (data partition and encryption), Redis [81] as the storage back-end, and uses Apache Thrift [82] to implement the query planner and query processing logic.

Our comprehensive evaluation on the Youtube dataset [33] with 1 million nodes confirms that all atomic operations are of practical performance. For set queries, GraphSE^2 retrieves a content list with 500 entities within 10 ms. For an average user (130 friends), GraphSE^2 takes at most 20 ms if the set operation involves two indexing terms (attributes); and it takes no more than 100 ms for five indexing terms. Regarding the computational operations, GraphSE^2 takes 100 ms to handle arithmetic computations over 10^4 entities, and 450 ms to sort 128 entities. As a summary, most of the queries for an average user are processed within 1 s, and throughput is reduced at most 49% compared to the plaintext queries.

Organisation. The rest of this chapter is structured as follows. We discuss related work in Section 3.2. In Section 3.3, we introduce the background knowledge about social search and the needed cryptographic primitives. After that, we describe the system overview in Section 3.4 and present the encrypted and distributed graph data model and the design of atomic operations in Section 3.5. In Section 3.6, we introduce the realisation of privacy-preserving social search queries and their security. Next, we describe our prototype implementation in Section 3.8 and evaluate the performance in Section 3.9. We give a conclusion in Section 3.10.

3.2 Related Work

Privacy-preserving graph query processing. There exist various designs that aim to answer a certain type of queries over the encrypted graph. Structured encryption [23] is proposed in the framework of SSE and supports adjacency and neighbouring queries. Some recent work is proposed to support privacy-preserving subgraph queries [83, 84]. However, all the above designs enable limited query functionality. Another line of work on privacy-preserving graph processing is to perform shortest-path queries over the encrypted graph. Protocols for this type of queries are devised via ORAM [78], structured encryption [21], or garbled circuit (GC) [22, 79]. To implement more complicated algorithms, protocols are proposed to use secret sharing and Homomorphic Encryption for Breadth-first search [76], PageRank [64], and approximate eigen-decomposition [85]. We stress that the above work targets on different query functionality other than social search. Note that a recent framework named GarphSC [24] can generate data-oblivious garbled circuits for graph algorithms such as PageRank and Matrix Factorisation. Because oblivious data structures are adapted for large graphs and all computations are realised via GC, it does not appear to achieve low latency for social search queries.

Encrypted database system. Our system is also related to encrypted database systems [15–18, 63]. CryptDB [15] is the first practical encrypted database system, which is built on Property-Preserving Encryption (PPE). It supports SQL queries over encrypted relational data records. BlindSeer [16] proposes a Bloom filter-based index and leverages GC to evaluate arbitrary boolean queries with keywords and ranges. Arx [63] follows the design of CryptDB to support SQL queries, but it uses SSE and GC to reduce the leakage from PPE. Seabed [17] uses Additively Symmetric Homomorphic Encryption to perform an efficient aggregation over the encrypted data and develops a schema with padding to mitigate the inference attack [86]. EncKV [18] adapts SSE and Order-Revealing Encryption schemes to design an encrypted and distributed key-value store. However, all the encrypted databases mentioned above are neither designed for graph data nor optimised for social search.

Graph processing system. In the plaintext domain, a large number of graph processing systems [29, 87, 88] (just to list a few) are proposed to support efficient large graph processing. However, all the above systems only support queries over the graphs in unencrypted

Table 3.1: Notations and terminologies.

Notation	Meaning
$\mathcal{SF}$	the service front-end
$\mathcal{ISC}_i$	the index server cluster deployed in P_i
$\mathcal{IS}$	the index server
id	the unique identifier of entity
e_{id}	the encrypted entity id
t	an indexing term in the form of $\mathsf{edge\text{-}type}{:}\mathsf{id}_\mathrm{u}$[1]
$\mathsf{stag}(t)$	the PRF value of the indexing term t
$h(\mathsf{id}, t)$	the cryptographic hash value computed from given identifier and indexing term
DB	an inverted indexed database $\{(t, \{(\text{sort-key}, \mathsf{id})\})\}$
$\mathsf{DB}(t)$	a list of $\{(\text{sort-key}, \mathsf{id})\}$ indexed by t
$\{\mathbf{E}\}$	the encrypted posting list with $(\langle\text{sort-key}\rangle^A, e_{\mathsf{id}})$ pairs
P_i	the i-th party in GraphSE^2 ($i \in \{0, 1\}$)
TP	the trusted party
x	a numerical value
$\mathbf{X}$	a matrix
$\langle x\rangle_i^*$	the Additive/Yao's share of a numerical value x in P_i
$\langle \mathbf{X}\rangle_i^*$	the Additive/Yao's share of a matrix $\mathbf{X}$ in P_i
$\mathcal{GC}$	a garbling scheme

1: We use t and w interchangeably.

form, which are unable to address privacy concerns of sensitive data leakage. Authenticated graph query [89] is proposed to verify the correctness of graph queries, which could be a complementary work to prevent attacks from malicious adversaries.

3.3 Background

We give a list of needed notations in our system construction and security analysis in Table 3.1. The detailed definitions of preliminaries we used are given in the following sections.

3.3.1 Social Graph Model

The social graph consists of nodes (aka entities) and edges (aka relationships of entities) in social networks. As the social graph is a sparse graph [29], it is normally represented via a set of adjacency lists. Like [29], we refer to these adjacency lists as *posting lists*.

Formally, the social graph is an edge-labeled and directed graph $G = (\mathrm{V}, \mathrm{E})$, where $\mathrm{V} = \{\mathrm{v}_1, \mathrm{v}_2, ...\}$ is the entity set and $\mathrm{E} = \{\mathrm{e}_1, \mathrm{e}_2, ...\}$ is the relationship set. Each posting list contains a list of entities $\{\mathbf{v}\}$, which are (sort-key, id) pairs. The sort-key is an integer that indicates the importance of the entity in a posting list, and the id is its unique identifier.

The posting lists are indexed by the inverted index, and modelled by the edges in social graph: All edges in G can be represented as a triad $\mathrm{e} = (\mathrm{u}, \mathrm{v}, \mathsf{edge\text{-}type})$ which consists

of its egress, ingress nodes ($\mathrm{u}, \mathrm{v} \in \mathrm{V}$) plus an edge-type which is a string representing the relationship between nodes (e.g., friend, like). The inverted indexing term t is in the form of edge-type:id_u. For example, the user may use friend:id_i to get the posting list of user i's friends.

3.3.2 Oblivious Cross-Tags (OXT) Protocol

Oblivious Cross-Tags (OXT) Protocol [19] is an SSE protocol, which proceeds between client C and server S. It provides an efficient way to perform conjunctive queries in the encrypted database[1]. Here we provide a high-level description as needed for the basic operations of our proposed system.

The protocol has two types of data structures. Firstly, for every keyword w, an inverted index, referred as '$\mathsf{TSet}(w)$', is built to point to the set $\mathsf{DB}(w)$ of all entity identifiers ids associating with w. Each $\mathsf{TSet}(w)$ is identified by an indexing term called $\mathsf{stag}(w)$, and all id values in $\mathsf{TSet}(w)$ are encrypted via a secret key κ_T. Both $\mathsf{stag}(w)$ and κ_T are computed as a PRF applied to w with C's secret keys. Another data structure called 'XSet' is built to hold a list of hash values $h(\mathsf{id}, w)$ (called 'xtag') over all entity identities id and keywords w contained in id, where h is a certain (public) cryptographic hash function. The above two data structures are stored on the server-side.

To search a conjunctive query $(w_1, w_2, ..., w_n)$ with n keywords, C sends the 'search token' $\mathsf{stag}(w_1)$ related to w_1 (called 's-term', we assume it to be w_1 in the above query) to S, which allows the server to run $\mathsf{TSet.Retrieve}(\mathsf{TSet}, \mathsf{stag}(w_1))$ and retrieve $\mathsf{TSet}(w_1)$ from the TSet. In addition, C sends 'intersection tokens' $\mathsf{xtoken}(w_1, w_i)$ (called 'xtraps') related to the $n-1$ keyword pairs (w_1, w_i) consisting of the 's-term' paired with each of the remaining query keywords w_i, $2 \leq i \leq n$ (called 'x-terms'). The xtraps allow the server to evaluate the cryptographic hash function of pairs (id, w_i) without knowing either keyword w_i or id. S checks the existence of $h(\mathsf{id}, w_i)$ in XSet and filters the $\mathsf{TSet}(w_1)$ to $n-1$ subsets of entities that contain the pairs (w_1, w_i). It only returns the entities that contain all $\{w_i\}_{1 \leq i \leq n}$ to the client. C finally uses K_w to recover the ids of entities.

As mentioned in [19], the security of OXT parameterised by a leakage function $\mathcal{L}_{\mathsf{OXT}} = (N, \psi, \mathsf{EP}, \mathsf{SP}, \mathsf{XP}, \mathsf{RP}, \mathsf{IP})$. It depicts what an adversary is allowed to learn about the database and queries via executing OXT protocol. Informally, considering a vector of queries $\mathbf{q} = (\mathsf{s} \wedge \psi(\mathsf{x}_2, ..., \mathsf{x}_n))$, which consists of a vector of s-terms s, a vector of boolean formulas ψ, and a sequence of x-term vectors $\mathsf{x}_2, ..., \mathsf{x}_n$. After executing q in a chosen database DB, the adversary only can learn:

- N: The total number of (id, w) pairs.
- ψ: The boolean formulae that the client wishes to query.
- EP: The repeat pattern in s.

[1]The scheme proposed in [20] supports disjunctive queries, but it consumes large storage space.

- SP: The size of posting lists for s.
- XP: The number of x-terms for each query.
- RP: The set of result id matching each pair of (s-term, x-term)-conjunction which is in the form $(\mathsf{s}, \mathsf{x}_i), 2 \leq i \leq n$.
- IP: The set of result id both existing in the posting lists of $\mathsf{s}[i]$ and $\mathsf{s}[j]$, which is only revealed when two queries $\mathsf{q}[i], \mathsf{q}[j], i \neq j$ have different s-terms but same x-terms.

3.3.3 Secure Computation

Additive Sharing and Multiplication Triplets. To additively share ($Shr^A(\cdot)$) an ℓ-bit value a, the first party P_0 generates $a_0 \in \mathbb{Z}_{2^l}$ uniformly at random and sends $a_1 = a - a_0 \mod 2^l$ to the second party P_1. The first party's share is denoted by $\langle a \rangle_0^A = a_0$ and the second party's is $\langle a \rangle_1^A = a_1$, the modulo operation is omitted in the description later. To reconstruct ($Rec^A(\cdot,\cdot)$) an additively shared value $\langle a \rangle^A$ in P_i, P_{1-i} sends $\langle a \rangle_i^A$ to P_i who computes $\langle a \rangle_0^A + \langle a \rangle_1^A$. Given two shared values $\langle a \rangle^A$ and $\langle b \rangle^A$, Addition ($Add^A(\cdot,\cdot)$) is easily performed non-interactively. In detail, P_i locally computes $\langle c \rangle_i^A = \langle a \rangle_i^A + \langle b \rangle_i^A$, which also can be denoted by $\langle c \rangle^A = \langle a \rangle^A + \langle b \rangle^A$. To multiply ($Mul^A(\cdot,\cdot)$) two shared values $\langle a \rangle^A$ and $\langle b \rangle^A$, we leverage Beaver's multiplication triplets technique [90]. Assuming that the two parties have already precomputed and shared $\langle x \rangle^A$, $\langle y \rangle^A$ and $\langle z \rangle^A$, where x, y are uniformly random values in $\mathbb{Z}_{2^l}$, and $z = xy \mod 2^l$. Then, P_i computes $\langle e \rangle_i^A = \langle a \rangle_i^A - \langle x \rangle_i^A$ and $\langle f \rangle_i^A = \langle b \rangle_i^A - \langle y \rangle_i^A$. Both parties run $Rec^A(\langle e \rangle_0^A, \langle e \rangle_1^A)$ and $Rec^A(\langle f \rangle_0^A, \langle f \rangle_1^A)$ to get e, f, and P_i lets $\langle c \rangle_i^A = i \cdot e \cdot f + f \cdot \langle x \rangle_i^A + e \cdot \langle y \rangle_i^A + \langle z \rangle_i^A$.

Garbled Circuit and Yao's Sharing. Yao's garbled circuit (GC) is first introduced in [91], and its security model has been formalised in [92]. Yao's GC is a generic tool to support secure two-party computation. The protocol is run between a "garbler" with a private input x and an "evaluator" with its private input y. The above two parties wish to securely evaluate a function $f(x, y)$. At the end of the protocol, both parties learn the value of $z = f(x, y)$ but no party learns more than what is revealed from this output value. In details, the garbler runs a garbling algorithm $\mathcal{GC}$ to generate a garbled circuit F and a decoding table dec for function f. The garbler also encodes its input x to $\hat{x}$ and sends it to the evaluator. The evaluator runs an oblivious transfer (OT) protocol [93] with the garbler to acquire its encoded input $\hat{y}$. Finally, the evaluator can compute $\hat{z}$ from $F, \hat{x}, \hat{y}$, decode it with dec, and share the result z with the garbler. The security proof against a semi-honest adversary under the two-party setting is given in [94].

In the following parts, we assume that P_0 is the garbler and P_1 is the evaluator. GC can be considered as a protocol which takes as inputs the Yao's shares and produces the Yao's shares of outputs. In particular, the Yao's shares of 1-bit value a is denoted as $\langle a \rangle_0^Y = \{K_0, K_1\}$ and $\langle a \rangle_1^Y = K_a$, where K_0, K_1 are the labels representing 0 and 1, respectively. The evaluator uses its shares to evaluate the circuit and gets the output shares (another labels).

Additive shares can be switched to Yao's shares efficiently. To be more precise, two par-

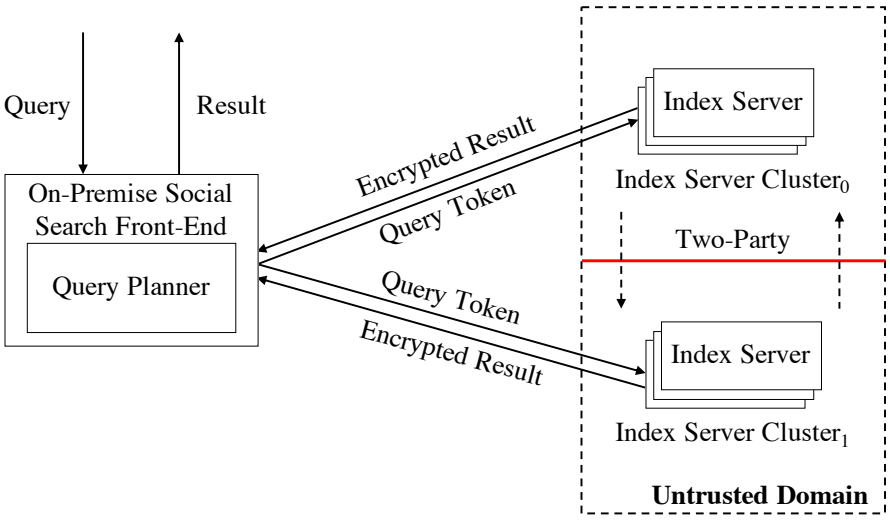


Figure 3.1: System architecture overview.

ties secretly share their additive shares $a_0 = \langle a \rangle_0^A$, $a_1 = \langle a \rangle_1^A$ in bitwise via Yao's sharing. The evaluator then receives $\langle a_0 \rangle^Y$ and $\langle a_1 \rangle^Y$ and evaluates the circuit $\langle a_0 \rangle^Y + \langle a_1 \rangle^Y \mod \langle 2^l \rangle^Y$ to get the label of a.

3.4 System Overview

3.4.1 System Architecture

As shown in Figure 3.1, GraphSE^2 has two entities: the on-premise social search service front-end ($\mathcal{SF}$) and the index server cluster ($\mathcal{ISC}$) with several index servers ($\mathcal{IS}$s) in an untrusted cloud. Note that this setting is consistent with many off-the-shelf social network service providers such as Airbnb [7] and Instagram [9], who use cloud data storage as the back-end to manage large graphs and massive user-generated data contents. Also, such architecture is now natively supported by public clouds, e.g., AWS Outposts [95]. GraphSE^2 aims to improve the protection of data confidentiality at the back-end, which is usually the high-value target for adversaries in practice.

During the setup phase, $\mathcal{SF}$ partitions the social graph to disjoint subgraphs and builds two instances of SSE indexes of each subgraph for the queries on structured information. The generated indexes are uploaded to two non-colluded $\mathcal{ISC}$s with multiple $\mathcal{IS}$s, respectively. The sort-keys are co-located with the corresponding indexes in the form of additive shares on the above two $\mathcal{ISC}$s for the arithmetic operations and sorting. Specifically, each $\mathcal{IS}$ has one of the two additive shares, and it pairs with a counter-party in the other cluster, which maintains the same index but holds the other share. Upon receiving a query from its users, $\mathcal{SF}$ uses a query planner to parse the query into atomic operations (see Section 3.5.3) to generate a query plan. It then sends the query tokens of atomic operations to all $\mathcal{IS}$s to execute the query plan. After that, each $\mathcal{IS}$ requests the structured information via the

Table 3.2: Supported social search operators in GraphSE^2 and its essential atom operations.

Query Operator	Example (from [29])	Atomic operations			
		Index Access	*Set Operations*	*Arithmetic*	*Sorting*
term	(**term** friend:1)	√			√
and	(**and** friend:1 friend:2)	√	√		√
or	(**or** friend:1 friend:2)	√	√		√
difference	(**difference** friend:3 (**and** friend:1 friend:2))	√	√		√
apply	(**apply** friend: friend:1)	√	√	√	√

tokens. Based on the matched encrypted contents, it executes arithmetic operations and scoring/ranking algorithms with its counter-party. Finally, the encrypted result is returned to $\mathcal{SF}$.

In this architecture, we consider a scenario of secure computation sourcing where the in-house $\mathcal{SF}$ assigns the computation to the $\mathcal{IS}$s in two untrusted but non-colluding clusters $\mathcal{ISC}_0$ and $\mathcal{ISC}_1$. Such a model of secure multi-party computation is formalised in [96] and applied in many existing studies [32, 97, 98]. Built on this model, GraphSE^2 offers two advantages: (i) $\mathcal{SF}$ is not required to be involved with any computation after it distributes the data to the servers, and (ii) the computation process can benefit from the mixture of multi-party computation protocols that enable efficient arithmetic operation, comparison, and sorting at the same time. Note that the communication between $\mathcal{IS}$s will not be the system bottleneck, because $\mathcal{IS}$s can be deployed in cloud clusters with dedicated datacenter networking support. This is consistent with prior studies based on the same architecture [32].

3.4.2 High-level Description

Before introducing the details of our system, we elaborate on the design overview and underlying design intuitions. To query large social graphs, GraphSE^2 develops an encrypted and distributed graph model. It is built on graph modelling, searchable encryption, and the standard data partition algorithm. Each server evenly stores an encrypted disjoint part of the whole graph. Meanwhile, this model is designed to co-locate the encrypted contents with the disjoint part containing the users who generate or relate to the contents. As a result, GraphSE^2 not only maximises the system scalability but also preserves data locality for low query latency.

To facilitate the realisation of various social search queries in the encrypted domain, GraphSE^2 first splits these complex queries into two stages, i.e., content search over the structured social graph and computational operations on the retrieved contents. Within the above stages, queries are further decomposed into atomic operations, i.e., *Index Access*, *Set Operations*, *Arithmetic* operations, and *Sorting*. Since the first stage commonly performs set operations over the social graph, GraphSE^2 realises our proposed graph model via a well-

known searchable encryption scheme for boolean queries (aka OXT [19]). The second stage requires a combination of different computations to further analyse user contents. For example, collaborative filtering [99] first obtains the scores of user contents via several addition and multiplication operations and then sorts the scores for an accurate recommendation.

To accelerate sophisticated computations in the second stage, GraphSE^2 mixes different secure computation protocols. Note that such philosophy also appears in recent privacy-preserving computation applications [31, 32]. Unlike prior work, GraphSE^2 customises the mixed protocols for social search queries and adapts them to our distributed graph model. In particular, GraphSE^2 represents the importance (score) of user-generated contents as the additive shares and deploys two distributed OXT instances at two non-colluded server clusters to store both the graph partitions and corresponding shares, respectively. Doing so allows GraphSE^2 to support parallel and batch addition and multiplication without the interaction between servers[2]. To achieve fast sorting, GraphSE^2 first converts additive shares to Yao's shares inside garbled circuits and then invokes a tailored distributed sorting protocol via GC. Each pair of servers in two clusters can perform local sorting in parallel, and then the intermediate results are aggregated for global sorting. Within the protocol, the underlying scores are hidden against servers from either of the two parties.

3.4.3 Threat Assumptions

In this work, we assume that $\mathcal{SF}$ is a private server dedicatedly maintained by the OSN service provider. It is a trustworthy party in the proposed model. Similar to the real-world OSN service provider (e.g. Airbnb), all users should submit their queries to $\mathcal{SF}$ through webpages or mobile apps. We assume that $\mathcal{SF}$ utilises the secure channel and cryptographic techniques to protect users' secrets. On the other hand, we assume that all $\mathcal{IS}$s are located in the untrusted domain. Meanwhile, we consider that the two clusters are semi-honest but not colluding parties. Each cluster performs social search faithfully but intends to learn additional information such as query terms, result ids and ranking values from the graph. Besides, those clusters hold user data and perform query functions, and thus they are high-value targets of adversaries. We assume that the two clusters can be compromised by two different passive adversaries, but the two adversaries will not collude. GraphSE^2 aims to protect the confidentiality of the private information in the social graph when the data storage back-end of the social search service is deployed at an untrusted domain.

3.4.4 Query Operators

GraphSE^2 follows a typical plaintext social search system [29] to define the operators (see Table 3.2).

[2]Multiplication involves a round of interaction between two servers, but they are in the same partition of two clusters.

In general, all operators in GraphSE^2 aim to retrieve posting lists from the encrypted graph index. The simplest form of these operators is **term**, which retrieves a single posting list via an *Index Access* operation. Like the other social search system, GraphSE^2 also supports **and** and **or** operators, which yield the intersection and union of posting lists via *Set Operations*, respectively. In addition, it supports the **difference** operator, which yields results from the first posting list that are not present in the others. Moreover, GraphSE^2 supports the unique query operator of Unicorn system [29], i.e., **apply**. The operator allows GraphSE^2 to perform multiple rounds of posting list retrieval to retrieve contents that are more than one edge away from the source node.

To enable quality search services (e.g., friendship-based recommendation), the retrieved posting lists should be scored/ranked before returning to users. As mentioned in Section 3.4.1, the additive shares of sort-keys are stored with its indexes. As a result, most of the query operators (e.g., **term**, **and**, **difference** and **or**) can use these shares to perform *Sorting* on the retrieved contents. Furthermore, it is often useful to return results in an order different from sorting by sort-keys. For instance, collaborative filtering [99] evaluates an arithmetic formula about friendships and ratings on items to produce the personalised scores for recommended items. The new score is a better prediction than the sort-keys, as the later only reflects the overall preference in the community (e.g., the hit-count on the item). The defined operators natively support arithmetic computations via the additive shares affixed with indexes. Specifically, the **apply** operator has the capability to support the secure evaluation on complicated scoring formulas with *Arithmetic* operations: It can access different types of entities (e.g., user's friends, items liked by users, etc.) in a multiple round-trip query, which means it can combine the scores of different entities and cache the intermediate result for next round computations.

3.5 The Proposed System

3.5.1 Encrypted Graph Data Model

To support social search operations in [29] on an encrypted social graph (see Section 3.3.1 for details), GraphSE^2 creates the OXT index (i.e., TSet and XSet, see Section 3.3.2 for details) for encrypted graph structure access in $\mathcal{ISC}$s, and the additive shares are integrated with the corresponding index to support complex computations. Specifically, to support simple graph structure data access, each posting list is encrypted and stored as a TSet tuple in the $\mathcal{ISC}$: $(\mathsf{stag}(\mathsf{edge\text{-}type}{:}\mathsf{id}_u), \{\mathbf{E}\})$. The TSet tuple consists of the stag of indexing term as the key and the encrypted posting list $\{\mathbf{E}\}$ as the value. Each element in $\{\mathbf{E}\}$ is an encrypted tuple $\mathbf{E}_{\mathsf{id}} = (\langle \text{sort-key} \rangle^A, e_{\mathsf{id}})$, which keeps the encryption e_{id} of entity id. Additionally, the sort-key of entity is shared as additive sharing value. GraphSE^2 associates it with the encrypted entity id to support complex computations. Moreover, GraphSE^2 evaluates the cryptographic hash function of $(\mathsf{id}, \mathsf{edge\text{-}type}{:}\mathsf{id}_u)$ pairs to generate an XSet for complex

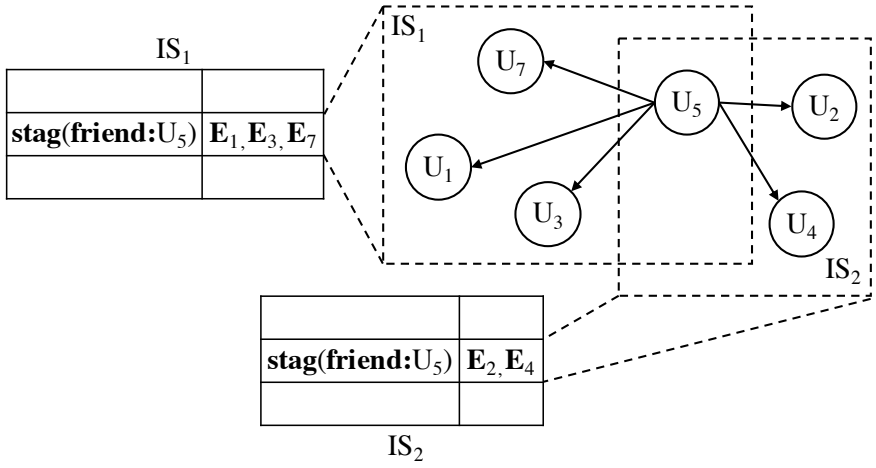


Figure 3.2: Our encrypted and distributed data model, the arrows indicate the friend relationships between users.

set operations.

3.5.2 Encrypted and Distributed Graph Index

In order to support the system to process the query in parallel, GraphSE^2 distributes the encrypted graph across multiple index servers for each cluster. GraphSE^2 devises a partition strategy that shards the posting lists by hashing on result id. Figure 3.2 gives an example of the proposed partition strategy in an $\mathcal{ISC}$ with two $\mathcal{IS}$s. We employ a **modulo** partition strategy, which split the original posting list into multiple non-duplicate parts, but other graph partition strategies (e.g., [87]) can also be applied to shard the social graph. The design has three advantages in the context of the distributed environment. First, it maintains the availability in the event of server failure. Furthermore, the sharding strategy enables the distributed system to finish most of the set operations and the consequent scoring, ranking and truncating in $\mathcal{IS}$s. It splits the computation loads into distributed servers to improve the efficiency and also cuts down the communication cost between $\mathcal{IS}$s and $\mathcal{SF}$. Finally, it does not affect the security of GraphSE^2 because the adversary who compromises an $\mathcal{ISC}$ gets the same view (the whole encrypted database) as the adversary in a single OXT instance. If the adversary cannot access all $\mathcal{IS}$s in the $\mathcal{ISC}$, only the view on a fraction of the encrypted database is learned.

3.5.3 Atomic Operations

As mentioned in Section 3.4.4, the social search queries are implemented by a set of operators. We observe that these operators can be decomposed to a set of atomic operations. We now describe the implementation of these atomic operations in the encrypted domain. For each atomic operation, we explain how we adapt and optimise it in the proposed system.

Algorithm 1 Index Access

Input: TSet, Indexing Term t
Output: Encrypted Result $\mathsf{TSet}(t)$

1: **function** INDEXACCESS(TSet, t)
2: $\mathcal{SF}$ inputs indexing term t, and $\mathcal{IS}$ inputs TSet;
3: $\mathcal{SF}$ computes $\tau \leftarrow \mathsf{stag}(t)$;
4: $\mathcal{SF}$ sends τ to $\mathcal{IS}$;
5: $\mathcal{IS}$ computes $\mathsf{TSet}(t) \leftarrow \mathrm{TSet.Retrieve}(\mathsf{TSet}, \tau)$;
6: **return** $\mathsf{TSet}(t)$;
7: **end function**

Index Access

We start with *Index Access* operation, which is used to retrieve the neighbouring nodes of the target user with the given edge-type (e.g., friend, likes) from the social graph. Algorithm 1 outlines the searching procedure using TSet operations. On receiving the search keyword, $\mathcal{SF}$ firstly generates a search token τ, which is $\mathsf{stag}(t)$ of the indexing term t. $\mathcal{IS}$ can use τ to search TSet and get the encrypted posting list $\mathsf{TSet}(t)$ as the return. *Index Access* operation can be easily extended to run in parallel. More specifically, $\mathcal{SF}$ broadcasts search token τ to all $\mathcal{IS}$s. After that, each $\mathcal{IS}$ uses τ to get its local partition of the whole encrypted posting list and sends it back.

Security. The security of *Index Access* is guaranteed by the security property of TSet. Informally, *Index Access* is $\mathcal{L}_{\mathbf{T}}$-semantically-secure against adaptive attacks where $\mathcal{L}_{\mathbf{T}}$ is the leakage function of TSet. $\mathcal{L}_{\mathbf{T}}$ is well-defined and discussed in [19]. It ensures *Index Access* only leaks the number of edges in the encrypted social graph.

Set Operations

This operation involves the boolean expression with multiple indexing terms. GraphSE^2 uses it to query the encrypted graph-structured data and finds the neighbouring nodes and the corresponding user-generated content that satisfy the given boolean expression. In GraphSE^2, we adapt the OXT protocol to support this atomic operation, but some of the other SSE protocols supporting conjunctive queries (e.g. [100]) can also be readily adapted as the building block of GraphSE^2. The OXT protocol supports conjunctive queries of the form $t_1 \wedge t_2 \wedge ... \wedge t_n$ natively, but it can be extended to support the boolean query of the form $t_1 \wedge \psi(t_2, ..., t_n)$, where t_1 is the 's-term', and $\psi(t_2, ..., t_n)$ is an arbitrary boolean expression [19]. As shown in Algorithm 2, the extended protocol follows the basic steps to obtain search tokens and search in TSet and XSet interactively. Nevertheless, it introduces additional steps (line 3, 13, 15–17 in Algorithm 2) to solve the boolean expression $\psi(t_2, ..., t_n)$. Specifically, $\mathcal{SF}$ substitutes all indexing terms t_i to boolean variables v_i ($i = 2, ..., n$) and generates a boolean function $\hat{\psi}(v_2, ..., v_n)$. $\mathcal{SF}$ then sends $\hat{\psi}(v_2, ..., v_n)$ to $\mathcal{IS}$. $\mathcal{IS}$ sets the value of v_i to the truth values of $h(\mathsf{id}_c, t_i) \in \mathsf{XSet}$. Then, it evaluates $\hat{\psi}(v_2, ..., v_n)$ and returns $\mathbf{E}_c$ as a result if $\hat{\psi}$ outputs true.

The BOOLEANQUERY algorithm can be utilised to enable set operations of social search

Algorithm 2 Boolean Query

Input: TSet, XSet, Query $(t_1 \wedge \psi(t_2, ..., t_n))$ with s-term t_1
Output: Encrypted Result R

function BOOLEANQUERY(TSet, XSet, $\bar{t}, \psi$)($\bar{t}$ is the indexing term list $(t_1, ..., t_n)$, and ψ is an arbitrary boolean expression)
  $\mathcal{SF}$ inputs indexing term $\bar{t}, \psi$, and $\mathcal{IS}$ inputs TSet, XSet;
  $\mathcal{SF}$ initialise a boolean expression $\hat{\psi}(v_2, ..., v_n)$ from ψ and sends it to $\mathcal{IS}$;
  $\mathcal{SF}, \mathcal{IS}$ runs $\mathsf{TSet}(t_1) \leftarrow$ INDEXACCESS(TSet, t_1);
  $\mathcal{IS}$ parses $\mathsf{TSet}(t_1)$ to $\{\mathbf{E}\}$;
  for $l = 2 : n$ **do**
    $\mathcal{SF}$ computes $\mathsf{xtoken}(t_1, t_l)$;
    $\mathcal{SF}$ sends $\mathsf{xtoken}(t_1, t_l)$ to $\mathcal{IS}$;
  end for
  $\mathcal{IS}$ initialises $R \leftarrow \{\}$;
  for $c = 1 : |\{\mathbf{E}\}|$ **do**
    for $l = 2 : n$ **do**
      $\mathcal{IS}$ uses $\mathsf{xtoken}(t_1, t_l)$ to compute $h(\mathsf{id}_c, t_l)$;
      $\mathcal{IS}$ lets $v_l = (h(\mathsf{id}_c, t_l) \in \mathsf{XSet})$;
    end for
    if $\hat{\psi}(v_2, ..., v_n) = `True'$ **then**
      $\mathcal{IS}$ adds $\mathbf{E}_c$ in R;
    end if
  end for
  return R;
end function

queries as shown in Table 3.2, which will be discussed in the following section.

Security. In cryptographic terms, the OXT protocol is proved to be $\mathcal{L}_{\mathsf{OXT}}$-semantically-secure against adaptive attacks, where $\mathcal{L}_{\mathsf{OXT}}$ is the leakage function defined in [19]. It ensures that the untrusted server only learns the information defined in the leakage function, but no other information about the query and underlying dataset. We refer the reader to Section 3.3.2 for more details.

Arithmetic

GraphSE^2 uses *Arithmetic* operations to support complex scoring functions over the retrieved content from *Set Operations*. *Arithmetic* operations in GraphSE^2 involve the secure two-party computation between two $\mathcal{ISC}$s. Here, we introduce the simplest model of GraphSE^2, where each $\mathcal{ISC}$ only has one IS, for ease of presentation on how to use additive shares (see Section 3.3.3 for detailed definition) to compute addition and multiplication under the two-party setting. Note that this model can be extended to support multiple pairs of $\mathcal{IS}$s.

In GraphSE^2, the posting list is generalised as a matrix, and the arithmetic operations are evaluated over the matrix. The reason for that is, instead of running the scoring function with arithmetic operations multiple times for each item of the posting list, the batch processing can reduce the system overhead and support scoring algorithms in parallel. We denote the matrix of sort-keys returned from a structured information query by $\mathbf{S}$, and the

corresponding shared matrix is denoted by $\langle \mathbf{S} \rangle^A$. Given two shared matrices $\langle \mathbf{A} \rangle^A$ and $\langle \mathbf{B} \rangle^A$, the addition operation ($Add^A(\langle \mathbf{A} \rangle^A, \langle \mathbf{B} \rangle^A)$) can be evaluated non-interactively by computing $\langle \mathbf{C} \rangle^A = \langle \mathbf{A} \rangle^A + \langle \mathbf{B} \rangle^A$ in each party. To multiply two shared matrices ($Mul^A(\langle \mathbf{A} \rangle^A, \langle \mathbf{B} \rangle^A)$), two $\mathcal{IS}$s generate the multiplication triplets, which are shared matrices: $\langle \mathbf{X} \rangle^A, \langle \mathbf{Y} \rangle^A, \langle \mathbf{Z} \rangle^A$. $\mathbf{X}$ has the same dimension as $\mathbf{A}$, $\mathbf{Y}$ has the same dimension as $\mathbf{B}$, and $\mathbf{Z} = \mathbf{X} \times \mathbf{Y} \mod 2^l$. $\mathcal{IS}_i$ computes $\langle \mathbf{E} \rangle_i^A = \langle \mathbf{A} \rangle_i^A - \langle \mathbf{X} \rangle_i^A$ and $\langle \mathbf{F} \rangle_i^A = \langle \mathbf{B} \rangle_i^A - \langle \mathbf{Y} \rangle_i^A$, and sends it to its counter-party. Both parties then recover $\mathbf{E}, \mathbf{F}$ and let $\langle \mathbf{C} \rangle_i^A = i \cdot \mathbf{E} \times \mathbf{F} + \langle \mathbf{X} \rangle_i^A \times \mathbf{F} + \mathbf{E} \times \langle \mathbf{Y} \rangle_i^A + \langle \mathbf{Z} \rangle_i^A$.

The multiplication operation relies on the triplets, which should be generated before the actual computation. In addition, each party keeps their $\langle \mathbf{X} \rangle^A, \langle \mathbf{Y} \rangle^A$ in secret during the generation process; otherwise, they can recover $\mathbf{A}, \mathbf{B}$ after two parties exchanged $\langle \mathbf{E} \rangle^A, \langle \mathbf{F} \rangle^A$. Thus, GraphSE^2 introduces a secure offline protocol [32] to generate the triplets via OT, it utilises the following relationship: $\mathbf{Z} = \langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_0^A + \langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_1^A + \langle \mathbf{X} \rangle_1^A \times \langle \mathbf{Y} \rangle_0^A + \langle \mathbf{X} \rangle_1^A \times \langle \mathbf{Y} \rangle_1^A$ to compute the shares of $\mathbf{Z}$. The resulting offline protocol is only required to compute the shares of $\langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_1^A$ and $\langle \mathbf{X} \rangle_1^A \times \langle \mathbf{Y} \rangle_0^A$ as the other two terms can be computed locally.

We illustrate the computing process of $\langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_1^A$ in the offline protocol. The basic step of the offline protocol is to use $\langle \mathbf{X} \rangle_0^A$ and a column from $\langle \mathbf{Y} \rangle_1^A$ to compute the share of their product. This is repeated for each column in $\langle \mathbf{Y} \rangle^A$ to generate $\langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_1^A$. Therefore, for simplicity, we focus on the above basic step: We assume that the size of $\langle \mathbf{X} \rangle^A$ is $\mathrm{s} * \mathrm{t}$ and we denote each element in $\langle \mathbf{X} \rangle^A$ as $\langle x_{i,j} \rangle_0^A$, $i = 1, ..., \mathrm{s}$ and $j = 1, ..., \mathrm{t}$. In addition, we assume each column of $\langle \mathbf{Y} \rangle^A$ has t elements, which are denoted as $\langle y_j \rangle_1^A$, $j = 1, ..., \mathrm{t}$. The computation process is listed as follows:

- $\mathcal{IS}_0$ runs a correlated-OT (COT) protocol [93] and sets the correlation function to $f_b(x) = \langle x_{i,j} \rangle_0^A \cdot 2^b + x \mod 2^l$ for $b = 1, ..., l$.
- For each bit b of $\langle y_j \rangle_1^A$, $\mathcal{IS}_0$ chooses a random value r_b for each bit and runs $\mathsf{COT}(r_b, f_b(r_b); \langle y_j \rangle_1^A[b])$ with $\mathcal{IS}_1$.
- If $\langle y_j \rangle_1^A[b] = 0$, $\mathcal{IS}_1$ gets $r_b \mod 2^l$; If $\langle y_j \rangle_1^A[b] = 1$, $\mathcal{IS}_1$ gets $f_b(r_b) = \langle x_{i,j} \rangle_0^A \cdot 2^b + r_b \mod 2^l$. It is equivalent to get $\langle y_j \rangle_1^A[b] \cdot \langle x_{i,j} \rangle_0^A \cdot 2^b + r_b \mod 2^l$ in $\mathcal{IS}_1$ side.
- $\mathcal{IS}_1$ sets $\langle \langle x_{i,j} \rangle_0^A \cdot \langle y_j \rangle_1^A \rangle_1^A = \Sigma_{b=1}^l \langle y_j \rangle_1^A[b] \cdot \langle x_{i,j} \rangle_0^A \cdot 2^b + r_b \mod 2^l$, and $\mathcal{IS}_0$ sets $\langle \langle x_{i,j} \rangle_0^A \cdot \langle y_j \rangle_1^A \rangle_0^A = \Sigma_{b=1}^l (-r_b) \mod 2^l$.

After computing $\langle \langle x_{i,j} \rangle_0^A \cdot \langle y_j \rangle_1^A \rangle^A$, the j-th element of the i-th row in $\langle \langle \mathbf{X} \rangle_0^A \times \langle \mathbf{Y} \rangle_1^A \rangle^A$ is $\Sigma_{j=1}^{\mathrm{t}} \langle \langle x_{i,j} \rangle_0^A \cdot \langle y_j \rangle_1^A \rangle^A$. Analogously, $\mathcal{IS}_0$ and $\mathcal{IS}_1$ can compute the share of $\langle \mathbf{X} \rangle_1^A \times \langle \mathbf{Y} \rangle_0^A$ in the same way.

Security. Additive sharing scheme offers security guarantees to *Arithmetic* operations in GraphSE^2 via its computational indistinguishable property. More specific, as discussed in [101], the scheme can create a uniformly distributed input and output to protects the original input/output of *Arithmetic* operation under the threat model of GraphSE^2, i.e., semi-honest but non-colluding two-party.

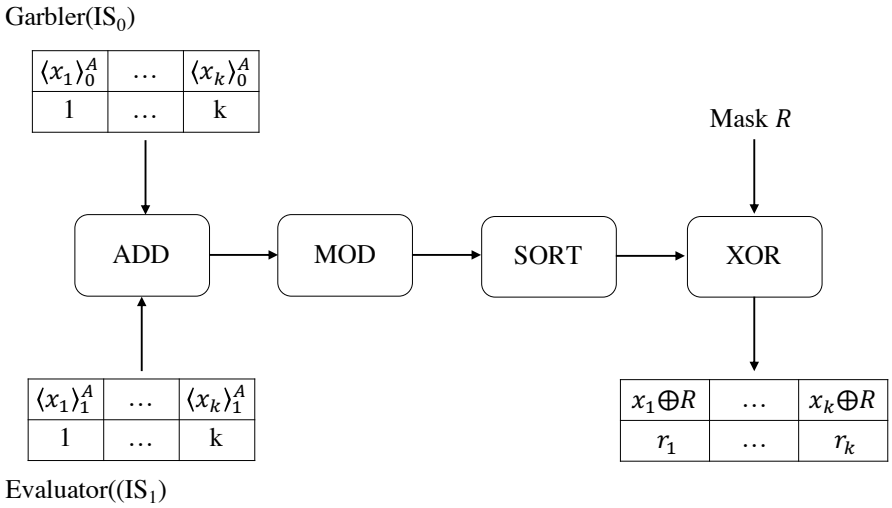


Figure 3.3: Local sorting process for one pair of garbler and evaluator. The input of the garbler is at the top, and the input/output of the evaluator is on the bottom.

Sorting

This is a required operation in order to rank the computed scores from *Arithmetic* operations. A naive solution is to recover all scores from additive shares in $\mathcal{SF}$ and sort them as plaintext. However, transmitting all rank results to $\mathcal{SF}$ is a bandwidth-consuming task; the sort operation can be very inefficient as the result. Therefore, GraphSE^2 chooses to mix the additive sharing scheme and Yao's garbled circuit (see Section 3.3.3 for details) to support arithmetic operations and comparison at the same time, as it avoids the communication overhead from sending the shares back to $\mathcal{SF}$. To protect the privacy of score values, the generated circuit should have a fixed sequence of comparison for a given size of inputs (i.e., achieving the trace-oblivious), and it should not reveal the actual scoring value after circuit evaluation.

Local Sorting. To enable sorting on $\mathcal{IS}$s, GraphSE^2 leverages an efficient scheme in [31] to switch from additive sharing to Yao's sharing. It then adopts the sorting network [102] to generate the optimised sorting circuit. Finally, the garbler concatenates the sorting network with an XOR gate and applies a random mask R to mask the score values. As a result, the evaluator can use decode table dec to figure out the rank, but it does not know the score values. Thus, the local sorting algorithm in GraphSE^2 can be divided into five phases. Figure 3.3 illustrates the process of local sorting.

Given $\mathcal{IS}_0$ as the garbler and $\mathcal{IS}_1$ as the evaluator, both parties pre-share a scoring vector $\mathsf{x} = \{\langle x_i\rangle^A\}_{i=1}^{\mathrm{k}}$, GraphSE^2 runs the protocol LocalSort(x) to sort the vector and returns a sorted vector in descending order of x, the protocol can be summarised as follows:

- Phase 1: $\mathcal{IS}_0$ runs $\mathcal{GC}$ to generate the circuit in Figure 3.3 as well as its decode table dec. It then sends the circuit and the decode table dec to $\mathcal{IS}_1$. Doing so ensures that $\mathcal{IS}_1$ only can see the final result with random mask R.

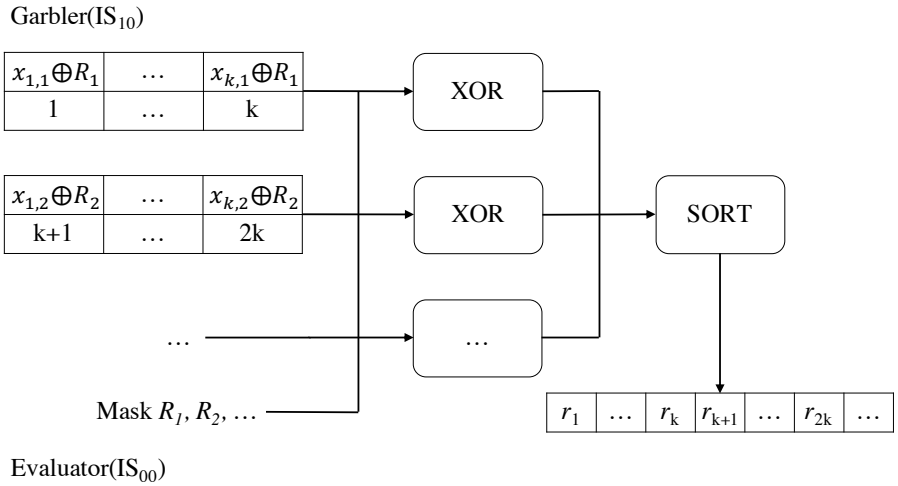


Figure 3.4: Global sorting process in the coordinators. The input of the garbler is the masked score vector with a payload that indicates the position of the score in vector, and the input of the evaluator is the random masks.

- Phase 2: $\mathcal{IS}_0$ sends the encoded inputs of its additive shares $\{\langle x_i \rangle_0^A\}_{i=1}^{\mathrm{k}}$ with a payload vector $\{i\}_{i=1}^{\mathrm{k}}$ indicating the position. This prevents $\mathcal{IS}_1$ from learning the additive shares of $\mathcal{IS}_0$.
- Phase 3: $\mathcal{IS}_1$ retrieves the encoded inputs of its additive shares $\{\langle x_i \rangle_1^A\}_{i=1}^{\mathrm{k}}$ and payload vector from $\mathcal{IS}_0$ via OT. This prevents $\mathcal{IS}_0$ from learning the additive shares of $\mathcal{IS}_1$.
- Phase 4: $\mathcal{IS}_0$ generates the encoded input of a random mask R to perform the last XOR gate to protect the vector.
- Phase 5: $\mathcal{IS}_1$ uses the given inputs to evaluate the circuit and uses dec to decode the outputs.

Since the circuit puts a mask R after sorting, $\mathcal{IS}_1$ only gets the ranking $\{r_i\}_{i=1}^{k}$ without knowing the actual scores.

Global Sorting. The above sorting strategy is a suitable and efficient solution for the simplest model, i.e., only one $\mathcal{IS}$ in each $\mathcal{ISC}$. However, it can be problematic when each $\mathcal{ISC}$ has several $\mathcal{IS}$s. In this case, no $\mathcal{IS}$ can provide a full sorted list as each $\mathcal{IS}$ only has a disjoint part of the whole graph. Hence, $\mathcal{SF}$ still needs to perform another inefficient plaintext sorting.

Therefore, GraphSE^2 uses a specific protocol which runs by a chosen coordinator of each $\mathcal{ISC}$. The protocol can perform an extra round of sorting upon the results from local sort while keeping the scoring value in secret. Assuming that each $\mathcal{ISC}$ has n different $\mathcal{IS}$s, $\mathcal{IS}_{00}$ and $\mathcal{IS}_{10}$ are chosen to be the coordinators for $\mathcal{ISC}_0$ and $\mathcal{ISC}_1$, respectively. After local sorting, evaluator in $\mathcal{ISC}_1$ sends a vector of masked scoring values $\{x_{i,j} \oplus R_j\}$ where $i = 1, ..., \mathrm{k}$ and $j = 1, ..., \mathrm{n}$ to $\mathcal{IS}_{10}$ and garbler in $\mathcal{ISC}_0$ sends the masks $\{R_j\}_{j=1}^{\mathrm{n}}$ to $\mathcal{IS}_{00}$. In the global sorting, GraphSE^2 switches the roles of $\mathcal{IS}_{10}$ and $\mathcal{IS}_{00}$ (i.e., $\mathcal{IS}_{10}$ is the garbler, and $\mathcal{IS}_{00}$ is the evaluator) for two reasons: It prevents $\mathcal{IS}_{10}$ from evaluating two circuits at

the same time, as $\mathcal{IS}_{10}$ also needs to evaluate another local sorting circuit for its partition. Furthermore, it facilitates pipeline data processing. The partial result of each $\mathcal{IS}$ can be sent to $\mathcal{IS}_{10}$ and $\mathcal{IS}_{00}$ for global sorting separately. Once the $\mathcal{IS}_{10}$ and $\mathcal{IS}_{00}$ get the first result, they can start to run encoding and OT. The protocol GLOBALSORT(x) is summarised as follows:

- Phase 1: $\mathcal{IS}_{10}$ runs $\mathcal{GC}$ to generate the circuit in Figure 3.4 and the decode table dec. It then sends the circuit and dec to $\mathcal{IS}_{00}$.
- Phase 2: $\mathcal{IS}_{10}$ sends the encoded inputs $\{x_{i,j} \oplus R_j\}$ where $i = 1, ..., \mathrm{k}$ and $j = 1, ..., \mathrm{n}$ with a payload vector $\{i\}_{i=1}^{\mathrm{n}\cdot\mathrm{k}}$.
- Phase 3: $\mathcal{IS}_{00}$ retrieves encoded masks $\{R_j\}_{j=1}^{\mathrm{n}}$ via OT.
- Phase 4: $\mathcal{IS}_{00}$ uses the given inputs to evaluate the circuit and uses dec to decode the outputs. The result is sent to $\mathcal{SF}$ in descending order.

Security. The security properties of garbled circuit [92] and OT [93] ensure the security of *Sorting* in GraphSE^2 in the following three aspects: Firstly, no adversary can learn the input of its counter-party when sorting (i.e., the other additive share in LOCALSORT(x) and the masked score/mask in GLOBALSORT(x)). Secondly, the output of LOCALSORT(x) is masked by a one-time mask, which is a uniformly random number. It protects the original score vector because the evaluator only learns the masked values from the output. Finally, for the output of GLOBALSORT(x), only the decode table of the rank is sent to the evaluator, which also ensures that the evaluator only learns the global rank without knowing the actual ranking score.

3.6 Query Realisation

In GraphSE^2, $\mathcal{SF}$ receives queries as the query strings in the form of s-expression. It is composed of several operators to describe the set of results the client wishes to receive (see Table 3.2). In the following sections, we introduce the operators in GraphSE^2 and their security properties.

3.6.1 Graph Operators

GraphSE^2 uses atomic operations in Section 3.5.3 to realise all operators in Table 3.2. In this section, we present the detailed constructions of these operators. Note that we only consider the operators as the outermost operators, i.e., they are not nested in any other query strings, because the query plan generation highly depends on the outermost operators.

term. The **term** operator runs an *Index Access* operation to retrieve a posting list. In addition, if there is a requirement to rank the result, the *Sorting* operation is able to return a sorted posting list which puts the record with higher relevance at the beginning of the list.

and. This operator is natively supported by the BOOLEANQUERY algorithm. As mentioned in Section 3.5.3, conjunctive queries with nested queries (e.g., $t_1 \wedge \psi(t_2, ..., t_n)$) are processed by evaluating the boolean expression ψ in XSet. It is obvious that the **and** operator is executed in a sub-linear time, as its complexity is proportional to the size of $\mathsf{TSet}(t_1)$.

difference. The **difference** operator is extended from BOOLEANQUERY algorithm. Considering the query (**difference** friend:3 (**and** friend:1 friend:2)) from Table 3.2, it aims to find the friends of **user**$_3$, who are neither **user**$_1$'s friends nor **user**$_2$'s friends. The boolean expression, in this case, is friend:1 $\wedge$ friend:2, but the results that satisfy the expression are removed from the results of the query (**term** friend:3). In summary, the **difference** operator excludes the results that satisfy the boolean expression ψ. Therefore, the s-expression with **difference** operator is represented as $t_1 \wedge \neg\psi(t_2, ..., t_n)$. Comparing with the **and** operator, it returns the results only if the boolean expression ψ returns false instead of true.

or. The complexity of the original approach for processing disjunctive queries is linear to the size of the database [19]. To achieve a sub-linear time complexity, we leverage the above **difference** and **term** operators to build a new disjunctive query operator. In particular, the s-expression starts with the **or** operator can be processed via a list of **difference** expressions and an additional **term** expression. For instance, if a disjunctive query has three indexing terms: t_1, t_2, t_3, the corresponding s-expression is (**or** t_1 t_2 t_3), and it is parsed as (**difference** t_1 (**or** t_2 t_3)), (**difference** t_2 t_3), (**term** t_3). The above three s-expressions return three different sets of results, and the composite of them is the final result of the **or** operator. The correctness of the above approach can be easily proved by the set operation: $t_1 \vee t_2 \vee t_3 = (t_1 n (t_2 \vee t_3)) \vee (t_2 n t_3) \vee t_3$.

In general, a disjunctive s-expression with n indexing terms can be rewritten as $n-1$ s-expressions with the **difference** operator and 1 s-expressions with the **term** operator. The complexity is proportional to $|t| \cdot M$, where $|t|$ is the number of disjunctive indexing terms and M is the result size of the most frequent term $\max(\{|\mathsf{TSet}(t_i)|\}_{i=1}^{n})$.

3.6.2 Apply Operator

The **apply** operator is a unique operator in Unicorn [29], which enables graph-traversal. The basic idea is to retrieve the results of nested queries and use these results to construct and execute a new query. For example, given an s-expression (**apply** friend: friend:id_1), $\mathcal{SF}$ issues a query (**term** friend:id_1) and collects N users, it then generates the second query (**or** friend:$\mathsf{id}_{1,1}$ ... friend:$\mathsf{id}_{1,N}$) to get the entities that are more than one edge away from the user id_1 in the encrypted graph-structured data.

GraphSE^2 defines a query structure to construct the **apply** operator. In details, the query structure is a tuple of (*prefix*, *s*, *filter*), where the *prefix* (e.g., friend:) is prepended to the given id to form the indexing terms, *s* is an s-expression with N indexing terms (e.g., (**term ?**)), and *filter* indicates the ranking algorithm for its results. To execute an **apply** opera-

Algorithm 3 Apply

Input: Outer Query Structure QS_O, Nested Query Structure QS_N, Array of id
Output: Encrypted Result O

function APPLY(QS_O, QS_N, id)(in $\mathcal{SF}$)
  for $i = 1 : QS_N.s.size$ **do**
    $term \leftarrow QS_N.prefix||\mathsf{id}[i]$;
    $QS_N.s[i] \leftarrow term$;
  end for
  $o \leftarrow \text{Search}(QS_N.s, QS_N.filter)$;
  for $i = 1 : o.size$ **do**
    $term \leftarrow QS_O.prefix||o[i].id$;
    $QS_O.s[i] \leftarrow term$;
  end for
  $O \leftarrow \text{Search}(QS_O.s, QS_O.filter)$;
  return O;
end function

Input: Query q, Result Filter f
Output: Encrypted Result r

function SEARCH(q, f)(in $\mathcal{IS}$)
  $r \leftarrow f(Execute(q))$;
  return r
end function

tor, $\mathcal{SF}$ pre-processes the input id with a given *prefix* in the query structure and uses processed id to execute the s-expression s from the query structure. $\mathcal{ISC}$ handles the query from the s-expression and applies the designated *filter* in the query structure to refine the result. Consequently, $\mathcal{SF}$ can retrieve a list of id as the result of the nested query structure. GraphSE^2 leverages as input the retrieved id and outer query structure to repeat the above procedure until it reaches the outermost query structure. Algorithm 3 gives the detailed implementation of **apply**.

The **apply** operator processes the necessary steps on behalf of its users to improve the efficiency of GraphSE^2. For example, it would be possible for users to ask recommendation from friends: both $\mathcal{SF}$ and client can execute a two-step query to retrieve friend list in advance and issue an additional query to get the recommendation. Compared to the latter strategy, the **apply** operator runs in $\mathcal{SF}$ can highly reduce the workload on the client-side: it saves the network latency of transmitting intermediate result between $\mathcal{SF}$ and client, addition to the computational cost of aggregation and regeneration. Furthermore, $\mathcal{SF}$ can further optimise the query and adopt the different scoring strategy by giving its semantic context (as shown in the following example).

Example: Friend Recommendation. The friend recommendation is a good example of the use of the **apply** operator. According to the Homophily theory [103], people with higher similarity have a higher probability of becoming friends. In this context, the system aims to recommend the friends-of-friends to its user according to the order of similarities. Hence, we apply a simple ranking function which returns the sorted similarity value directly for both outer and nested query for this application.

Table 3.3: Performance estimation of Friend Recommendation implementations with 1 million users.

	friends-of-friends	**apply**
Est. # of users/posting list	130	48k
Est. Storage overhead	7.28GB	2.4TB
Query delay	200 s	1-2 s

It is also possible to implement the friends-of-friends query without the **apply** operator. Intuitively, friends-of-friends also can be treated as an edge type, GraphSE^2 may explicitly store the friends-of-friends list and use the indexing term friends-of-friends:id to index it. Therefore, the friend recommendation problem is easily processed by the **term** operator. However, such a naive solution blows up the memory consumption on $\mathcal{IS}$: as shown in Table 3.3, the estimated size of friends-of-friends posting list is almost 370x larger than the original friend list of a typical user [29]. In GraphSE^2, each encrypted tuple occupies 56 bytes memory space (See Section 3.9.2 for the detailed discussion), it indicates that the friends-of-friends posting lists for 1 million users consume 2.4TB RAM.

The **apply** operator also reduces the query latency: it is expensive to sort the posting lists of friends-of-friends:id with 48k entities inline in garbled circuit (it needs 200 s to evaluate the corresponding circuit). In comparison, introducing an extra round to query enables $\mathcal{IS}$ to truncate the result, and makes the sorting process more efficient. For example, if GraphSE^2 applies a *filter* to return the top 10 results to $\mathcal{SF}$ for the nested query, the result size can be reduced to around 1000 users with a higher similarity. Also, the reduced result size is moderate to evaluate sort circuit on it. Under our settings in Section 3.9, the system replies either a full result list after 5-6 s or a truncated result list after 1-2 s.

3.7 Security Analysis

In Section 3.5.3, we discuss the security of each atomic operation. Here, we analyse the security of the overall system. Specifically, we formulate the security of GraphSE^2 based on the prior work of SSE [19, 30] and further combine the security of additive sharing and garbled circuit to depict the security of query operators. Throughout the analysis, we consider a query in GraphSE^2 containing a boolean formula ψ and a tuple of indexing terms $(t_1, t_2, ..., t_n)$.

Overview. The main idea of analysing the security of GraphSE^2 is similar to that in SSE scheme [19, 30]. Specifically, the analysis constructs a simulator of GraphSE^2 to show that the adversary in GraphSE^2 only learns the controlled leakage parameterised by a leakage function $\mathcal{L}$, after querying a vector of queries $\mathbf{q}$. Note that the simulator of GraphSE^2 is slightly different from the original SSE simulator, we outline these different points as the sketch of our security analysis. Firstly, we update the leakage function of SSE (OXT in GraphSE^2) to additionally capture the ranks leaked in query results. Secondly, we slightly modify the

capability of adversaries to fit our two-party model: an adversary in our system is able to see the view on the corrupted cluster as well as the output of the counter-party. Under the new adversary model, the joint distribution of the outputs of both the adversary and the counter-party can be properly simulated by an efficient algorithm with the updated leakage function. Finally, as GraphSE^2 has the submodules implemented by SSE, additive share scheme, and Yao's garbled circuit, the simulator of GraphSE^2 can be constructed by combining the simulators of these submodules. For instance, our simulator uses the output of SSE simulator as the input of garbled circuit simulator in order to simulate the query operators with structured data access and sorting.

3.7.1 Leakage Function

Recall the security definition from [19, 30]: The security of SSE is parameterised by a leakage function $\mathcal{L}_{\mathbf{SSE}}$, which depicts the scope of information about data and queries that the adversary is allowed to learn through the interaction with the server.

Therefore, we start by giving the leakage function of query operators in GraphSE^2. As shown in Section 3.5.3, all operators in GraphSE^2 inherit the leakage of OXT [19] (i.e., TSet leakage $\mathcal{L}_{\mathbf{T}}$ and OXT leakage $\mathcal{L}_{\mathsf{OXT}}$).

Furthermore, all operators produce a sorted list as the return; it additionally introduces a new leakage about the rank of retrieved entities. We define a new leakage function $\mathcal{L}_{\mathrm{R}}$ to capture this new leakage. In particular, $\mathcal{L}_{\mathrm{R}}$ consists of two sub-functions $\{f_0, f_1\}$ which are defined as follows:

- f_0: it takes as input the transcript of OXT and outputs a sorting circuit F_{sort} and its input labels.
- f_1: it takes as input the transcript of OXT and outputs the rank r_{id} for every e_{id}, where $r_{\mathsf{id}} \in \mathcal{N}$ is the rank of e_{id}.

Note that the adversary can only corrupt one of the two parties in our system which means the adversary only can access one of the above two sub-functions during the simulation.

The overall leakage function $\mathcal{L}$ consists of the leakage from OXT as well as $\mathcal{L}_{\mathrm{R}}$.

3.7.2 Security Proofs

Based on the above leakage function, we give a security analysis for GraphSE^2 following the real/ideal paradigm.

To start with, we define the real/ideal models for the query operators (e.g., **term**, **and**, etc.) that only involve structured data access and garbled circuit sorting; we denote the execution of the above query operators as query protocol Π_1.

The query protocol Π_1 is executed between a client C ($\mathcal{SF}$) and two parties $P_i, i \in \{0,1\}$ ($\mathcal{ISC}$s). In the real model, an adversary $\mathcal{A}$ can choose a social dataset DB and let C generate the corresponding encrypted database EDB and give it to $\mathcal{A}$. Then, $\mathcal{A}$ chooses a query list $\mathbf{q}$ to run in the two-party servers. To respond, each party firstly accesses EDB to retrieve contents and gives the transcript to $\mathcal{A}$. Later, they perform two-party sorting via garbled circuit scheme. During the sorting, $\mathcal{A}$ is able to see all inbound/outbound messages in the corrupted party.

We let $\mathsf{VIEW}^i_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ be the entire view of P_i in an execution of Π_1. Let $\mathsf{OUT}^i_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ be the outputs of party P_i at the end of a Π_1 execution. Considering the security assumption of untrusted but non-colluded adversaries on the two-party setting, an adversary $\mathcal{A}$ can corrupt one party at most. The assumption restricts that $\mathcal{A}$ only can get the entire view of the corrupted party and the outputs from the counter-party. Hence, $\mathsf{VIEW}^i_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ and $\mathsf{OUT}^{(1-i)}_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ are exactly the real model of $\mathcal{A}$ who corrupts P_i. In this case, we denote the adversary as $\mathcal{A}_i$ and set:

$$\mathbf{Real}_{\Pi_1,\mathcal{A}_i}(1^\lambda, \mathsf{DB}, \mathbf{q}) \stackrel{def}{=} (\mathsf{VIEW}^i_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q}), \mathsf{OUT}^{(1-i)}_{\Pi_1}(1^\lambda, \mathsf{DB}, \mathbf{q}))$$

In the ideal model, the EDB of the chosen DB is generated by the simulator of OXT $\mathcal{S}_{\mathsf{OXT}}$. In the query phase, each party processes the chosen query list $\mathbf{q}$ and returns the transcript to $\mathcal{A}$ via $\mathcal{S}_{\mathsf{OXT}}$. Then, they hand their inputs to a trusted party TP to perform sorting. We denote the above protocol executed in the ideal model as $\Pi_1^{'}$. The view of $\mathcal{A}_i$ in the ideal model consists of the view on P_i and the output of counter-party $P_{(1-i)}$. We set:

$$\mathbf{Ideal}_{\Pi_1',\mathcal{A}_i}(1^\lambda, \mathsf{DB}, \mathbf{q}) \stackrel{def}{=} (\mathsf{VIEW}^i_{\Pi_1'}(1^\lambda, \mathsf{DB}, \mathbf{q}), \mathsf{OUT}^{(1-i)}_{\Pi_1'}(1^\lambda, \mathsf{DB}, \mathbf{q}))$$

Before we formalise the security of Π_1, we give the definition of *computationally indistinguishable*:

Definition 1 (Computationally Indistinguishable)**.** *Assuming a distribution ensemble $\mathcal{X} = \{\mathcal{X}_i\}_{i\in\mathcal{I}}$ is a sequence of random variables indexed by $\mathcal{I}$. Two distribution ensembles $\mathcal{X} = \{\mathcal{X}_i\}_{i\in\mathcal{I}}$ and $\mathcal{Y} = \{\mathcal{Y}_i\}_{i\in\mathcal{I}}$ are computationally indistinguishable, denoted as $\mathcal{X} \stackrel{c}{\equiv} \mathcal{Y}$, if for every probabilistic polynomial-time (PPT) distinguisher $\mathcal{D}$, there exists a negligible function $\mathbf{negl}(\cdot)$, such that for every $i \in \mathcal{I}$:*

$$|\Pr[\mathcal{D}(\mathcal{X}_i) = 1] - \Pr[\mathcal{D}(\mathcal{Y}_i) = 1]| \leq \mathbf{negl}(\lambda)$$

.

The security of Π_1 is defined as follows:

Definition 2. *Let Π_1, $\Pi_1^{'}$ be as above. protocol Π_1 is said to be $\mathcal{L}$-semantically secure where $\mathcal{L}$ is the leakage function defined as before if for every non-adaptive adversary $\mathcal{A}_i$ in the real model, there exists a simulator $\mathcal{S}_i$ in the ideal model such that:*

$$\{\mathbf{Real}_{\Pi_1,\mathcal{A}_i}(1^\lambda, \mathsf{DB}, \mathbf{q})\}_{i\in\{0,1\}} \stackrel{c}{\equiv} \{\mathbf{Ideal}_{\Pi_1',\mathcal{S}_i}(1^\lambda, \mathsf{DB}, \mathbf{q})\}_{i\in\{0,1\}}$$

We further define our adaptive model as in [19]. In such a model, the query list $\mathbf{q}$ is not given to the challenger in the above two games. Instead, $\mathcal{A}$ adaptively chooses each query after receiving EDB.

We assume that Π_1 runs in a *hybrid model* where parties are given access to the trusted party computing the ideal function of OT. We show that Π_1 is secure with the given leakage function in this hybrid model. It follows from the standard composition theorems [104] that Π_1 is secure with the given leakage function if the trusted party is replaced by secure protocol (i.e., real OT).

Theorem 1. *In the* OT*-hybrid model execution, protocol* Π_1 *is* $\mathcal{L}$*-semantically secure against non-adaptive adversaries, assuming that* OXT *protocol is* $\mathcal{L}_{\mathsf{OXT}}$*-semantically secure against non-adaptive adversaries, that the garbled circuit construction is secure against semi-honest but non-colluding adversaries.*

Proof. Let $\mathcal{A}_i$ denote an adversary corrupts P_i attacking the protocol in a hybrid model where the parties run OT via trusted entities (OT-hybrid model). As the view of garbler and evaluator are different in Π_1, we consider separately the cases $i = 0$ and $i = 1$. We further denote the ensemble of C, the counter-party of P_i and the trusted party TP as the challenger $\mathcal{C}$ in our simulation. In both cases, we show that we can construct a simulator $\mathcal{S}_i$ running in our ideal model where the parties involve a trusted entity computing $\Pi_1^{'}$, which returns the same view as $\mathcal{A}_i$'s in the hybrid model.

$\mathcal{S}_i$ is constructed as follows:

1. $\mathcal{S}_i$ chooses the DB and $\mathbf{q}$ and simulates $\mathcal{A}$. It firstly gives DB to $\mathcal{C}$. $\mathcal{C}$ runs $\mathcal{S}_{\mathsf{OXT}}(\mathcal{L}_{\mathsf{OXT}}(\mathsf{DB}))$ and return EDB to $\mathcal{S}_i$.
2. For the given query list $\mathbf{q}$, $\mathcal{S}_i$ processes $\mathcal{S}_{\mathsf{OXT}}(\mathcal{L}_{\mathsf{OXT}}(\mathsf{DB}, q))$ and receives the output $\mathbf{tr}_i$ from $\mathcal{S}_{\mathsf{OXT}}$. The output $\mathbf{tr}_i$ is identical to the transcript of OXT except that the query result of OXT only includes the encrypted id e_{id}, but in $\Pi_1^{'}$, the query result is a list of encrypted tuples $\{\mathbf{E}\} = \{(\langle x \rangle^A, e_{\mathsf{id}})\}$.
   This completes the simulation of the structured data access part of Π_1.
3. Next, $\mathcal{S}_i$ hands $\mathbf{tr}_i$ to $\mathcal{C}$ and simulates the sorting procedure. As each party runs the same query on the same partition of the OXT index, the size of the transcript is equal. $\mathcal{C}$ runs $f_0(\mathbf{tr}_0, \mathbf{tr}_1)$ to generate an array of sorting circuits $\mathbf{F}$ with $|\mathbf{tr}|$ circuits and $f_1(\mathbf{tr}_0, \mathbf{tr}_1)$ to generate an array of ranks $\mathbf{R}[e_{\mathsf{id}}]$ indexing by the e_{id} from $\mathbf{tr}$.
4. For each $\mathbf{tr}_i[j], 1 \leq j \leq |\mathbf{tr}|$, if $i = 0$ (garbler):
   (a) $\mathcal{S}_0$ sends $\mathbf{tr}_0[j]$ to the $\mathcal{C}$ and receives $\mathbf{F}[j]$ and its input labels $\langle \mathrm{X} \rangle_0^Y$ from $\mathcal{C}$. In addition, $\mathcal{S}_0$ is able to get the labels $\langle \mathrm{X}_0 \rangle_1^Y$ of its additive shares $\langle \mathrm{X} \rangle_0^A$. There is no message from the counter-party.
   (b) The counter-party (P_1) uses its $\mathbf{tr}_1[j]$ to retrieve a $\mathcal{L}_{\mathrm{R}}$-simulated ranked list $\{\bar{\mathbf{E}}\} = \{(r_{\mathsf{id}}, e_{\mathsf{id}})\}$ from $\mathbf{R}[e_{\mathsf{id}}]$.

(c) After execution, $\mathcal{S}_0$ outputs $\{\mathsf{EDB}, \mathbf{tr}_0[j], \mathbf{F}[j], \langle \mathrm{X} \rangle_0^Y, \langle \mathrm{X}_0 \rangle_1^Y\}$ as $\mathsf{VIEW}^0_{\Pi'_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ and $\{\{\bar{\mathbf{E}}\}\}$ as $\mathsf{OUT}^1_{\Pi'_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$.

5. If $i = 1$ (evaluator):
   (a) $\mathcal{S}_1$ simulates the OT protocol by generating the labels $\langle \mathrm{X}_1 \rangle_1^Y$ for each evaluator's input $\langle x \rangle_1^A$.
   (b) $\mathcal{S}_1$ retrieves ranks from $\mathbf{R}[e_{\mathsf{id}}]$ and constructs $\{\bar{\mathbf{E}}\}$.
   (c) Then, in the usual way (e.g. [92, 94]), $\mathcal{S}_1$ simulates a sorting circuit F_{sort} in such a way that given the input labels chosen by $\mathcal{S}_1$, the output of the circuit is precisely $\{\bar{\mathbf{E}}\}$.
   (d) Finally, $\mathcal{S}_1$ outputs $\{\mathsf{EDB}, \mathbf{tr}_1[j], F_{sort}, \langle \mathrm{X}_0 \rangle_1^Y, \langle \mathrm{X}_1 \rangle_1^Y, \{\bar{\mathbf{E}}\}\}$ as $\mathsf{VIEW}^1_{\Pi'_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$ and $\{\emptyset\}$ as $\mathsf{OUT}^0_{\Pi'_1}(1^\lambda, \mathsf{DB}, \mathbf{q})$.

To complete the proof, we use the above simulator to show that

$$\{\mathbf{Real}_{\Pi_1, \mathcal{A}_i}(1^\lambda, \mathsf{DB}, \mathbf{q})\}_{i \in \{0,1\}} \stackrel{c}{\equiv} \{\mathbf{Ideal}_{\Pi'_1, \mathcal{S}_i}(1^\lambda, \mathsf{DB}, \mathbf{q})\}_{i \in \{0,1\}}$$

1. EDB and $\mathbf{tr}$ in the real model are generated by running the OXT protocol, while in the ideal model, they are simulated by given the output of the leakage function.
2. For $i = 0$ (garbler):
   (a) In the real model, the garbler constructs the sorting circuit correctly based on garbler's inputs, while in the ideal model the garbled circuit is a dummy circuit in $\mathbf{F}$ with the given input labels from $\mathcal{C}$.
   (b) In the real model, the evaluator gets the output $\{\bar{\mathbf{E}}\}$ after evaluating the circuit. In the ideal model, the output is obtained from the trusted entity via the output of leakage function $\mathcal{L}_{\mathrm{R}}$.
3. For $i = 1$ (evaluator):
   (a) In the real model, F_{sort} and the input label sent by the garbler is correctly generated based on garbler's inputs, while in the ideal model the garbled circuit is simulated based on the randomly generated input labels and the output of leakage function $\mathcal{L}_{\mathrm{R}}$ from $\mathcal{C}$. It ensures that the circuit returns the same output in two models.
   (b) Both in the real and ideal model, the garbler does not output a result at the end of the protocol execution.

The security properties of OXT ensures the indistinguishability of structured data access part. For the sorting part, The privacy property of garbled circuit [92] ensures that the adversary only can learn the inputs from the circuit and output with a negligible probability. It concludes that for every non-adaptive adversary $\mathcal{A}_i$, it has a negligible probability to learn more information than the defined leakage function $\mathcal{L}$. □

We now show that our theorem is also valid for adaptive models.

Theorem 2. *In the* OT*-hybrid model execution, protocol* Π_1 *is* $\mathcal{L}$*-semantically secure against adaptive adversaries, assuming that* OXT *protocol is* $\mathcal{L}_{\mathsf{OXT}}$*-semantically secure against adaptive adversaries, that the garbled circuit construction is secure against semi-honest but non-colluding adversaries.*

Proof. First of all, OXT protocol has been proved to be $\mathcal{L}_{\mathsf{OXT}}$-semantically secure against adaptive adversaries [19]. The only concern here is how to handle the adaptivity towards sorting.

To simulate the response for adaptive queries, the simulator should adaptively generate the ranks $\mathbf{R}[e_{\mathsf{id}}]$ for e_{id} list This is in contrast to the non-adaptive simulator, where it can generate $\mathbf{R}[e_{\mathsf{id}}]$ as determined by the leakage. Instead, the simulator uses the STag and e_{id} list from the transcript to maintain a bidimensional array $\mathbf{R}[STag, e_{\mathsf{id}}]$. It adaptively updates $\mathbf{R}[STag, e_{\mathsf{id}}]$ for each new e_{id} which does not exist in $\mathbf{R}[STag, e_{\mathsf{id}}]$. □

Finally, we show that the defined operator (i.e., **apply**) is also secure after involving arithmetic computations. We slightly modify our hybrid model by adding an ideal function f_A, which evaluates arbitrary addition/multiplications using the ideal function of additive sharing scheme. The new hybrid model is defined as (f_A,OT)-hybrid model.

Theorem 3. *In the (*f_A*,*OT*)-hybrid model execution, protocol* Π_1 *is* $\mathcal{L}$*-semantically secure against adaptive adversaries, assuming that* OXT *protocol is* $\mathcal{L}_{\mathsf{OXT}}$*-semantically secure against adaptive adversaries, that the garbled circuit construction is secure against semi-honest but non-colluding adversaries.*

Proof. We have shown that the output ranks can be simulated without using the actual additive shares in Theorem 1, 2. Informally, it also indicates that Π_1 can also securely perform structured data access and sorting under the (f_A,OT)-hybrid model. In addition, previous work shows that the additive sharing scheme can securely compute the given arithmetic formulas [101]. The standard sequential modular composition [104] implies that Π_1 with sub-protocols evaluating f_A and OT remains secure with the same leakage $\mathcal{L}$ in the real model. □

Discussion. Note that there exist some emerging threats against the building blocks of GraphSE^2. Regarding SSE, leakage-abuse attacks [36, 105] can help an attacker to explore the information learned during queries. To mitigate them, recent studies on padding countermeasures [105, 106] and forward/backward privacy [107, 108] are proposed and shown to be effective. We leave the integration of these advanced security features to our system as future work. Regarding sorting, GraphSE^2 reveals the rank of the query result. Recent work [109, 110] demonstrates that the underlying data values are likely to be reconstructed

if an adversary knows ranks and some auxiliary information of queries and datasets. Currently, we do not consider such a strong adversary, and how to fully address the above threat remains as an interesting problem.

3.8 Implementation

We implement a prototype system for evaluating the performance of GraphSE^2. To build this prototype, we first realise the cryptographic primitives in Section 3.3. Specifically, we use the symmetric primitives, i.e., AES-CMAC and AES-CBC, from Bouncy Castle Crypto APIs [111]. In addition, we use a built-in curve from Java Pairing-based Cryptography [112] library (Type A curve) to support the group operations in OXT. The security parameter of symmetric key encryption schemes is 128-bit, and the security parameter of the elliptical curve cryptographic scheme is 160-bit. Regarding the secure two-party computation, we set the field size to 2^{31}. Therefore, we can use regular arithmetics on Java integer type to implement the modulo operations, as it is significantly faster than the native modulo operation in Java BigInteger type (i.e., we observed that it is 50x faster). We involve this optimisation into the implementation of additive sharing scheme in the finite field $\mathbb{Z}_{2^{31}}$, the addition (multiplication) operations is calculated by several regular addition and multiplication operations with the modulo operation. The oblivious transfer and garbled circuit protocols are implemented by using FlexSC [113]. It implements the extended OTs in [93] and several optimisations for the garbled circuit, which make it a practical primitive under Java environment.

The prototype system consists of three main components: the *encrypted database generator*, the *query planner* in $\mathcal{SF}$ and the *index server daemon* in $\mathcal{ISC}$. The *encrypted database generator* is running on a cluster with Hadoop [114]. It partitions the plaintext data, runs the adapted OXT to convert the data into encrypted tuples with the additive share of sort-keys and stores these tuples on each $\mathcal{IS}$. We leverage Spark [80] to execute these tasks in-memory and enable the pipelining data processing to further accelerate this process. The generated tuples are stored in the in-memory key-value store Redis [81] in the form of TSet on each $\mathcal{IS}$ for querying purpose later. In addition, the generated XSet is kept in the external storage of each $\mathcal{IS}$ to support the set operations. All queries are handled by the *query planner* and *index server daemons* by following the query processing flow in Section 3.4.1. Thrift thread pool proxy [82] is deployed to handle the queries in *index server daemons*.

To improve the runtime performance of our prototype, each posting list is segmented into fixed-size blocks indexed by its $\mathsf{stag}(t)$ and a block counter c for the stag. As the final result of the block counter indicates the total number of blocks for each stag, it is also stored in Redis after the whole posting list is converted to encrypted tuples. Those counters enable $\mathcal{IS}$ to retrieve multiple tuples in parallel. We also introduce a startup process for OXT protocol and secure two-party computation in *index server daemons*. In terms of the OXT matching, the index server daemon creates a Bloom filter [115] to load the XSet into memory during the

Table 3.4: Statistics of Youtube social network dataset.

Node type	# of nodes	Edge type	# of edges
User	1157827	**friend**	4945382
Group	30087	**follow**	293360

startup process. In our prototype, we deploy the Bloom filter from Alexandr Nikitin as it is the fastest Bloom filter implementation for Java [116]. We set the false positive rate to 10^{-6}, and the generated Bloom filter only occupies a small fraction of $\mathcal{IS}$ memory. Besides generating the Bloom filter, each index server also pre-computes several multiplication triplets and sorting circuits and periodically refreshes it to avoid extra computational cost on-the-fly.

Our prototype system implementation consists of four main modules with roughly 3000 lines of Java code, we also implement a test module with another 1000 lines of Java code.

3.9 Experimental Evaluations

3.9.1 Setup

Platform. We deploy the *index server daemons* in a cluster ($\mathcal{ISC}$) with 6 virtual machine instances in the Microsoft Azure platform. All instances are E4-2s v3 instances, configured with 2 Intel Xeon E5-2673 v4 cores, 32GB RAM, 64GB SSD external storage and 40Gbps virtualised NIC. Another D16s v3 instance is created in Azure to run $\mathcal{SF}$ with the *query planner* and client; it is equipped with 16 Intel Xeon E5-2673 v3/v4 cores, 64GB RAM, 128GB SSD external storage and 40Gbps virtualised NIC. We also have the other three E4-2s v3 instances controlled by $\mathcal{SF}$; we use them to run the *encrypted database generator* for generating the encrypted index. All instances are installed with Ubuntu Server 16.04LTS.

Dataset. We use a Youtube dataset from [33], which is an anonymised Youtube user-to-user links and user group memberships network dataset. The detailed statistical summary is given in Table 3.4. We recognise the user-to-user links as **friend** edge and user group memberships as **follow** edge from this dataset. The generated posting lists are indexed by above two edge types and user ids. As the social network in our Youtube dataset is an unweighted network, we randomly generate a weight between 1 and 100 for each edge to evaluate the arithmetic and sort operations of GraphSE^2.

Baseline. To evaluate the performance of GraphSE^2, we create a graph search system by removing/replacing cryptographic operations in this baseline system. Specifically, we leverage a hash function to generate xtags instead of using expensive group operations. The index and sort-key are stored in plaintext, which means that the $\mathcal{IS}$ can compute and sort without any network communication for OT and multiplication triplets. Finally, the query planner provides the indexing term in plaintext instead of the stag to the $\mathcal{IS}$ as the query token. Nonetheless, we still use the PRF value of the indexing term and the block counter

TSet Tuple

16 bytes Indexing term	**36 bytes** Encrypted index	**4 bytes** Additive share

Plaintext Tuple

16 bytes Indexing term	**8 bytes** Plaintext index	**4 bytes** sort-key

XSet (xtag)

In ciphertext	**128 bytes**
In plaintext	**16 bytes**

Block Counter (same in Tset and Plaintext)

16 bytes Indexing term	**4 bytes** # of blocks

Figure 3.5: A tuple-wise storage overhead comparison between the encrypted database and plaintext database.

as tuple index, because we want to keep the table structure of TSet unaltered to make our system comparable to the baseline. We use this baseline system to evaluate the overhead from cryptographic operations as GraphSE^2 implements the same operators as Facebook Unicorn [29].

3.9.2 Evaluation Results

EDB Generation. Firstly, we demonstrate the runtime performance of the *encrypted database generator*. The generator needs to partition and create additive shares from the original plaintext data, and to generate the adapted OXT index for each $\mathcal{IS}$. GraphSE^2 uses $\mathcal{SF}$ to locally generate the partitions and additive shares for our dataset and then uses the dedicated cluster to generate the encrypted graph index in parallel. The result on our 5 million records dataset shows that it only takes 54 s to pre-process data on $\mathcal{SF}$ and 7.4 mins to generate the encrypted index via Spark.

Storage. Recall that GraphSE^2 uses adapted OXT index to support boolean queries over the encrypted graph, which needs to generate two dedicated data structure (i.e., TSet and XSet). As a result, GraphSE^2 consumes more storage capacity than the baseline system (see Figure 3.5), because it is required to keep more information (i.e., xtag in ciphertext), and because it stores encrypted index which is larger than the corresponding plaintext. By using the TSet, we observe that our system increases the memory consumption of Redis by 85% (557MB in TSet and 300MB in plaintext), which is slightly smaller than the theoretical memory consumption overhead (100% according to Figure 3.5). The reason is that GraphSE^2 also keeps the number of blocks of each posting list (see Section 3.8) to accelerate the tuple retrieving process[3]. As shown in Figure 3.5, the block counter requires an additional 20 bytes of memory consumption for each indexing term both in TSet and in plaintext. It introduces the same extra cost on GraphSE^2 as well as in the baseline system and makes the memory consumption overhead smaller than the theoretical expectation.

[3]If the size of the posting list is unknown, the system needs to sequentially retrieve the tuple from blocks, as the key is derived from stag and block counter. Otherwise, the tuples can be retrieved in parallel.

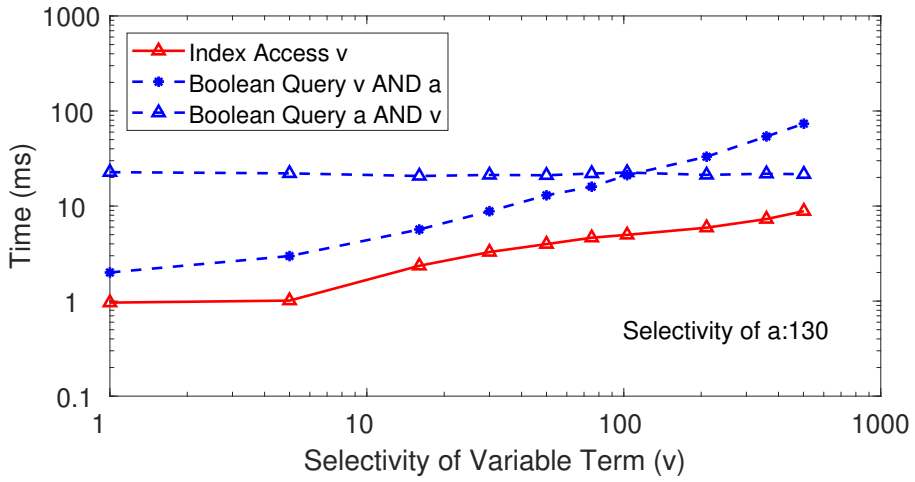


Figure 3.6: Query delay for two-keyword set queries.

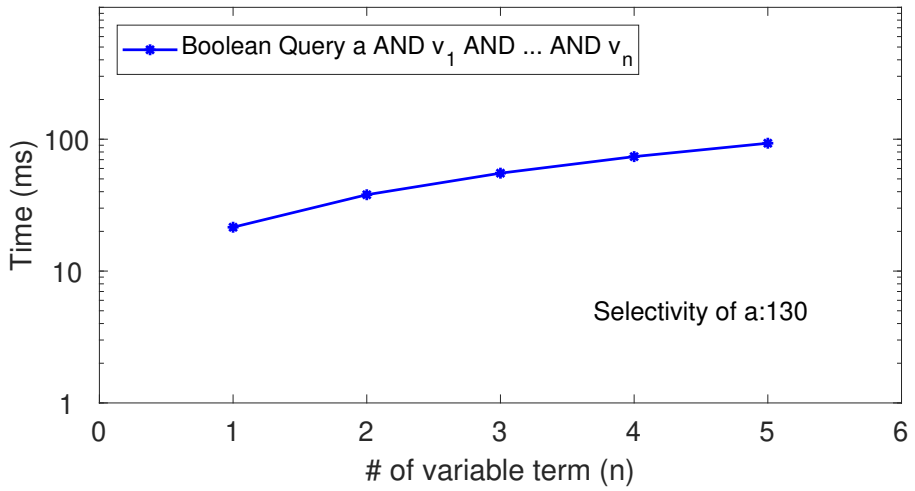


Figure 3.7: Query delay for multiple-keyword set queries.

But the Bloom filter successfully saves the memory consumption in runtime, because the size of the Bloom filter only depends on the false positive rate and the number of total elements inside (the number of edges in our system) [115], and it is much smaller than XSet itself. By fixing the false positive rate to 10^{-6}, the runtime overhead of XSet in our system is identical to the baseline system (only 18MB in RAM).

Query Delay. To understand the query delay introduced by cryptographic primitives, we measure the cryptographic overhead from these cryptographic primitives independently. To evaluate the query delay introduced by the set operations, we choose an indexing term **a** from **friend** edges with fixed selectivity (130, as the average user has 130 friends according to Unicorn paper [29]), and we further choose several variable indexing terms **v** from **friend** edges with selectivity from 1 to 502. Figure 3.6 illustrates query delay on Index Access **v** and two variants of two-term Boolean Query. In Index Access **v** query, the query only consists of s-term **v**, and the figure shows that its execution time is linear to the size of corresponding posting lists. The other two-term queries combine the previous queries with the fixed term **a**. In the first of these two queries, we use **a** as x-term, each tuple from $\mathsf{TSet}(\mathbf{v})$ should be

Table 3.5: Benchmark of sorting circuit size and evaluation time, the garbled sorting algorithm is bitonic merging/sorting, we use it to sort 2^l vector.

Vector length	2	4	8	16	32	64	128
# of AND Gates	4382	9148	19448	41968	91616	201664	446336
GC evaluation time (ms)	17.3	20.3	31.5	48.2	101.0	206.5	440.0
GC comm. overhead (MB)	0.12	0.41	0.46	0.97	2.10	4.49	9.80

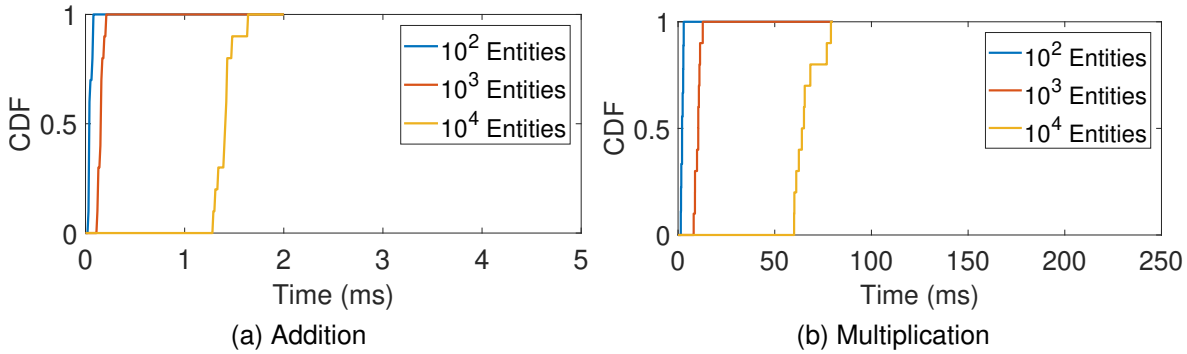


Figure 3.8: The execution time for addition and multiplication operations on two vectors with 10^2, 10^3, 10^4 entities.

checked wtih the cost of an exponentiation, which requires 2 to 79 ms to process. In the last one, **a** is used as s-term, where we observe that the execution time is kept invariable (20 ms), irrespective of the variable selectivity of the xterm **v**. It demonstrates that GraphSE^2 can respond to a query related to the moderate users with a tiny latency, and it is also able to reply the query about popular users with a slightly bigger but still modest delay.

The secure addition and multiplication are supported by additive sharing scheme with a relatively small overhead, because in most cases, the two servers non-interactively do the computation tasks by using regular arithmetic operations, and because the expensive tasks, such as multiplication triplets generation, are pre-computed in the startup process. Figure 3.8a and 3.8b demonstrate the execution delay of addition and multiplication over the different size of vectors. We can see that the addition operation with two vectors containing 10^4 entities can be done within 2 ms. For the multiplication operation, it needs 80 ms to compute the product of 10^4 entities because it requires several efficient but non-negligible communications with the counter-party.

As the sorting algorithm is implemented by the garbled circuit, we provide a benchmark about the size of the circuit and the corresponding evaluation time. The results are listed in Table 3.5. Note that we do not report the circuit generation time as it is generated in the startup process. The reported result demonstrates the practicality of our sorting strategy: the local sorting generally involves fewer entities after partition (30 if the system sorts a result list with 130 users), which takes 100 ms to sort. Additionally, the local sort can help to truncate the result before sending it for global sorting, which makes the sorting algorithm in garbled circuit more efficient (less than 440 ms for a vector with 128 entities).

Table 3.6: Throughput (Queries/sec) comparison of global sorting for query with different operators.

Operators	**term**	**and/diff.**	**or**
Baseline	350	321	301
Our system	80	80	80

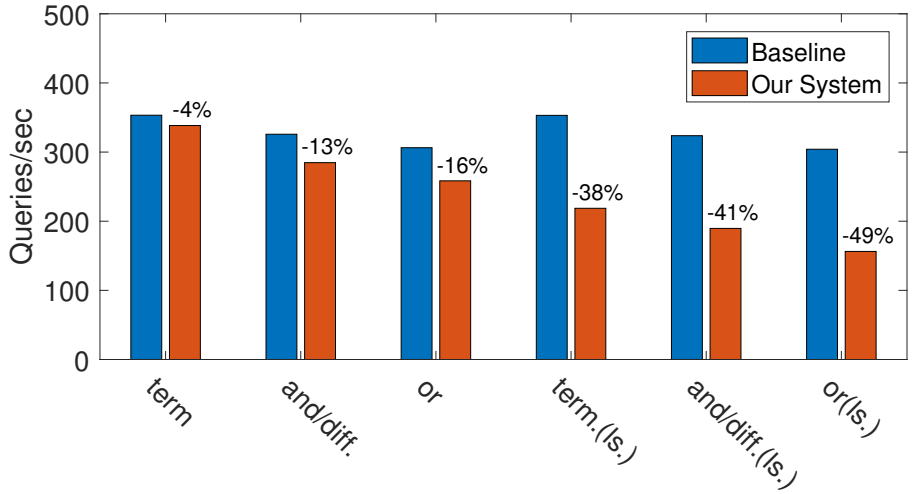


Figure 3.9: Throughput of different types of Query Operators with 10000 concurrent clients running under GraphSE^2 and baseline, all operators except **term** have two keywords. Diff. stands for **difference** operator; ls. stands for locally sorted in $\mathcal{IS}$ after applying the operators.

We further examine the query delay from set operations under multiple-keyword setting by using the same **a** as s-term, but we add more variable terms $\{\mathbf{v_n}\}, \mathbf{n} \in [1, 5]$ as x-term. Figure 3.7 shows that each additional x-term increases the query delay by 20 ms, which means a query with 5 x-terms can still be processed within 100 ms.

Communication. We measure the inter-cluster communication overhead because it includes the main communication overhead in GraphSE^2, which is to send the garbled sorting circuit as well as the labels of inputs to the counter-party. Note that this is much larger than sending the multiplication triplets (12 bytes for each) and the encrypted id (16 bytes for each). We demonstrate the communication overhead for different size of the circuit in Table 3.5. It shows that for an average user with approximately 130 friends, the two-party only requires to transmit 9.80MB data to sort them. This overhead is negligible both in our evaluation platform (40Gbps NIC in Azure intranet) and the other public clouds such as AWS.

Throughput. To evaluate the impact of our system on throughput, we measure the server throughput for different types of operators. For each operator, we compare the throughput results between GraphSE^2 and the baseline. Figure 3.9 and Table 3.6 show the throughput test results for GraphSE^2 and baseline. In all the cases, we group our 6 instances into 2 clusters with 3 instances to fulfil the two-party settings. All instances are running with only one core involved in the computation, and we simulate 10000 parallel client processes to send the query to the server, which ensures 100% workload on the server-side. The results

show that the throughput penalty is mainly from the sorting: the local sorting decreases the throughput by 38% to 49%. We also observe that the global sorting is the bottleneck of the whole system (see Table 3.6), it gives a constant query throughput for all operators, which means it runs longer to obtain the final result. However, for the operators without sorting, the throughput loss is modest (only 4% to 16%).

3.10 Conclusion

This chapter presents an encrypted graph database, named GraphSE^2. It enables privacy-preserving rich queries in the context of social network services. Our system leverages the advanced cryptographic primitives (i.e., OXT, and the mixing protocol with the additive sharing and the garbled circuit) with strong security guarantees for queries on structured social graph data, and queries with computation, respectively. To lead to a practical performance, GraphSE^2 generates an encrypted index on a distributed graph model to facilitate parallel processing of the proposed graph queries. GraphSE^2 is implemented as a prototype system, and our evaluation on YouTube dataset illustrates its efficiency on social search.

Chapter 4

Privacy-Preserving Outlier Detection

Outlier detection is widely used in OSN for various application scenarios, e.g., contextual filtering and spam detection. In this chapter, we implement a privacy-preserving outlier protocol for streaming data based on GraphSE^2 to demonstrate the practicality of GraphSE^2. The protocol decomposes the outlier detection algorithm into several phases and recognises the necessary cryptographic operations in each phase. It realises several cryptographic modules via the operators defined by GraphSE^2 to support the above cryptographic operations and composes them in the overall protocol to enable outlier detection over encrypted datasets. To support efficient updates, it integrates the sliding window model to periodically evict the expired data in order to maintain a constant update time. We build a prototype of PPOD and systematically evaluate the cryptographic modules and the overall protocols under various parameter settings. Our results show that PPOD can handle encrypted incremental datasets with a moderate computation and communication cost.

4.1 Introduction

The increasing demands of local and global secrecy and private computations inside a cloud atmosphere reveal essential requirements and commitments to technologies attaining a high level of security. In the past few years, the advances in technologies related to the Internet of Things (IoT) has led to a boom in a broad spectrum of areas such as cloud computing, data mining, and information security. In particular, cloud service providers are to remove the burden of data management using cost-efficient data mining approaches. Hence, it is quite natural for both individuals and organisations to outsource their information data into a cloud server and allow this entity to process the data and run different data mining algorithms on the user's behalf. However, storing/processing sensitive data on untrusted cloud servers may raise serious privacy and security concerns for time-series related data in IoT applications.

One of the significant data processing tasks in IoT applications is anomaly detection (outlier detection).[1] Anomaly detection is the process of finding unusual patterns in data, and it has many applications in [117], intrusion detection [118], and fraud detection [119]. In the context of IoT devices, the anomaly detection can be used to remotely detect the malicious behaviours of IoT sensors, which are compromised by attackers [120]. The generated data is incremental/temporal (time-series data) and the volume of data to be analysed is effectively large and unbounded [117, 121]. Hence, an anomaly detection algorithm in this

[1] We use these two terms interchangeably in this chapter.

setting should be efficient in terms of computational costs and effective in terms of detection accuracy. While encryption can be used to address data privacy issues, it prevents the server from mining/processing the encrypted data unconditionally. In this chapter, we shall propose a new mechanism "Privacy-Preserving Outlier Detection (PPOD)" that addresses the problem of mining on encrypted data efficiently and effectively. Moreover, in order to make the process of anomaly detection more effective, we aim to consider the temporal relationships in time-series by leveraging the ideas in autoregression forecasting models in the context of uni/multivariate time-series. These models can detect deviation based anomalies by considering temporal relationships of measurements in time-series [122]. The privacy-preserving anomaly detection is a significant area of research and none of the state-of-the-art techniques has addressed it in the presence of temporal data.

Our system architecture comprises of a user (Gateway) and two semi-honest but non-colluding servers in charge of performing secure outlier detection (See Fig. 4.2). A PPOD for an incremental dataset D contains four algorithms: (1) Data Preprocessing: to generate an encrypted incremental dataset ED and distribute shares of received data points to servers. (2) Initialisation: to apply forms of secure multiparty computations to model the outliers of ED. This phase outputs the initial list of k-distances. (3) Query: to run a data mining algorithm on servers and detect anomalies associated with ED in a privacy-assured form. (4) Update: to take into account the newly arrived data points, compute their k-distances, and decide if they are anomaly or not. Hence, the specific contributions of this work are as follows:

- design a PPOD scheme based on well-known cryptographic protocols/primitives such as additive secret sharing and Yao's garbled circuit and efficient data mining anomaly detectors such as k-nearest neighbours (kNN) suitable for current IoT cloud services. We also prove that our PPOD scheme is secure with a given leakage function in a hybrid model, where parties are given access to the trusted party computing the ideal function of oblivious transfer (OT) [123].
- to handle incremental datasets efficiently, our PPOD incorporates the sliding window model and adapts proper plaintext outlier detection algorithms [124, 125] for streams for efficient and secure outlier detection.
- implement such a construction using computer simulations and analyse its accuracy and efficiency on incremental datasets for different system parameters. Our evaluations on a real-world dataset with 4200 16-dimensional data points show that PPOD has a practical performance: it can answer outlier queries within 217 ms and take 9 s to update the outlier model after receiving a new data point.

Organisation. The rest of this chapter is structured as follows. We discuss related work in Section 4.2. In Section 4.3, we introduce the background knowledge of distance-based outlier detection algorithms and the needed cryptographic primitives. Then, we describe

the system overview and its threat model in Section 4.4. A detailed construction of cryptographic modules and protocols is presented in Section 4.5. In Section 4.6, we briefly discuss the security of PPOD. Next, we describe our prototype implementation and evaluation results in Section 4.7. We give a conclusion in Section 4.8.

4.2 Related Work

Privacy-preserving outlier detection. The research in privacy-preserving outlier detection has two main streams, i.e., differential privacy-based approaches [126–128] and cryptographic-based (secure computation-based) approaches [129–131]. The differential privacy-based approaches rely on the data perturbation technique to add noise to protect the inputs from the multi-party [126]. To address the collusion issue in [126], Random Multiparty Perturbation technique [128] is proposed to allow each party to use a unique and different perturbation matrix to randomise their data. A recent differential privacy-based work [127] leverages a relaxed version of differential privacy to process the data in data streams. However, the differential privacy-based approaches lead to an accuracy loss in practice, while our PPOD does not degrade the accuracy comparing to the outlier detection algorithm for unencrypted datasets. The secure computation-based approaches are devised via Yao's garbled circuit [131], Homomorphic Encryption [129] and the hybrid approach like in this chapter [130]. Note that the above approaches are designed for the multi-party setting, i.e., each party has its private input, which are not suitable in the application scenario of this chapter (outsourced outlier detection).

Distance-based Outlier detection for incremental datasets (data streams). A large number of outlier detection algorithms (e.g. [124, 125, 132]) are proposed to support efficient outlier detection over the incremental datasets (or data streams). However, all the above algorithms only can process the data in an unencrypted form. Furthermore, these algorithms involve range queries which has multiple dedicated attacks for its encrypted version [110, 133] recently.

4.3 Preliminaries

4.3.1 Distance-based Outlier Detection

We briefly review the formal definitions of distance-based outlier detection. A more detailed introduction can be found in [134].

Distance-based outlier detection aims to detect an abnormal data point (a.k.a., outliers) via a distance measure between the target point and other points in a given dataset. In particular, a neighbour of an n-dimensional data point $\mathbf{p} = (p_1, ..., p_n)$ in the distance-based approach is defined as follows.

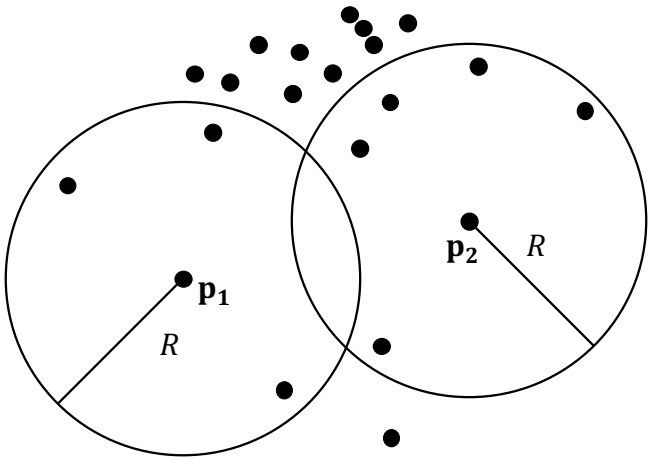


Figure 4.1: An example of the distance-based outlier.

Definition 3 (Neighbour)**.** *Given a distance threshold $R > 0$, a data point $\mathbf{q}$ is a neighbour of the target point $\mathbf{p}$ if the distance $\mathtt{d}(\mathbf{p}, \mathbf{q})$ between them is not greater than R, where $\mathtt{d}(\cdot, \cdot)$ is a distance measurement function.*

In the distance-based outlier detection approach, normal data points are assumed to have a dense neighbourhood while outliers are far apart from their neighbours (i.e., have a sparse neighbourhood). Therefore, the distance-based approach utilises the number of neighbours to detect outliers in a dataset.

Definition 4 (Distance-based Outlier)**.** *Given a dataset D and a positive integer count threshold k, a data point $\mathbf{p}$ is a distance-based* **outlier** *in D if it has less than k neighbours. Otherwise, it is called an* **inlier***.*

Fig. 4.1 depicts a scenario where the distance threshold R is fixed and $k = 5$. According to the above definition, a point $\mathbf{p}$ is an outlier if there are less than 5 points within the distance R from $\mathbf{p}$ (excluding $\mathbf{p}$ itself). Thus, $\mathbf{p_1}$ is an outlier while $\mathbf{p_2}$ is an inliner in the example.

4.3.2 Outlier Detection for Incremental Datasets

The distance-based outlier detection can be exploited to detect outliers in an incremental dataset too, where the dataset is continuously updated with the newly-presented data points. In this work, we adopt the so-called count-based window model as in the previous works [124, 125].

Definition 5 (Count-based Sliding Window)**.** *Given a window size W and a slide size S. Each window has a starting count C_{start} and an ending count $C_{end} = C_{start} + W$. The window 'slides' periodically after receiving a specific number of new data points, causing C_{start} and C_{end} to increase by S.*

In this sliding window model, each data point is associated with a counting number $C_{\mathbf{p}}$. A data point $\mathbf{p}$ is active if its counting number satisfies the following $C_{start} < C_{\mathbf{p}} \leq C_{end}$ (i.e., W active points).

To detect outliers over incremental datasets, a naive solution is to re-compute the neighbours for all active points when the window slides, which can be computationally expensive. Thus, recent studies devised incremental algorithms: during the update, only the data points which have at least one added/expired neighbour will be updated. In particular, those algorithms [124, 125] involve two steps:

- **Expired slide processing:** Data points in the expired slide are removed from the outlier set $\mathbf{O}$ and data point set $\mathbf{P}$. However, the expired point can still resident in the neighbour list of active points [124].
- **New slide processing**: For each new data point $\mathbf{p}'$, the algorithm computes its neighbourhood information to determine whether $\mathbf{p}'$ is an outlier or not (# of neighbours of $\mathbf{p}' \geq k$). Then, for each neighbour point $\mathbf{p}$ of $\mathbf{p}'$, the neighbour information will be updated regarding the newly-added distance (if $\mathsf{d}(\mathbf{p}, \mathbf{p}') \leq R$, then $\mathbf{p}$ has one new neighbour). Finally, the algorithm rechecks $\mathbf{p}$ to decide its outlier status according to the new neighbourhood information of $\mathbf{p}$.

Note that PPOD follows the above two steps to update the outlier model securely. Thus, it will not incur accuracy loss compared to the algorithms for plaintext data.

4.3.3 Secure Computation

We briefly review the secure computation technologies used in this chapter. Furthermore, we introduce the secure conversion method, which helps to mix efficient, secure protocols for different computations (e.g., addition, multiplication and sorting) together to support complex computations that involved in our secure outlier detection protocol efficiently. Readers can find a more detailed introduction in [31, 101].

Additive sharing and multiplication triplets. To additively share ($\mathsf{Shr}^{\mathsf{A}}(\cdot)$) an ℓ-bit integer a between two parties P_0 and P_1, the client generates $a_0 \in \mathbb{Z}_{2^l}$ uniformly at random and computes $a_1 = a - a_0 \mod 2^l$. The first party's share is denoted by $\langle a \rangle_0^A = a_0$ and the second party's is $\langle a \rangle_1^A = a_1$, the modulo operation is omitted in the description later. To reconstruct ($\mathsf{Rec}^{\mathsf{A}}(\cdot,\cdot)$) a shared value $\langle a \rangle^A$, each party sends its share to the client who computes $\langle a \rangle_0^A + \langle a \rangle_1^A$. Given two shared values $\langle a \rangle^A$ and $\langle b \rangle^A$, Addition ($\mathsf{Add}^{\mathsf{A}}(\cdot,\cdot)$) is easily performed non-interactively. In detail, P_i locally computes $\langle c \rangle_i^A = \langle a \rangle_i^A + \langle b \rangle_i^A \mod 2^l, i \in \{0,1\}$, which also can be denoted by $\langle c \rangle^A = \langle a \rangle^A + \langle b \rangle^A$.

To multiply ($\mathsf{Mul}^{\mathsf{A}}(\cdot,\cdot)$) two shared values $\langle a \rangle^A$ and $\langle b \rangle^A$, we leverage Beaver's multiplication triplets technique [90]. Assuming that the two parties have already precomputed and shared $\langle x \rangle^A$, $\langle y \rangle^A$ and $\langle z \rangle^A$, where x, y are uniformly random values in $\mathbb{Z}_{2^l}$, and $z = x \cdot y \mod 2^l$. Then, P_i computes $\langle e \rangle_i^A = \langle a \rangle_i^A - \langle x \rangle_i^A$ and $\langle f \rangle_i^A = \langle b \rangle_i^A - \langle y \rangle_i^A$. Both parties run $\mathsf{Rec}^{\mathsf{A}}(\langle e \rangle_0^A, \langle e \rangle_1^A)$ and $\mathsf{Rec}^{\mathsf{A}}(\langle f \rangle_0^A, \langle f \rangle_1^A)$ to get e and f, and P_i lets $\langle c \rangle_i^A = i \cdot e \cdot f + f \cdot \langle x \rangle_i^A + e \cdot \langle y \rangle_i^A + \langle z \rangle_i^A, i \in \{0,1\}$.

Garbled circuit and Yao's sharing. Yao's garbled circuit (GC) is first introduced in [91], and its security model has been formalised in [92]. GC is a generic tool to support secure two-party computation. The protocol is run between a "garbler" with a private input x and an "evaluator" with its private input y. The above two parties wish to securely evaluate a function $f(x, y)$. At the end of the protocol, both parties learn the value of $z = f(x, y)$, but no party learns more than what is revealed from this output value.

In the rest of this chapter and without loss of generality, we assume that P_0 is the garbler and P_1 is the evaluator. GC can also be considered as a protocol which takes as inputs the Yao's shares and produces the Yao's shares of outputs. In particular, the Yao's shares of 1-bit value $a \in \{0, 1\}$ is denoted as $\langle a \rangle_0^Y = \{K_0, K_1\}$ and $\langle a \rangle_1^Y = K_a$, where K_0, K_1 are the labels representing 0 and 1, respectively. The garbler runs a garbling algorithm $\mathcal{GC}$ to generate the garbled circuit and its encoded inputs in the form of Yao's shares. Then, the garbler sends the Yao's shares corresponding to its input to the evaluator. Meanwhile, the evaluator runs an oblivious transfer (OT) [93] protocol with the garbler to acquire the Yao's shares corresponding to its input. Then, the evaluator uses the received shares to evaluate the generated circuit and gets the output shares (other labels).

Conversion. Secure computations based on the above two schemes can be combined by converting one representation of intermediate values to the other [31]. Additive shares can be switched to Yao's shares ($\mathsf{A2Y}(\cdot)$) efficiently. To be more precise, two parties share their additive shares $a_0 = \langle a \rangle_0^A$, $a_1 = \langle a \rangle_1^A$ in a bitwise fashion via Yao's sharing. The evaluator then receives $\langle a_0 \rangle^Y$ and $\langle a_1 \rangle^Y$ and evaluates the circuit $\langle a_0 \rangle^Y + \langle a_1 \rangle^Y$ to get the label of a. Similarly, Yao's shares of a can be converted to additive shares using a subtraction circuit ($\mathsf{Y2A}(\cdot)$). In specific, the garbler chooses a random value $a_0 \in \mathbb{Z}_{2^l}$ as $\langle a \rangle_0^A$ and gives the Yao's share of a_0 to the evaluator, who evaluates the subtraction circuit $\langle a_1 \rangle^Y = \langle a \rangle^Y - \langle a_0 \rangle^Y$. The evaluator can recover a_1 locally and set it as $\langle a \rangle_1^A$.

4.4 System Overview

4.4.1 System Architecture

Fig. 4.2 shows the system architecture of the PPOD system. There are two entities in the PPOD system: the private gateway connected with data acquisition units (DAUs) and the server with the outlier detection service in an untrusted cloud. Note that this setting reflects the system model of many industrial corporations such as AgentVi [135] and HoneyWell [136], who provide data collection facilities and anomaly detection services while outsourcing the computation part of the service to the cloud service provider. In addition, popular cloud providers start to offer the incremental anomaly detection services in their dedicated data mining platform, e.g., Amazon Kinesis [137] and Azure Machine Learning Studio [138]. Our PPOD system aims to protect the confidentiality of the outsourced data in such a trend of using the data analysis cloud platform.

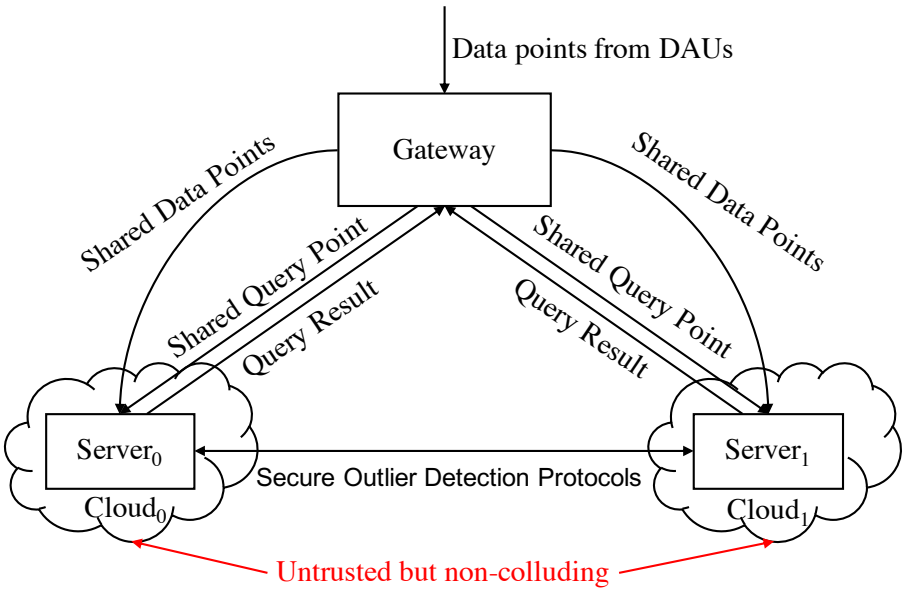


Figure 4.2: System overview.

Our system flow involves four phases: (1) *Data Preprocessing*: For each new data point from DAUs, the gateway preprocesses this point to meet the input requirement of the additive sharing scheme and shares it between two untrusted but non-colluding cloud servers S_0 and S_1. (2) *Initialisation*: During this phase, the servers execute secure computation protocols to compute k-nearest neighbours of each point and to determine the outlier list based on the k-distance (i.e., the distance between the data point and its k-th nearest neighbour). Additionally, each server stores the computed k-nearest neighbours list and k-distances as a reference for the update phase. (3) *Query*: The user of the PPOD system can submit a query point to the gateway to check whether the point is an outlier or not regarding the current outlier model. The query point is also preprocessed and shared by the gateway, and later the server leverages the share to measure the distance between outliers and the query point. The query point is an outlier in the current model if the computed distance is not greater than an outlier threshold, which is set by the system user. (4) *Update*: For each new data point, the server follows the same procedure as in the initialisation phase to find the k-nearest neighbours of new point and to find the new outlier based on the distance metric ($\mathsf{d}(\cdot,\cdot), R$ and k). Moreover, the server also updates the k-nearest neighbours' information of the new-coming data points. In this stage, the server combines the pre-computed information and new distance information to update the k-nearest neighbours list for these affected points. At last, the server refers to the new k-nearest neighbours information to decide the status (i.e., outlier or inlier) of these points.

Our system considers a server-aid computation scenario where the internal gateway distributes the computation tasks to two untrusted but non-colluding cloud servers. Such a two-server approach has been formalised [96] and widely utilised in the literature [28, 32, 97, 139] to protect the data confidentiality in the outsourcing computation context.

Table 4.1: Notations for the outlier detection algorithms.

Notation	Meaning
$\mathbf{p}$	A data point with n-dimensional coordinates $(p_1, ..., p_n)$
$\mathtt{d}(\mathbf{p}, \mathbf{q})$	The distance between data point $\mathbf{p}$ and $\mathbf{q}$
$\mathbf{p}.id$	The identity of $\mathbf{p}$
$\mathbf{p}.D$	The unordered list of the k-nearest neighbours of $\mathbf{p}$ in the form of $\{\mathbf{q_i}.id, \mathtt{d}(\mathbf{p}, \mathbf{q_i})\}_{i=1}^{k}$
$\mathbf{p}.D^k$	The distance between $\mathbf{p}$ and its k-th nearest neighbour
$\langle \mathbf{p} \rangle$	$\mathbf{p}$'s secret shares (The coordinates, D, D^k and $C_\mathbf{p}$ are stored as secret shares)
$\mathbf{P}$	A data point set
$\langle \mathbf{P} \rangle$	The data point set keeps the secret shares of data points

4.4.2 Threat Model

In this work, we assume that the gateway and the attached DAUs are maintained by a data analytics service provider, which is a trusted party. Meanwhile, we consider that the two servers belong to two different semi-honest but non-colluding parties (e.g., two cloud providers). They will follow our protocol honestly, but they are interested in learning the underlying private information, which, in our case, are the coordinates of data points. In the rest of the chapter, we use $\mathcal{A}_i$ to denote the adversary who compromises S_i. In our security model, we require that the $\mathcal{A}_i$ is capable of seeing the protocol messages in S_i and tries to infer the user's private information. However, $\mathcal{A}_i$ should not learn any information about its counter-party's data beyond the protocol output. This model aims to protect the confidentiality of data points when the data analytics service providers outsource the computation task to the public cloud.

4.5 PPOD Protocol Construction

We now explain the construction of PPOD in more details. The notations we used for the algorithms are summarised in Table 4.1.

4.5.1 Cryptographic Modules

In order to explain the design clearly, we break the protocol into common used cryptographic modules implemented by the cryptographic primitives (see Section 4.3 for details). In this section, we discuss the design and implementation of these cryptographic modules.

Distance measurement

In this work, we leverage the squared Euclidean distance to measure the distance between two data points. Note that such a distance metric is commonly used in outlier detection algorithms [140, 141] as it requires less computations (i.e., only addition and multipli-

Algorithm 4 k-nearest neighbours

Input: Shared point set $\langle \mathbf{Q} \rangle$, Shared point $\langle \mathbf{p} \rangle$, Parameter k
Output: Yao's shares of the unordered kNN list L

function kNN($\langle \mathbf{Q} \rangle$, $\langle \mathbf{p} \rangle$, k)
    $S \leftarrow \{\}$
    for each $\langle \mathbf{q} \rangle \in \langle \mathbf{Q} \rangle$ **do**
        $\langle d \rangle^A = \langle \mathsf{d}(\langle \mathbf{p} \rangle, \langle \mathbf{q} \rangle) \rangle^A$
        put $\{\langle \mathbf{q} \rangle.id, \langle d \rangle^A\}$ into S
    end for
    return SORTSHUFFLE(S, k)
end function

Algorithm 5 k-distance

Input: Yao's shares of the kNN list L
Output: Yao's shares of the maximum value in L

function kDIST(L)
    return MAX(L)
end function

cation). However, directly computing $\Sigma_{1 \le i \le n}(p_i - q_i)^2$ is not applicable in our system due to the non-negative input restriction of additive sharing scheme, i.e., if there exists an $1 \le i \le n$ such that $p_i - q_i < 0$, the square operation will produce the additive shares of an undesired result $(2^l - (p_i - q_i))^2$. A naive solution for this issue is to make a comparison and swap before computing $p_i - q_i$ to ensure that $p_i - q_i \ge 0$, yet it requires additional steps to convert the additive shares to Yao's shares (for comparison) and convert it back (for computation), which leads to extra computation and communication cost. Therefore, the distance measurement function $\mathsf{d}(\cdot, \cdot)$ in our system is defined as $\mathsf{d}(\mathbf{p}, \mathbf{q}) \triangleq \Sigma_{1 \le i \le n}((p_i)^2 + (q_i)^2 - 2 \cdot p_i \cdot q_i)$, which can avoid all negative results as well as the expensive comparison and swap. Noted that this metric also works when data points are secretly shared. In particular, the two servers can run arithmetic operations to compute their shares as

$$\langle \mathsf{d}(\langle \mathbf{p} \rangle, \langle \mathbf{q} \rangle) \rangle^A = \Sigma_{1 \le i \le n}((\langle p_i \rangle^A)^2 + (\langle q_i \rangle^A)^2 - 2 \cdot \langle p_i \rangle^A \cdot \langle q_i \rangle^A),$$

independently.

k-nearest neighbours (kNN) and k-distance

To detect outliers in a given set of data points, PPOD employs the distance metric in Section 4.5.1 to compute the share of distances and utilises these shares to compute the k-nearest neighbours (kNN) and k-distance of each data point. Then, it compares the k-distance with the parameter R; if the k-distance is greater than R, the data point is an outlier in the current model. Note that this approach can detect the outliers defined in Section 4.3.1: the k-distance of a data point is greater than R is equivalent to the point has less than k neighbours within the given range R, and it is an outlier.

The simplest way to securely compute the kNN list and k-distance is to retrieve those

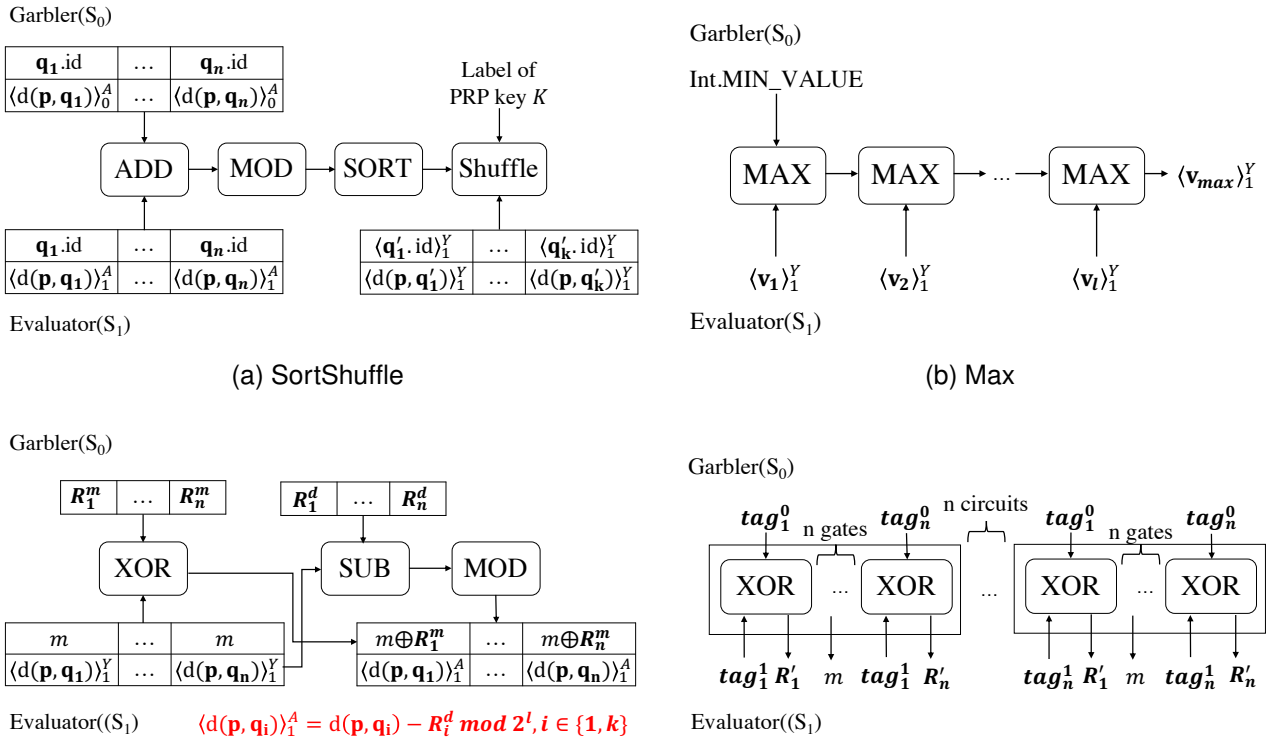


Figure 4.3: The circuit structure of cryptographic sub-modules.

information from a sorted point list after evaluating a sorting circuit over the share of distances. However, the above solution still has two security issues. First, sorting reveals the order of data points, while some recent works [110, 133] demonstrated that it is possible to precisely reconstruct the underling values (i.e., distances) if an adversary knows the rank and some auxiliary information. Furthermore, different kNN lists may include the same data point, which means that the adversary can compare the identities/distance shares in different kNN lists to learn extra information about common neighbours for different data points. Thus, to protect the privacy of data points, the procedures for kNN and k-distance evaluation should not reveal the order as well as the identities/distance shares.

Algorithm 4 and 5 outline the overall process of computing kNN and k-distance for a point $\mathbf{p}$. These algorithms employ two cryptographic sub-modules to implement the secure sorting (SORTSHUFFLE) and comparison (MAX). Besides, we provide two other cryptographic modules to preprocess (RANDOMISE) and post-process (DERANDOMISE) the kNN list and k-distance to hide the repeat patterns. Fig. 4.3 summarizes these cryptographic sub-modules.

SORTSHUFFLE. Fig. 4.3a shows the structure of the secure sorting module for kNN computation, our system follows the standard procedure (see Section 4.3.3 for details) to evaluate the circuit in Fig. 4.3a and receives an unordered kNN list as the result. The secure sorting module inputs the shares of distance to an A2Y$(\cdot)$ function implemented via an efficient scheme in [31] to convert the additive shares to Yao's shares. It then adopts the sorting circuit based

on sorting network [102] to sort the Yao's shares from the A2Y$(\cdot)$ function. In addition, the garbler concatenates the sorting circuit with a shuffling circuit based on a pseudorandom permutation (
acrshortprp) [142], and the evaluator supplies a random key K to disrupt the order of the kNN list and remove the remaining points. Finally, the SORTSHUFFLE circuit outputs the Yao's shares of the unordered kNN list, which ensures that the order does not reveal in the sorting procedure.

MAX. To retrieve the k-distance from an unordered kNN list, the KDIST algorithm employs the MAX circuit shown in Fig. 4.3b. The circuit takes as inputs a list of Yao's shares (e.g., Yao's shares of distances in the kNN list), and it consists of a chain of MAX gates to compute the maximum value (e.g., the k-distance in KDIST) of the given inputs. In order to protect the underlying values (the distances), the output of the MAX circuit is also in the form of Yao's shares.

RANDOMISE/DERANDOMISE. Each entity in the kNN list comprises two data types, i.e., the values that indicate the computed distance, and the identities which assist the server to work consistently. To hide the repeat patterns after kNN and k-distance evaluations, we should protect the above two data types when the server converts Yao's shares back to the additive shares for the storage purpose. We design a RANDOMISE function (see Fig. 4.3c) to achieve the above goal: To protect the distance value, the garbler generates a new random value and garbles the circuit for Y2A$(\cdot)$ to re-share the distance as the additive shares; To hide the identity on servers, we introduce a flag independent from the data point id to aid the server to find the position of corresponding shares in its counter-party before starting the computations. More specifically, the evaluator selects a magic number m to de-identify its local points and leverages random numbers R^m generated by the gabler to mask m via XOR operations. After circuit evaluation, the garbler stores the generated random vectors as its local data point shares and the evaluator takes the output of the circuit as the new data point shares.

The DERANDOMISE function is used to pair the randomised shares between two servers. As shown in Fig. 4.3d, for a randomised list with n elements, the DERANDOMISE function generates n^2 XOR gates revealing the "paired" positions, i.e., the position where the XOR gate returns m. The server then exploits its local shares to run the following secure protocols according to the revealed position. After computing, two servers run RANDOMISE function again to invalidate the revealed patterns.

4.5.2 Data Preprocessing

Overview. Input preprocessing runs for all data points receiving from some DAUs (e.g., sensors). As shown in Algorithm 6, the gateway performs a two-step preprocessing over the received data points before giving them to the server for secure outlier detection. The first step is to dissolve the input format mismatch between the client and the server. Namely,

Algorithm 6 Input Preprocessing Phase

Input: Data point set $\mathbf{P}$

1: **for** each $\mathbf{p} \in \mathbf{P}$ **do**
2: $\quad \widetilde{\mathbf{p}} \leftarrow \text{NORMALISE}(\mathbf{p})$
3: $\quad [\![\mathbf{p}]\!] \leftarrow \text{ROUNDING}(\widetilde{\mathbf{p}}, l_D)$
4: $\quad$ **for** i=1 to n **do**
5: $\quad\quad \langle [\![p_i]\!] \rangle^A \leftarrow \mathsf{Shr}^{\mathsf{A}}([\![p_i]\!])$
6: $\quad$ **end for**
7: $\quad$ send $\langle \mathbf{p} \rangle_0 = \{\mathbf{p}.id, \langle [\![\mathbf{p}]\!] \rangle_0^A\}$ to P_0 and $\langle \mathbf{p} \rangle_1 = \{\mathbf{p}.id, \langle [\![\mathbf{p}]\!] \rangle_1^A\}$ to P_1
8: **end for**

the data point from DAUs consists of fractional numbers and may also include negative numbers, while the additive sharing scheme in our protocol only works over non-negative integers. Thus, the gateway should pre-process these received coordinates via normalisation and rounding to meet the input requirement of cryptographic primitives before it shares the data to servers After preprocessing, the gateway generates the additive shares for these adjusted data points and distributes the generated shares to two cloud servers. The detailed construction of the above two preprocessing steps are discussed below:

1) NORMALISE: This function runs to eliminate the negative numbers in coordinates. For each coordinate, we assume that the maximum/minimum values are fixed at the beginning of data collection, as it is possible for the gateway to know these parameters referring to the hardware specification of DAUs. Therefore, the gateway can store the maximum/minimum values (d_i^{max} and d_i^{min}) for each $i \in [1, n]$. When the gateway receives a data point $\mathbf{p}$, it extracts its coordinate value p_i and computes $(p_i - d_i^{min}) \cdot (d_i^{max} - d_i^{min})^{-1}$, which outputs a value $\widetilde{p_i} \in [0, 1]$ as the corresponding normalised coordinate value for p_i.

2) ROUNDING: After normalisation, the coordinates of a normalised point $\widetilde{\mathbf{p}}$ have only positive fractional numbers in $[0, 1]$. To handle fractional coordinate values, we introduce a rounding factor l_D to scale up the fractional number into an integer $[\![p_i]\!] = \lfloor \widetilde{p_i} \cdot 2^{l_D} \rfloor$, while preserving l_D bits in the fractional part of the original number. This is a common strategy adopted in several prior works [32, 143]. As illustrated in the evaluation, the accuracy of the outlier model is not affected under a deliberately selected l_D.

Discussion. The input preprocessing should be applied to all new arrival data points on the gateway before the gateway gives it to the server. Nevertheless, this will not incur a heavy workload on the gateway and lead to a noticeable delay to the system performance for the following two reasons: First, the input preprocessing phase can run independently for each data point. Thus, the gateway can leverage parallel processing to handle the received data points in a batch, which can highly improve the preprocessing process. Besides, the gateway does not involve any computation task other than input preprocessing under the two-server setting. The main computation of the outlier detection algorithm is located on the server.

Algorithm 7 Initialisation Phase

Input: Shared data point set $\langle \mathbf{P} \rangle$, Parameter k, $\langle R \rangle^A$
Output: Shared outlier List $\langle O \rangle$

1: **for** each $\langle \mathbf{p} \rangle \in \langle \mathbf{P} \rangle$ **do**
2: $\langle temp \rangle^Y \leftarrow \text{kNN}(\langle \mathbf{P} \rangle, \langle \mathbf{p} \rangle, k)$
3: $\langle \mathbf{p} \rangle.D \leftarrow \text{RANDOMISE}(\langle temp \rangle^Y)$
4: $\langle dist \rangle^Y \leftarrow \text{kDIST}(\langle temp \rangle^Y)$
5: $\langle \mathbf{p} \rangle.D^k \leftarrow \text{Y2A}(\langle dist \rangle^Y)$
6: **if** $\langle dist \rangle^Y > \text{A2Y}(\langle R \rangle^A)$ **then**
7: Add $\langle \mathbf{p} \rangle$ into $\langle O \rangle$
8: **end if**
9: **end for**

4.5.3 Initialisation

Overview. For the first batch of the preprocessed data points (their additive shares) from the gateway, the server invokes the initialisation phase to create the outlier model. To realise this phase, our system adapts the D_n^k outlier detection algorithm from [141] as it can be implemented via arithmetic operations and sorting only, which perfectly suits the secure computation model we used. In particular, the server uses the received data points and some pre-set parameters to execute the algorithm and gets the k-nearest neighbours of each data point as well as the corresponding k-distance (denoted as D^k). Consequently, it compares D^k with the distance threshold R to find the outliers (i.e., if D^k is greater than R, the point is an outlier). The server also stores the computed information, i.e., k-nearest neighbours, D^k values and the distance threshold R (as additive shares) to support the update phase (see Section 4.5.5). The completed procedure of the D_n^k-based privacy-preserving outlier detection is shown in Algorithm 7.

Discussion. The initialisation is a time-consuming procedure, as it follows a nested loop strategy, i.e., it traverses each pair of data points, which infers an $\mathrm{O}(\beta^2)$ computational complexity, where β is the number of data points in the batch. Despite the relatively higher computation cost, we argue that this phase only needs to run once for the entire outlier modelling process, and the model can be updated within $\mathrm{O}(S \cdot W)$ (see Section 4.5.5 for details).

In terms of security, the algorithm with the nested loop strategy executes the same sequence of operations over all data points. Hence, the initialisation phase is a data-oblivious process under the two-party secure computation context, that is, the initialisation phase only reveals the information about outliers. Conversely, the other information, such as the coordinates, kNN list, and the memory access pattern during the outlier detection process, can be kept in secret.

Algorithm 8 Query Phase

Input: Shared query point $\langle \mathbf{q} \rangle$, Parameter $\langle \epsilon \rangle^A$
Output: An assertion (True or False)
1: $D \leftarrow \{\}$
2: **for** each $\langle \mathbf{o} \rangle \in \langle O \rangle$ **do**
3: Compute $\langle d \rangle^A \leftarrow \langle \mathsf{d}(\langle \mathbf{q} \rangle, \langle \mathbf{o} \rangle) \rangle^A$ and put $\langle d \rangle^A$ into D
4: **end for**
5: $C \leftarrow \{\}$
6: **for** each $\langle d \rangle^A \in D$ **do**
7: Compute $\langle r \rangle^Y \leftarrow \mathsf{A2Y}(\langle d \rangle^A) \leq \mathsf{A2Y}(\langle \epsilon \rangle^A)$ and put $\langle r \rangle^Y$ into C
8: **end for**
9: **return** $\textsc{Or}(C)$

4.5.4 Outlier Query

Overview. The system user can issue a data point query to check whether the data is an outlier or not by referring to the outlier model on the server side. The query consists of a preprocessed data point and an outlier threshold. Once the server receives a query, it evaluates the distance between the query point and outliers using the given additive shares. Consequently, it utilises the garbled circuit to compare the computed distances and the threshold and produces the final assertion without revealing any sub-result (i.e., which distance is smaller than the threshold). Algorithm 8 outlines the query process on each server. Next, we present the detailed construction of the assertion function.

Assertion function. In the assertion function , the server makes a comparison between all distances and the given outlier threshold (line 5 – 8 in Algorithm 8). If one of those distances is not greater than the threshold, the query point is considered as an outlier, so the server returns 'True'; otherwise, it returns 'False'. Finally, the output assertion is generated via an OR gate, which mixes each pair of distance comparison as the final output.

Discussion. Query phase is an efficient stage, as it only performs arithmetic operations and comparison with the known outlier list. Therefore, its computational complexity is bounded by the size of the outlier list **O**, which is much smaller than the other phases. In addition, Algorithm 8 is also a data-oblivious algorithm because it loops for each outlier to produce the result. During this process, the server only knows the final assertion, but not any intermediate result (e.g., each pair of comparison result) and the input.

4.5.5 Model Update

Overview. In the model update phase, each server receives a new batch of the preprocessed data points and computes a new outlier model, which takes these new data points into consideration. To ensure the efficiency of this phase, the update protocol for PPOD uses the sliding window model and maintains a list of active points and only recomputes/reports the outliers for the active points. Also, the update algorithm only updates the data points

Algorithm 9 Update Phase

Input: Active point set $\langle \mathbf{P_a} \rangle$, Incoming point set $\langle \mathbf{Q} \rangle$, Parameter $\langle R \rangle^A$
Output: Shared outlier List $\langle \mathbf{O} \rangle$

1: Remove expired points from $\langle \mathbf{P_a} \rangle$ and $\langle \mathbf{O} \rangle$
2: **for** each $\langle \mathbf{q} \rangle \in \langle \mathbf{Q} \rangle$ **do**
3: Compute $\langle \mathbf{temp} \rangle^Y \leftarrow \text{kNN}(\langle \mathbf{P_a} \rangle, \langle \mathbf{q} \rangle, k)$
4: $\langle dist \rangle^Y \leftarrow \text{kDIST}(\langle \mathbf{temp} \rangle^Y)$
5: $\langle \mathbf{q} \rangle.D^k \leftarrow \text{Y2A}(\langle dist \rangle^Y)$
6: **if** $\langle dist \rangle^Y > \text{A2Y}(\langle R \rangle^A)$ **then**
7: Add $\langle \mathbf{q} \rangle$ to $\langle \mathbf{O} \rangle$
8: **end if**
9: Add $\langle \mathbf{q} \rangle$ to $\langle \mathbf{P_a} \rangle$
10: **for** each $\langle \mathbf{t} \rangle^Y \in \langle \mathbf{temp} \rangle^Y$ **do**
11: Recover $\mathbf{t}.id$ from $\langle \mathbf{t}.id \rangle^Y$
12: Retrieve $\langle \mathbf{a} \rangle$ from $\langle \mathbf{P_a} \rangle$ based on $\mathbf{t}.id$
13: **if** $\langle \mathbf{a} \rangle \in \langle \mathbf{O} \rangle$ **then**
14: $\langle \mathbf{a} \rangle.D \leftarrow \text{DERANDOMISE}(\langle \mathbf{a} \rangle.D)$
15: $\langle \mathbf{a} \rangle.D.\text{ADD}(\{\mathbf{q}.id, \text{Y2A}(\langle \mathbf{t}.d \rangle^Y)\}))$
16: $\langle temp_a \rangle^Y \leftarrow \text{SORTSHUFFLE}(\langle \mathbf{a} \rangle.D, k)$
17: $\langle \mathbf{a} \rangle.D \leftarrow \text{RANDOMISE}(\langle temp_a \rangle^Y)$
18: $\langle \mathbf{a} \rangle.D^k \leftarrow \text{Y2A}(\text{kDIST}(\langle temp_a \rangle^Y))$
19: **if** $\text{kDIST}(\langle temp_a \rangle^Y) \leq \text{A2Y}(\langle R \rangle^A)$ **then**
20: Remove $\langle \mathbf{a} \rangle$ from $\langle \mathbf{O} \rangle$
21: **end if**
22: **end if**
23: **end for**
24: **end for**

that affect by the added/expired data points, which is consistent to the incremental algorithms [124, 125] for the plaintext outlier detection scheme. In particular, the update protocol removes the expired points from the active point set $\mathbf{P_a}$ and the outlier list $\mathbf{O}$. Then, it computes the kNN and k-distance information for the new data point by utilising the remaining $\mathbf{P_a}$ and determines whether the new point is an outlier. Later, the protocol updates the points in $\mathbf{P_a}$ which are also in the kNN list of the new data point. The procedure of the update phase is given in Algorithm 9.

Discussion. The simplest solution is to run the initialisation protocol for the updated dataset. However, as mentioned in Section 4.5.3, the initialisation is an inefficient phase ($\mathrm{O}(|\mathbf{P}|^2)$, where $|\mathbf{P}|$ is the size of dataset). Compared to the naive approach in the above, the complexity of the proposed update approach is lower: For each new data point, the update phase only refers the active data points to compute the kNN list, which takes $\mathrm{O}(W)$, where W is the sliding window size, and $\mathrm{O}(k)$ to update the existing information. And the whole update procedure runs for the new points after sliding (add S new points), which indicates that the overall runtime complexity is $\mathrm{O}(S \cdot W)$

In terms of the security, the update approach does not guarantee the data-oblivious, because it retrieves the id of the kNN list when it updates for the existing data points (line

Algorithm 10 Ideal Function $\mathcal{F}_k$

Parameters: Client C and servers S_0, S_1.
Input: On input $\langle \mathbf{p} \rangle_i, i \in \{0,1\}$ from C, stores it locally.
kNN$_{id}$: On input the query point $\langle \mathbf{q} \rangle_i, i \in \{0,1\}$ and the shared point set $\langle \mathbf{P} \rangle_i$ from S_i, the functionality returns an unordered point list with id only.
kNN$_{dist}$: On input the query point $\langle \mathbf{q} \rangle_i, i \in \{0,1\}$ and the shared point set $\langle \mathbf{P} \rangle_i$ from S_i, the functionality returns an unordered list of the shared distances only.
kDist: On input the query point $\langle \mathbf{q} \rangle_i, i \in \{0,1\}$ and the shared point set $\langle \mathbf{P} \rangle_i$ from S_i, the functionality returns the shared distance between the query point and its k^{th} nearest neighbour.
Update: On input the kNN list $\langle \mathbf{p} \rangle_i.D$ of point p and a shared points $\langle \mathbf{q} \rangle_i$ with the shared distance from S_i, the functionality updates $\langle \mathbf{p} \rangle_i.D$ and returns an unordered list of the shared distances only.

11 – 12 in Algorithm 9). Nevertheless, we stress that this is the only additional leakage comparing with the other phases, and it enables a more efficient update phase.

4.6 Security Analysis

We give the security analysis following the classic paradigm of comparing the real-world execution of the protocol to an ideal-world execution where a trusted third party evaluates the functions on behalf of the involved parties. The only difference is that we consider an ideal world that the adversary is allowed to learn the kNN of a new arrival data point when adding it into the sliding window. Note that we leverage an OT-hybrid model where parties are given access to the trusted party computing the ideal function of OT. The following theorem shows that the PPOD protocol is secure with the given leakage function in this hybrid model. Thus, the PPOD protocol remains secure if the trusted party is replaced by the real OT.

To start with, we give a security analysis for the secure kNN and k-distance modules in Section 4.5.1, as our PPOD protocol highly depends on these modules.

Theorem 4. *Consider a protocol where clients distribute shares of data points among two servers who run our PPOD protocol from Section 4.5. In the* OT*-hybrid model, the protocol Π_k realises the ideal function $\mathcal{F}_k$ in Algorithm 10 in presence of semi-honest but non-colluding adversaries.*

Proof. We denote the secure kNN and k-distance protocols as Π_k, and our proof shows that Π_k securely realises the ideal functions $\mathcal{F}_k$ in Algorithm 10. As the adversary in our model only corrupts one server at most, and the view of two servers are slightly different (one garbler and one evaluator), we separately consider the scenario that the adversary $\mathcal{A}_i, i \in \{0,1\}$ corrupts S_i. For each $\mathcal{A}_i$, we describe how to construct a simulator $\mathbf{Sim}_i$ that simulates $\mathcal{A}_i$ in the ideal model. For two varieties of the kNN evaluations (i.e., kNN$_{id}$ and kNN$_{dist}$), the only difference between them is how they handle the output of SORTSHUFFLE. In particular, in kNN$_{id}$, $\mathbf{Sim}_0$ returns the identities from the trusted party to $\mathcal{A}_0$ and $\mathbf{Sim}_1$ should give

the simulated decoded information of identities to $\mathcal{A}_1$. On the other hand, in kNN$_{dist}$, both simulators are only required to return the random shares of distance to the adversary.

We claim that $\mathcal{A}_i$'s view in the real and ideal model is indistinguishable for the kNN evaluations: Since the security of the additive sharing scheme and multiplication triplets ensure the randomness of distance shares, and the protocol is a composition of a sequence of secure modules (SORTSHUFFLE, RANDOMISE). It follows from the modular composition theorem [104] that the adversaries' views are both identical. The kDist function is almost identical to the kNN functions except it connects the output of SORTSHUFFLE gate to a MAX gate to retrieve the maximum distance in kNN list. Therefore, we can follow the same path to show the security of the kDist function, i.e., the modular composition theorem is applied for SORTSHUFFLE gate, MAX gate, and Y2A gate to get the same view in real/ideal models. The update function only involves garbled circuit evaluation, and the security of the garbled circuit ensures that no adversary can learn the input (i.e., previous kNN list) from the output and the execution on the circuit. □

Algorithm 11 Ideal Function $\mathcal{F}_o$

Parameters: Client C and servers S_0, S_1.
Input: On input $\langle \mathbf{p} \rangle_i, i \in \{0, 1\}$ from C, stores it locally.
Initialise: On input the first batch of shared points $\langle \mathbf{P} \rangle_i, i \in \{0, 1\}$ from S_i, the functionality initialises the shared outlier list $\langle \mathbf{O} \rangle_i$.
Query: On input the shared query point $\langle \mathbf{q} \rangle$ from C, and the shared outlier list $\langle \mathbf{O} \rangle_i$ from S_i, the functionality returns 'True' or 'False' to indicate the query point is an outlier or not.
Update: On input the active shared points $\langle \mathbf{P_a} \rangle_i$ and the new batch of shared points $\langle \mathbf{Q} \rangle_i$ from S_i, the functionality updates the shared outlier list $\langle \mathbf{O} \rangle_i$. Besides, it returns the identities of kNN of new arrival data point $\langle \mathbf{q} \rangle_i \in \langle \mathbf{Q} \rangle_i$ sequentially.

We now provide the security proof of the PPOD protocol. The ideal function of our PPOD is given in Algorithm 11. The following theorem demonstrates the PPOD scheme is secure under the non-colluding semi-honest server model.

Theorem 5. *Consider a protocol where clients distribute shares of data points among two servers who run the PPOD protocol in Section 4.5. In the ($\mathcal{F}_k$, OT)-hybrid model, the PPOD protocol adopts the ideal function $\mathcal{F}_o$ with leakage consisting of k-nearest neighbours of new arrival data points in Algorithm 11 in semi-honest but non-colluding adversarial model.*

Proof. We follow the same setting to prove the security of the PPOD system. In the initialisation phase, $\mathbf{Sim}_i$ runs $\mathcal{A}_i$ and sends randomly generated shares in $\mathbb{Z}_{2^l}$ with identity as the shared points to $\mathcal{A}_i$. Besides, the computation and randomisation of kNN list and k-distance can be simulated by calling the ideal function $\mathcal{F}_k$.kNN$_{dist}$ and $\mathcal{F}_k$.kDist. Finally, $\mathbf{Sim}_0$ utilises a dummy circuit and simulates input labels and plays the role of the trusted server to send the detected outlier identities to simulate the view of $\mathcal{A}_0$. On the other hand, $\mathbf{Sim}_1$ relies on Π_k to get the comparison result between k-distance and threshold R and sends the simulated circuit with the same output to $\mathcal{A}_1$ as its view.

Now, we illustrate the security of PPOD in each phase, respectively: During initialisation, $\mathcal{A}_i$'s view in the real and ideal model is indistinguishable: $\mathbf{Sim}_i$ provides the random value as the shared points and simulates the garbled circuit via the output of $\mathcal{F}_k$ for the corresponding $\mathcal{A}_i$. Besides, it uses the ideal function $\mathcal{F}_o$ to return the result to $\mathcal{A}_i$.

For the query phase, the simulator leverages the random input to simulate the query points, and then, it can simulate the adversaries' view similarly as above. In particular, the distance shares is also a random number as it leverages the randomly generated multiplication triplets. Moreover, the simulator utilises the simulator of garble circuit to simulate the rest of the protocol and returns the assertion to the client. Therefore, the modular composition theorem also implies that the query protocol remains secure after combining the additive sharing scheme and the garbled circuit.

The update phase is almost identical to the initialisation phase, except that it additionally reveals the kNN list of new arrival data, and there is an extra round to update the information of these k-nearest neighbours. Specifically, the update phase requires to call $\mathcal{F}_k$.kNN$_{id}$ and updates the points with the returned id. As a result, the update phase is secure with one extra leakage, as it is the composition of the initialisation phase and the functionalities in $\mathcal{F}_k$.kNN$_{id}$ and $\mathcal{F}_k$.Update. As Π_k securely realises $\mathcal{F}_k$, the PPOD scheme also securely realises the $\mathcal{F}_o$ with the leakage of k-nearest neighbours of a new arrival data point in the ($\mathcal{F}_k$, OT)-hybrid model. □

4.7 Evaluation

Implementation. We implement our PPOD system in Java. To enable the efficient and secure two-party computation on the cloud server, we first implement the additive sharing scheme. The arithmetic operations in the additive sharing scheme are computed by several regular addition and multiplication operations with the modulo operation over Java primitive types. Note that the modulo operation implemented via Java primitive types (e.g. **long**, **int**) is much faster than the native modulo operation in Java **BigInteger** type (about 50x faster). For the oblivious transfer (OT) and garbled circuit protocol, we leverage FlexSC [113], which includes the implementation of extended OTs [93] and the optimised garbled circuit scheme. To improve the runtime performance of our prototype, PPOD system maintains a pool of pre-computed multiplication triplets, and it periodically refreshes it to avoid extra computation/communication cost on-the-fly.

Setup. The experiments are executed on two AWS EC2 c5.4xlarge instances running Ubuntu 18.04LTS. Each instance has 16 cores and 48 GB of memory. Besides, we create a c5.large instance (4 cores and 8GB memory) serving as the client (i.e., gateway) in the PPOD system. It preprocesses and distributes the dataset to the above two more powerful servers to execute the PPOD protocol. Our servers are connected with a 10Gb NIC. To evaluate the performance of PPOD, we use a real-world dataset from UCI [144], which contains $4,200$ records of 16-dimension.

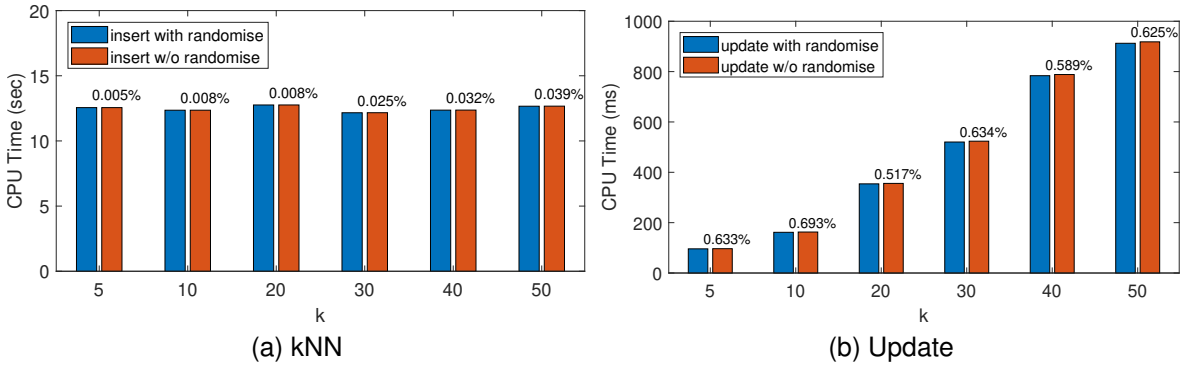


Figure 4.4: CPU time of the proposed secure kNN module when varying k. Numbers on top of the bars demonstrate the overhead ratio between red and blue bars.

Parameters. There are four parameters in our PPOD system: the window size W, the slide size S, the count threshold k and the distance threshold R. We evaluate the PPOD system under different W and k because they are the main factors affecting the performance of our PPOD. In particular, W determines the number of distance measurement functions to be executed as well as the input size of the SORTSHUFFLE circuit. On the other hand, k determines the size of RANDOMISE/DERANDOMISE and KDIST function, which are frequently used during the update phase. By default, we set $W = 400$, $S = 20$, $k = 50$ and $R = 25,000$ in our dataset. Unless specified otherwise, all the parameters take on their default values in the experiments.

In the rest of this section, we first benchmark the performance of the kNN module, and then we report the runtime performance of our PPOD.

4.7.1 Performance of the kNN module

CPU Time. Fig. 4.4 depicts the resulting CPU time of the secure kNN module in different phases. In particular, Fig. 4.4a shows the CPU time when adding a new point into the current model: despite the increasing of k, the CPU time of adding a new point is a constant (around 12s). This is because the kNN is executed during the initialisation phase and the update phase to process the new arrival points, and it involves the distance measurement computation and SORTSHUFFLE evaluation with the existing data points (380-400 points). Compared to the above two steps, the remaining steps, i.e., computing the k-distance and RANDOMISE with k inputs, can be done efficiently (less than 17 ms according to our evaluation).

The CPU time of updating (see Fig. 4.4b) an existing point is varying from 96 ms to 912 ms with the increasing of k. The update function of the kNN module only runs in the update phase to update the kNN list of the target point. The parameter k affects the runtime performance of the update function, since the parameter determines the size of kNN lists, and the server takes more time to evaluate a larger circuit to update if the size of kNN lists is larger. Finally, we examine the impact of the proposed RANDOMISE/DERANDOMISE

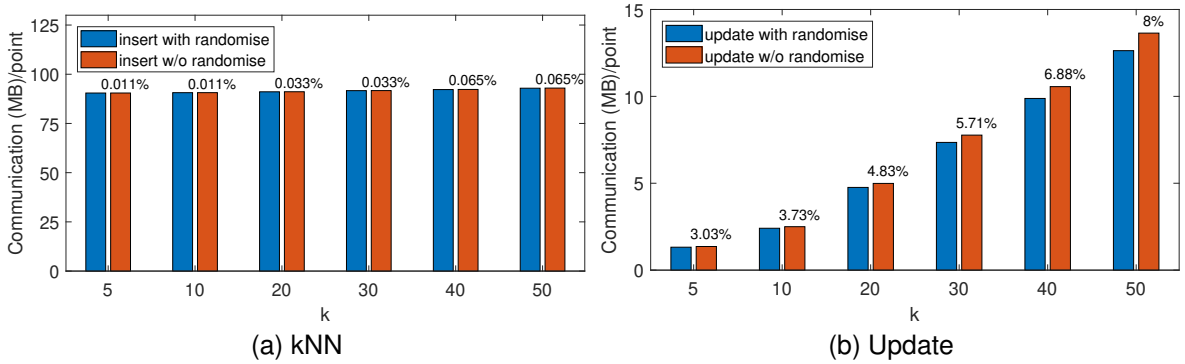


Figure 4.5: Communication overhead of the proposed secure kNN module with different k.Numbers on top of the bars demonstrate the overhead ratio between red and blue bars.

Table 4.2: Runtime performance of the PPOD system under default parameters.

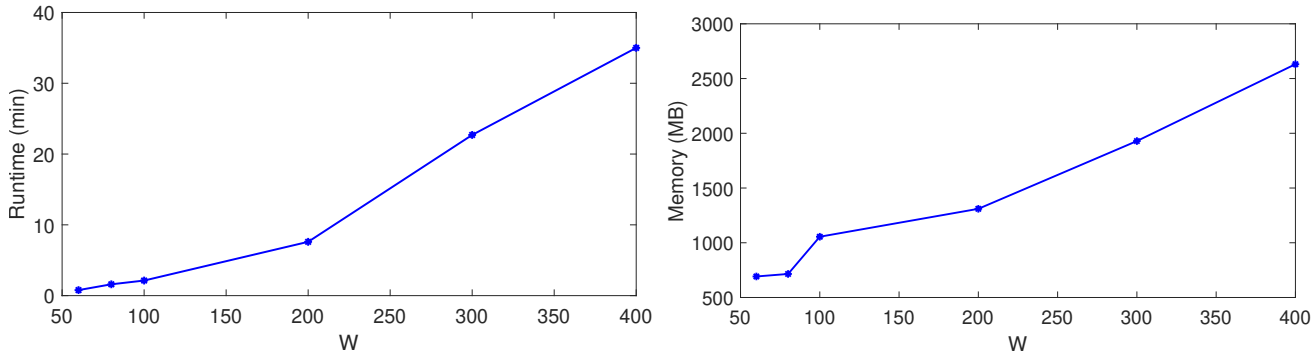
Phase	Preprocess	Initialisation	Query	Update
Time	46 ms	35 min	217 ms	9 s

cryptographic modules. As shown in Fig. 4.4, the costs of using these two modules are almost negligible (RANDOMISE: 0.6 - 5.7 ms, DEANDOMISE: 0.03 - 1.28 ms), because they only include simple circuit structure (i.e., Subtract gates and free XOR gates). Therefore, these two modules help our PPOD to achieve a better security guarantee with a small cost when computing the kNN.

Communication. Fig. 4.5 demonstrates the communication overhead of processing one data point via the kNN module. It shows a similar pattern as in the CPU time evaluation. Specifically, the garbler in the kNN module requires to send a constant size of the input (90 MB) to the evaluator, because the major part of the input is the SORTSHUFFLE circuit, and its size is dependent on k. The communication overhead of the update function is relatively small (1 MB - 12 MB), but it is proportional to k for the same reason as in the CPU time evaluation, i.e., the generated circuit size is proportional to k. The communication overhead slightly increases when the system facilities the RANDOMISE/Derandomise cryptographic modules to enhance the security of data points, especially for the DERANDOMISE module, where the size complexity is $\mathrm{O}(k^2)$. As shown in Fig. 4.5b, it incurs at most 8% more communication overhead when the randomisation is deployed. Nevertheless, we claim that this overhead is affordable, as it only consists of XOR gates, which is a small object comparing to the sorting circuit and it is easy to evaluate (free XOR gates).

4.7.2 Performance of PPOD

First, we note that our proposed PPOD achieves the same accuracy as running the plaintext outlier detection protocol [124] on the unencrypted dataset. Next, we illustrate the run-time performance of each phase of PPOD in Table 4.2. It shows that the preprocess and query can be done in several milliseconds, which indicates that the client (the gateway) can



(a) Runtime (b) Peak memory

Figure 4.6: The runtime and memory usage of the initialisation phase with varying W.

preprocess the data point with small computational resources and get a real-time query result regarding the current outlier model. In addition, although the initialisation needs 35 minutes to execute, it only runs for the first 400 data points. After initialisation, the system can update the existing point only in 9 s, which is a moderate runtime in the application context.

Impact of W. We further examine the runtime performance and memory usage of the initialisation phase for different Ws, as this phase highly depends on the window size W. Fig. 4.6 depicts the result runtime and memory usage respectively. When W increases, the CPU time and memory consumption are expected to increase as well. Besides, we observe that the memory consumption increases sharply when the W reaches 100 (see Fig. 4.6b). The increase of W not only affects the size of the generated circuit and the number of multiplication triplets but also the delay of evaluating the circuit and computing distance via triplets. Therefore, there are more objects residing in the memory for computation, and it leads to the rapid growth of memory consumption. However, such a memory consumption is in an acceptable level in our evaluation platform (48 GB memory) and the other public clouds such as Azure.

4.8 Conclusion

This chapter presents a Privacy-Preserving Outlier Detection (PPOD) protocol targeting the encrypted incremental dataset. Our PPOD protocol leverages the advanced cryptographic primitives (i.e., secure two-party computation protocols) to build several secure and efficient modules. In addition, it adopts the sliding window technique to ensure a practical performance during the update phase with new arrival data points. We implemented our PPOD as a prototype system, and we provided a performance evaluation based on a real-world dataset to demonstrates its accuracy and efficiency.

Chapter 5

Result Pattern Hiding Searchable Encryption for Conjunctive Queries

In this chapter, we revisit the security of the OXT protocol and propose a new SSE protocol, called Hidden Cross-Tags (HXT) to enhance the security of OXT. GraphSE^2 relies on OXT to support set operations. However, a recent study [36] shows that its leakage leads to the breach of database confidentiality. We avoid this leakage by adopting two additional cryptographic primitives - Hidden Vector Encryption (HVE) and probabilistic (Bloom filter) indexing into the HXT protocol. We propose a 'lightweight' HVE scheme that only uses efficient symmetric-key building blocks, and entirely avoids elliptic curve-based operations. At the same time, it affords selective simulation-security against an unbounded number of secret-key queries. Adopting this efficient HVE scheme, the overall practical storage and computational overheads of HXT over OXT are relatively small (no more than 10% for two keywords query, and 21% for six keywords query), while providing a higher level of security.

5.1 Introduction

Privacy of stored data is important in many applications. Yet it is becoming increasingly common for data to be hosted off-site, especially with the rise of cloud computing. However, cloud storage providers often cannot be trusted to respect the privacy of the data they host [145], especially in the face of malicious insiders. A simple solution is to encrypt the data before uploading it to the cloud. However, this would prevent the data from being searched [146, 147]. For example, users may wish to use their mobile phone to search their email. The cloud server will not be able to identify which documents match the search query if the user's email data is encrypted using standard encryption techniques.

Searchable symmetric encryption (SSE) [15, 19, 30, 148–152] offers a potential solution to this problem by providing a way for encrypted data to be searched securely. However, all SSE protocols must have a trade-off between security, performance, and functionality. The performance of SSE protocols is crucial in practice and needs to be comparable to the performance of search on unencrypted databases in order to remain feasible for most practical applications. Consequently, recent research has focused on high-performance SSE protocols that scale efficiently to large databases by adapting data structures used in efficient unencrypted database search to the encrypted database setting. In particular, an efficient class of SSE protocols [19, 30, 150–152] preprocessing the database using an *inverted-index* to allow keyword searches to be performed in sublinear time, with a careful choice of en-

cryption techniques applied to the index to hide its contents and the queries from the server, while allowing a flexible range of search queries. The benchmark in this class of SSE protocols (supporting conjunctive queries) is 'Oblivious Cross-Tags' (OXT) [19]. Nevertheless, to achieve their high performance, these protocols do trade-off security. In particular, they leak some 'partial' information to the server, both on the queries themselves, as well as on the database contents. Recent attacks [36, 86, 105, 153] have shown that such 'partial' leakages can sometimes be exploited in practical applications, especially when the attacker has available some auxiliary side information (which may be obtained publicly in many cases). This state of affairs motivates a re-examination of the security-efficiency trade-offs for SSE protocols and raises the question:

Is it possible to reduce the leakage in existing state-of-the-art efficient SSE protocols while preserving a practical level of performance?

In this chapter, we make progress on this question. We focus, like OXT, only on the setting of conjunctive keyword queries, since such queries tend to be the most common in many practical applications. In this setting, queries consist of any number of keywords, and the result consists of all documents containing all those keywords.

Overview of OXT**.** To explain the main technical ideas behind our construction, it is instructive to first briefly review the 'TSet' and 'XSet' data structures of OXT and how they are used to efficiently process conjunction queries of the form $w_1 \wedge \cdots \wedge w_n$. In OXT, TSet is a cryptographic data structure that associates a list of fixed-length data tuples to each keyword in a database. It is an encrypted inverted index that enables the owner to issue corresponding tokens to retrieve these lists related to the queried keywords. In particular, the client sends the server a 'search token' (called an stag) related to the keyword w_1 (called the 's-term' and denoted by $\mathbf{sterm}$), which allows the server to retrieve from the TSet, the set $\mathsf{DB}(w_1)$ of all database documents containing w_1. In addition, the client sends 'intersection tokens' (called 'xtraps') related to the $n-1$ keyword *pairs* (w_1, w_i) consisting of the 's-term' paired with each of the remaining query keywords w_i, $2 \leq i \leq n$ (called 'x-terms'). The intersection tokens allow the server to *filter* the set $\mathsf{DB}(w_1)$ to determine the $n-1$ subsets of documents $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ that contains the pairs (w_1, w_i), returning only those documents that contain all $\{w_i\}_{1 \leq i \leq n}$. The intersection subsets $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ are efficiently computed by the server using the 'XSet' data structure; the 'XSet' is in essence a list of hashed pairs $h(\mathsf{id}, w)$, over all database document identities id and keywords w contained in id, where h is a certain (public) cryptographic hash function. To filter $\mathsf{DB}(w_1)$ to compute $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$, the server runs through each document id in $\mathsf{DB}(w_1)$ and checks, using the $\mathbf{xtrap}$ token for (w_1, w_i), whether $h(\mathsf{id}, w_i)$ is in the XSet. Therefore, the server computation time is dominated by $|\mathsf{DB}(w_1)|(n-1)$ evaluations of h, which is proportional to just the number of database documents containing the *least frequent* 's-term' w_1, even if other 'xterm' keywords are much more common. However, this method also reveals to the server the Keyword Pair Result Pattern (KPRP, called RP in [19]), i.e. the partial result sets $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ for each pair (w_1, w_i).

Result Pattern (RP) Leakage in SSE protocols. Information leakage in SSE protocols can be classified into three types: storage leakage, query pattern leakage, and result pattern leakage. In this chapter, we focus on the third type of leakage, namely result pattern leakage (RP), i.e. information leaked to the server on the query results. *Ideally, in the conjunctive query SSE context, result pattern leakage would reveal only the Whole Result Pattern (WRP), i.e. the number (and possibly also, identities) of documents matching all query keywords.* Leaking the WRP is in most SSE settings unavoidable, as it would be leaked to the server via the document identities retrieved by the client after the query (unless an ORAM or PIR protocol is used, which currently seems impractical). But in OXT, RP leakage is larger than the WRP ideally desired, and comes in three forms: single keyword result pattern (SP) leakage, Keyword-Pair Result Pattern (KPRP) leakage and multiple keyword cross-query intersection result pattern (IP) leakage.

The KPRP leakage is a 'non-ideal' leakage related to multiple keywords in each query. This reveals *partial query results* to the database server; for an n keyword conjunction query $w_1 \wedge \cdots \wedge w_n$, with w_1 designated as the 's-term' keyword, the KPRP reveals to the server the set $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ of documents containing every *pair* of query keywords of the form (w_1, w_i), $2 \leq i \leq n$. This may leak significantly more information on the inverted-index and/or the query than what is leaked by WRP, which is the set $\cap_{j=1}^{n} \mathsf{DB}(w_j)$ of documents matching all n query keywords.

The recent proposed file-injection attacks [36] have an adapted version which exploits KPRP leakage in OXT to reveal all keywords of a conjunctive query with 100% accuracy. Thus, finding an SSE protocol that eliminates KPRP leakage may be a feasible way to mitigate such an attack towards the conjunctive queries on the inverted-index.

Our Contributions. In this chapter, we present the first efficient SSE protocol, called 'Hidden Cross-Tags' (HXT) that eliminates KPRP leakage presented in the state-of-the-art OXT protocol [19]. Our protocol removes the KPRP 'partial query' leakage component, leaving in its place only the minimal (in our SSE context) and significantly smaller WRP. Furthermore, it leaves other query and result pattern leakage components in existing SSE protocols (i.e. SP and IP) unchanged. Thus, in terms of security, our protocol offers strictly better guarantees than the OXT protocol, and is likely to significantly reduce the rate of query pattern leakage to the database server, as discussed above.

The improved security of our protocol as compared to OXT may be attributed to the adoption of two additional cryptographic primitives - namely, Hidden Vector Encryption (HVE) and probabilistic (Bloom filter) indexing [115]. HVE is a functional encryption primitive typically used in the public-key setting. All existing HVE schemes [1, 37, 154, 155] in the cryptographic literature use either prime order or composite order bilinear pairings over elliptic curve groups. In our HVE-based scheme, a dataset owner can encrypt a set $S \subseteq T = \{1, \ldots, n\}$, for some positive integer n, into a ciphertext c_S, which specifies a 'policy'. Using a master secret key msk, the owner can generate a search token $tk_{S'}$ for any subset

$S' = \{s'_1, \ldots, s'_\ell\}$ of T. Using the token $tk_{S'}$ for S' and the ciphertext c_S for S, anyone can efficiently check whether $S' \subseteq S$ or not, without leaking any partial information if $S' \not\subseteq S$, e.g. whether any particular element s'_i of S' is in S or not (note that in the scheme of [37], the set S is used to generate the token, while the set S' is encrypted in the ciphertext). Unfortunately, adopting such schemes into our protocol leads to a huge compromise in overall performance and efficiency. We address this issue by proposing a 'lightweight' HVE scheme that only uses efficient symmetric-key building blocks, and entirely avoids elliptic curve-based operations. At the same time, it affords selective simulation-security against an unbounded number of secret-key queries. Given this subset membership searchable encryption protocol, a natural idea to apply it to eliminate KPRP leakage in OXT would be to use it to *encrypt the* 'XSet' during set up: we let $S \subseteq T$ denote the XSet list of hashed pairs $h(\mathsf{id}, w)$, over all documents id and keywords w contained in id, and we encrypt S into a ciphertext c_S stored on the server using our HVE-based subset searchable encryption scheme. In the search phase with query $w_1 \wedge \cdots \wedge w_n$, the client issues the server an HVE search token $tk_{S'}$ for $S' = \{h(\mathsf{id}, w_i)\}_{i=2}^n$, $\mathsf{id} \in \mathsf{DB}(w_1)$. This allows the client to check whether $S' \subseteq S$, i.e., whether id contains all n keywords $\{w_i\}_{1 \leq i \leq n}$, without revealing the KPRP information on whether id contains any particular pair (w_1, w_i).

To demonstrate the high performance of our protocols and assess the efficiency overhead of our security improvements, we report our experimental results on the implementation of our protocol and its performance evaluation, compared with the original OXT protocol.

Security of HXT. We prove the client privacy of HXT against the "honest-but-curious" server in a natural extension of the security model used to prove the security of OXT [19], assuming the security of the underlying HVE scheme, the DDH assumption, and the security of the employed symmetric key primitives. The main difference from [19] is that the leakage function in our security model is weaker, as it eliminates the KPRP leakage component and replaces it with the smaller WRP.

Practicality of HXT. We believe HXT is a practical, more secure alternative to OXT for searching large databases. To support this statement, we discuss the practical storage, computation, and communication costs of HXT versus OXT (see Section 5.6 and Section 5.7 for more details and experimental results). In terms of storage, we note that, like HXT, practical implementations of OXT [19] also use a Bloom filter for the XSet, and thus have a similar false positive probability P_e and overhead factor in the number of array storage locations. One additional overhead in our HXT protocol is that each binary entry of our Bloom filter array is encrypted into 1 PRF generated value, i.e. typically 128 bits, whereas these bits are unencrypted in OXT, leading to up to 128 factor overhead (in bit length) in XSet storage for HXT over OXT. While this factor is quite large, our theoretical analysis shows that for a fixed $P_e = 10^{-6}$, the XSet Bloom filter storage size in OXT is 35 times smaller than the TSet hash table size (which is the same in HXT and OXT). Thus, the *overall* storage overhead factor of HXT over OXT without Bloom filter will typically be only $1 + 128/35 \approx 4.65$, which we believe is quite practical.

In terms of computational cost, the HXT has additional costs compared to OXT due to the use of HVE (whereas the practical implementations of OXT only use plain Bloom filter). However, our proposed HVE is based on symmetric-key primitives and bitwise operations. As a consequence, for typical parameters (e.g. $n = 1$ or $n = 2$ keyword queries with false positive rate $P_e = 10^{-6}$), we estimate that the *overall* server computation time overhead for HXT over OXT in such practical applications is likely to be less than 1% and is fully overlapped by I/O costs. Our implementation results indeed show a server runtime overhead between $2\% - 8\%$ for a two keywords query, and 21% for the six keywords query. The HXT client computational cost overhead factor over OXT is higher at $\approx 0.06 * \log_2(1/P_e) \approx 1.19$ for $P_e = 10^{-6}$, but this is likely acceptable as the overall client search time may be dominated by 'out of protocol' costs such as the communication time for downloading result documents. In terms of communications, HXT adds one extra round of communication over the OXT protocol, in which the server communicates to the client the Bloom filter subset it needs to check, and the client returns an HVE token to allow this search. The communication length is still, as in OXT, only proportional to the number of results for the *least-frequent* query keyword.

Additional Related Work. In practice, efficient unencrypted search algorithms usually use a precomputed database *index*. This allows keyword searches to be performed in essentially sublinear time with respect to the size of the database (or more precisely, in time proportional only to the number of *results* matching the query). A number of index-based SSE protocols have been proposed, each more efficient than its predecessor. The first secure encrypted index was proposed in [149], based on the form of *forward* index, storing for each document a Bloom filter containing all the document's keywords. This allows a single document to be searched in $\mathcal{O}(1)$ time but requires each document to be checked in turn, with complexity proportional to the number of documents in the database.

Curtmola et al. [30] were the first to propose using an *inverted-index* data structure, storing in a hash table for each keyword, the encrypted IDs of the documents that contain it (while hiding the number of documents matching each keyword), resulting in complexity proportional to the number of matching *results*, even for searching the whole document collection. However, [30] does not support multiple keyword *conjunctive* queries efficiently; it has complexity proportional to the number of documents matching the *most frequent* queried keyword. Later, [19] presented the OXT protocol, extending [30] by adding a XSet data structure, which lists hashed pairs of keywords and IDs of documents containing them, and reducing search complexity to be proportional to the number of results matching the *least frequent* queried keyword. Our HXT protocol is an improvement of OXT, replacing the XSet by another encrypted data structure to eliminate KPRP leakage while preserving a low search complexity.

Searchable Encryption protocols have also been studied extensively in the *public-key* setting. Such protocols allow any user with the public key to insert data but only allow the user with the private key to search. The use of public-key cryptography makes the proto-

Table 5.1: Notations and terminologies.

Notation	Meaning
λ	a security parameter
id_i	the document identifier of the i-th document
d	number of documents in the database
w	a keyword
W_i	forward index $\{$all w contained in $\mathsf{id}_i\}$
W	the set of all keywords $\cup_{i=1}^{d}\mathsf{W}_i$
ω	the number of elements in W
DB	database $(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d}$
$\mathsf{DB}(w)$	inverted-index $\{$all $\mathsf{id} : w \in \mathsf{W}_{\mathsf{id}}\}$
N	the number of all pairs of (id, w) in DB
n	max. number of conjunctive keywords per query
$\mathbf{sterm}$	the least frequent term among queried terms
$\mathbf{xterm}$	other queried terms (excluding $\mathbf{sterm}$)
$\mathcal{E}$	result set on server-side (encrypted)
R	result set on client-side
Q	the number of all queries
$[n]$	the set of integers $\{1, 2, \ldots, n\}$
k	the number of hash functions in a Bloom filter
m	the length of a Bloom filter
P_e	query false positive probability (per $\mathbf{sterm}$ result)
$\mathbf{negl}(\lambda)$	a negligible function in λ
$s \xleftarrow{\$} S$	uniformly sampling a random s from S
$a \leftarrow \mathcal{A}(\cdot)$	obtaining a as output of running algorithm $\mathcal{A}$

cols less efficient than SSE, but allows more powerful functionality and/or better security properties. The first such protocol was proposed by [156] as a generalisation of anonymous Identity-Based Encryption, and supporting equality queries. It was significantly further generalised in [37] to HVE, applied to conjunctive, subset and range searchable encryption queries. However, it is not clear how to use it to obtain efficient SSE protocols for conjunctive keyword queries. Our HXT protocol fills this gap by proposing a symmetric-key HVE, which is significantly more efficient and suffices for the symmetric-key setting of SSE.

5.2 Preliminaries

We first give a list of notations and definitions needed in our construction and security analysis. A summary of notations and terminologies used in this chapter is given in Table 5.1.

5.2.1 Hardness Assumptions

The security of our construction relies on the hardness of the decisional Diffie-Hellman (DDH) problem [157], the security and correctness of a PRF, and IND-CPA of a symmetric

encryption. We next briefly recall the formal definitions of these primitives and refer the interested reader to [158] for further details.

Definition 6 (DDH). *Let $\mathbb{G}$ be a cyclic group of prime order p, the* DDH *problem is to distinguish the ensembles $\{(g, g^a, g^b, g^{ab})\}$ from $\{(g, g^a, g^b, g^z)\}$, where the elements $g \in \mathbb{G}$, $a, b, z \in \mathbb{Z}_p$ are chosen uniformly at random. Formally, the advantage $\mathrm{Adv}^{\mathsf{DDH}}_{\mathcal{D},\mathbb{G}}(\lambda)$ for any PPT distinguisher $\mathcal{D}$ is defined as*

$$\left|\Pr[\mathcal{D}(g, g^a, g^b, g^{ab}) = 1] - \Pr[\mathcal{D}(g, g^a, g^b, g^z) = 1]\right|.$$

We say that the DDH *assumption holds if for any PPT distinguisher $\mathcal{D}$, its advantage $\mathrm{Adv}^{\mathsf{DDH}}_{\mathcal{D},\mathbb{G}}(\lambda)$ is negligible in λ.*

Definition 7 (PRFs). *Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets, and let $F\colon \{0,1\}^\lambda \times \mathcal{X} \to \mathcal{Y}$ be a function. We say that F is a* PRF *if for all efficient adversaries $\mathcal{A}$, $\mathcal{A}^{\mathsf{PRF}}_{F,\mathcal{A}}(\lambda)$ is negligible, for $\mathcal{A}^{\mathsf{PRF}}_{F,\mathcal{A}}(\lambda)$ defined as*

$$\left|\Pr[\mathcal{A}^{F(\kappa,\cdot)}(1^\lambda) = 1] - \Pr[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1]\right|,$$

where the probability is over the randomness of $\mathcal{A}$, and κ is chosen randomly from $\{0,1\}^\lambda$, and f is chosen randomly from the set of all functions with domain $\mathcal{X}$ to range $\mathcal{Y}$.

Definition 8 (Symmetric Encryption). *A symmetric encryption scheme* Sym *consists of a randomised encryption algorithm* Sym.Enc, *which takes a key $\kappa \in \{0,1\}^\lambda$ and a message $\mu \in \{0,1\}^*$ and returns a ciphertext c, and a deterministic decryption procedure* Sym.Dec, *which accepts the same key κ and the ciphertext c and outputs a message μ.*

A symmetric encryption scheme Sym is called IND-CPA if for all PPT adversaries $\mathcal{A}$, the $\mathrm{Adv}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{Sym}}(\lambda)$ defined as

$$|\Pr[\mathcal{A}^{\mathcal{O}(\kappa,0,\cdot,\cdot)}(1^\lambda) = 1] - \Pr[\mathcal{A}^{\mathcal{O}(\kappa,1,\cdot,\cdot)}(1^\lambda) = 1]|,$$

is negligible in λ, where κ is chosen at random from $\{0,1\}^\lambda$ and the oracle $\mathcal{O}(\kappa, b, \mu_0, \mu_1)$ returns $\perp$ if $|\mu_0| \neq |\mu_1|$, and otherwise it samples $\mathsf{Sym.Enc}(\kappa, \mu_b)$ and returns the result. The correctness is defined as usual.

5.2.2 T-set

A T-set is an expanded inverted-index data structure [19] used for efficient SSE. It is a cryptographic data structure that associates a list of fixed-length data tuples to each keyword in a database. Later it enables the owner to issue corresponding tokens to retrieve these lists related to the queried keywords. A syntax, a correctness definition, a security model, and an instantiation of such a hash table is given in [19]. Throughout this chapter, we identically adopt the notations, definitions and results (in particular, those of Section 5.2.3

and Section 5.6) of that paper regarding T-sets here. To summarise what we use in this work, we give an instantiation of a T-set along with a result regarding its security.

An instantiation of a T-set consists of three algorithms:

- $\mathsf{TSet.Setup}(\mathbf{T})$: The input to this algorithm is an array $\mathbf{T}$, and the output is TSet with b buckets of size s each and a key κ_T. This procedure places the i-th element of $\mathbf{T}[w]$ into bucket $\mathsf{TSet}[b]$, where b is obtained using two secure pseudorandom functions (PRFs) F and $\bar{F}$ and a hash function H.
- $\mathsf{TSet.GetTag}(\kappa_T, w)$: The input to this procedure is the key κ_T and a keyword w, it returns $\bar{F}(w)$.
- $\mathsf{TSet.Retrieve}(\mathsf{TSet}, \bar{F}(w))$: This procedure accepts TSet and the output of TSet.GetTag as its inputs and retrieves the same bucket $\mathsf{TSet}(b)$ and recovers $\mathbf{t} = \mathbf{T}[w]$.

The only leakage of T-set instantiation in [19] is

$$N = \sum_{w \in \mathsf{W}} |\mathbf{T}[w]| = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|, \tag{5.1}$$

that is the number of all pairs of (id, w) in DB.

Theorem 6 (Th. 7 of [19])**.** *For any keyword sequence* $\mathbf{q}$*, including an empty sequence, define* $\mathcal{L}_{\mathbf{T}}(\mathbf{T}, \mathbf{q})$ *as* (5.1)*. The T-set instantiation* Γ *is* $\mathcal{L}_{\mathbf{T}}$*-adaptively-secure assuming that* F *and* $\bar{F}$ *are secure* PRF*s and that* H *is a random oracle.*

5.2.3 Searchable Encryption: Definition and Security

In our single-writer single-reader setting, there are two parties: the data owner (called *client*) of the plaintext database and a cloud service provider (called *server*) that stores the encrypted database. The client can interactively perform search queries over the database. In more details, the client outsources her search service to the server. When she wants to perform a search query, she generates the search token by herself using her private key and forwards the token to the server. With the token, the server retrieves the encrypted identifier or documents for the client.

Formally, the syntax of our SSE protocol Π consists of the following algorithms:

- $\mathsf{SE.EDBSetup}(1^\lambda, \mathsf{DB}) \rightarrow (param, mk, \mathsf{EDB})$: Run by client, takes 1^λ and DB as inputs and returns the system public parameters $param$, master key mk and encrypted database EDB. $param$ is publicly known. mk is kept by client and EDB is stored in the server.
- $\mathsf{SE.Search}(param, mk, \psi(\mathbf{w}), \mathsf{EDB}) \rightarrow \mathsf{DB}(\psi(\mathbf{w}))$: A protocol, runs between client and server interactively. Client's inputs are $param$, mk and query $\psi(\mathbf{w})$, while server's inputs are $param$ and EDB. At the end of the protocol, client outputs document identifiers $\mathsf{DB}(\psi(\mathbf{w}))$ matching query $\psi(\mathbf{w})$, and server outputs nothing.

We say that Π is *computationally correct with false positive rate* P_e if for any database DB of size $\mathsf{poly}(\lambda)$ and any conjunctive query $\psi(\mathbf{w}) = w_1 \wedge \cdots \wedge w_n$, the following game $\mathbf{Cor}^{\Pi}_{\mathcal{A}}(\lambda)$ is won with probability at most $|\mathsf{DB}(w_1)| \cdot P_e + \mathbf{negl}(\lambda)$. In this game, the challenger runs $\mathsf{SE.EDBSetup}$ to get EDB from DB and simulates $\mathsf{SE.Search}$ on EDB and query $\psi(\mathbf{w})$ to compute the client search result S. The game is won (and returns 1) if $S \neq \mathsf{DB}(\psi(\mathbf{w}))$. We remark that this relaxed correctness definition, allowing a 'per $\mathbf{sterm}$ result' false positive rate P_e, is also required for the Bloom filter based practical implementation of OXT (though not formalised in [19]).

We consider the following security model for SSE, which is exactly the one from [19], except that our leakage function will reveal WRP instead of the KPRP (known as RP) in [19]. The model is parameterised by a leakage function $\mathcal{L}$, as described below, which captures information allowed to learn by an adversary from the interaction with a secure searchable encryption protocol. Loosely speaking, the security says that the server's view during a non-adaptive attack can be properly simulated given only the output of the leakage function $\mathcal{L}$.

Let $\Pi = (\mathsf{SE.EDBSetup}, \mathsf{SE.Search})$ be a searchable encryption protocol and $\mathcal{A}$ and $\mathcal{S}$ be two efficient algorithms. The security is formally defined via a real experiment $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ and an ideal experiment $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$ as follows:

- $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$: The adversary $\mathcal{A}(1^{\lambda})$ chooses a database DB and a query list $\mathbf{q}$. Then the experiment runs $\mathsf{SE.EDBSetup}(1^{\lambda}, \mathsf{DB})$ to get $(param, mk, \mathsf{EDB})$ and returns $param$ and EDB to $\mathcal{A}$. After that, for each $i \in |\mathbf{q}|$, the experiment runs the $\mathsf{SE.Search}$ on input $\mathbf{q}[i]$, and stores the resulted transcript and the client's output into $\mathbf{t}[i]$. Finally, EDB and $\mathbf{t}$ will be given to $\mathcal{A}$. Eventually, the experiment outputs the bit that $\mathcal{A}$ returns.
- $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$: The adversary $\mathcal{A}(1^{\lambda})$ chooses a database DB and a query list $\mathbf{q}$. Then the experiment runs $\mathsf{EDB} \leftarrow \mathcal{S}(\mathcal{L}(\mathsf{DB}, \mathbf{q}))$ and gives it to $\mathcal{A}$. Eventually, the experiment outputs the bit that $\mathcal{A}$ returns.

Definition 9 (Security)**.** *The searchable encryption protocol* Π *is called* $\mathcal{L}$*-semantically secure against non-adaptive attacks if for all PPT adversaries* $\mathcal{A}$ *there exists an efficient simulator* $\mathcal{S}$ *such that*

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

An adaptive model can also be defined correspondingly as in [19]. In such a model, the query list $\mathbf{q}$ will not be known to the challenger at the beginning of the real and ideal games. Instead, it selects repeatedly search query $\mathbf{q}$ after each experiment runs $\mathsf{SE.EDBSetup}$. Note that in the real game, the input to $\mathsf{SE.Search}$ is only EDB; and to generate EDB, the simulator $\mathcal{S}$ has only access to the leakage function $\mathcal{L}(\mathsf{DB}, \mathbf{q})$.

5.2.4 Bloom Filters

A Bloom filter (BF) is a probabilistic (indexing) data structure to represent a set $\mathcal{V} = \{s_1, s_2, \ldots, s_N\}$ of N elements. Its main functionality is to support membership queries. The idea is to choose k independent hash functions, $\{H_i\}_{1 \le i \le k}$, each with domain $\mathcal{V}$ and range $[m]$. The Bloom filter consists of a binary vector $\mathbf{b}$ of m bits, initially all 0. In order to set up BF for $\mathcal{V}$, for each element $s \in \mathcal{V}$, the bits at positions $\{H_i(s)\}_{1 \le i \le k}$ are changed to 1. To test membership of q, we check if $\mathbf{b}$ has 1's in all positions $\{H_i(q)\}_{1 \le i \le k}$, and if so, we conclude $q \in \mathcal{V}$ with high probability. Otherwise, we conclude $q \notin \mathcal{V}$ with probability 1. If $q \notin \mathcal{V}$ yet the membership test returns 1, we call it a "false positive" event. Given a BF set up for $\mathcal{V}$, and $q \notin \mathcal{V}$, the false positive probability for q over a uniformly random choice of $\{H_i(q)\}_{1 \le i \le k}$ is

$$P_e \le (1 - \mathrm{e}^{-k \cdot N/m})^k. \tag{5.2}$$

Parameters are chosen such that P_e is negligible. It can be seen from (5.2) that there is a trade-off between k and the probability of having a false positive: smaller k are preferred since they reduce the computation cost, but it is also necessary to maintain the desired false positive rate. Given N, P_e, the optimal choice of k is $k \approx \log_2(1/P_e)$, while the required $m \approx 1.44 \cdot \log_2(1/P_e) \cdot N$ (i.e. $k \approx 20$, $m \approx 29 \cdot N$ for $P_e = 10^{-6}$) [159].

5.2.5 Hidden Vector Encryption and its Security

Predicate encryption offers a new cryptographic mechanism that provides fine-grained access control over an encrypted database. In predicate encryption, decryption keys are associated with boolean predicates $f : \Sigma \longrightarrow \{0, 1\}$ over a pre-defined set of attributes Σ, while each ciphertext is associated with an attribute $I \in \Sigma$, and a payload messages $\mu \in \mathcal{M}$. A decryption key can be used to decrypt a ciphertext only if the attribute I satisfies $f(I) = 1$. A major application of this encryption paradigm is to outsource encrypted data to a server, and yet retain the ability to make queries on the data without revealing more information than absolutely necessary. This is, in principle, similar to the concept of SSE described in the previous subsection. Hidden Vector Encryption (HVE) is one such predicate encryption scheme that supports conjunctive, equality, comparison, and subset queries on encrypted data. While HVE was formally defined in the public-key setting in [37], we adopt their definition to the symmetric-key setting in order for it to be applicable in the context of SSE.

A symmetric-key HVE may be defined as an ensemble of the following four PPT algorithms:

- HVE.Setup(λ): takes a security parameter λ and outputs a master secret key msk. It also defines the message space $\mathcal{M}$.
- HVE.KeyGen($msk, \mathbf{v} \in \Sigma_*^m$): takes a predicate vector $\mathbf{v}$, the master secret key msk and outputs a decryption key $\mathbf{s}$.

- $\mathsf{HVE.Enc}(msk, \mu \in \mathcal{M}, \mathbf{x} \in \Sigma^m)$: takes as input a message μ, an index vector $\mathbf{x}$, and the master secret key msk and outputs the ciphertext $\mathbf{c}$ associated with $(\mathbf{x}, \mu)$.
- $\mathsf{HVE.Query}(\mathbf{s}, \mathbf{c})$: takes a ciphertext $\mathbf{c}$ corresponding to the index vector $\mathbf{x}$ and a decryption key $\mathbf{s}$ corresponding to the predicate vector $\mathbf{v}$ and outputs the message μ if $P_{\mathbf{v}}^{\mathsf{HVE}}(\mathbf{x}) = 1$.

We say that a symmetric-key HVE is correct [37] if for all security parameters λ, all $(\mu, \mathbf{x}) \in \mathcal{M} \times \Sigma^m$ and all predicate vectors $\mathbf{v}$, after sequentially running $\mathsf{HVE.Setup}(\lambda)$ to get msk, $\mathsf{HVE.KeyGen}(msk, \mathbf{v} \in \Sigma_*^m)$ to get $\mathbf{s}$, and $\mathsf{HVE.Enc}(msk, \mu \in \mathcal{M}, \mathbf{x} \in \Sigma^m)$ to get $\mathbf{c}$, if $P_{\mathbf{v}}^{\mathsf{HVE}}(\mathbf{x}) = 1$, then $\mathsf{HVE.Query}(\mathbf{s}, \mathbf{c}) = \mu$, otherwise

$$\Pr\left[\mathsf{HVE.Query}(\mathbf{s}, \mathbf{c}) = \perp\right] = 1 - \mathbf{negl}(\lambda).$$

The next step is to formally define the notion of semantic security for symmetric-key HVE against PPT adversaries. The definitions are presented in the simulation-setting, which naturally subsumes the traditional security definitions for HVE in the indistinguishability setting. Prior to presenting the formal definition, we present two auxiliary definitions that constitute the trivial leakage from any symmetric-key HVE scheme. Given a predicate vector $\mathbf{v} = (v_1, \ldots, v_m) \in \Sigma_*^m$, its *wildcard pattern* $\alpha(\mathbf{v})$ is a vector of the same size as the predicate vector $\mathbf{v}$, which is 1 if $v_j \neq *$, and 0 otherwise. Also, given a predicate vector $\mathbf{v} \in \Sigma_*^m$ and an index vector $\mathbf{x} \in \Sigma$, the decryption pattern $\beta(\mathbf{v}, \mathbf{x})$ is a boolean value such that $\beta(\mathbf{v}, \mathbf{x}) = 1$ if $P_{\mathbf{v}}^{\mathsf{HVE}}(\mathbf{x}) = 1$, and 0 otherwise. With these definitions in place, we now define the real and simulation experiments for a symmetric-key HVE scheme.

The Real Experiment

The real experiment for a symmetric-key HVE scheme involves a challenger and a stateful PPT adversary $\mathcal{A}$, who interact as follows:

- **Setup phase:** During this phase the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$ and gives it to the challenger. In response, the challenger runs $\mathsf{HVE.Setup}(\lambda)$ and outputs msk and message space $\mathcal{M}$. $\mathcal{M}$ is given to the adversary.
- **Query phase 1:** The adversary $\mathcal{A}$ adaptively chooses predicate vectors $\mathbf{v}_j$, for $j \in [q_1]$. The challenger then uses $\mathbf{v}_j$ and msk to run $\mathsf{HVE.KeyGen}$, and responds with the corresponding decryption keys $\mathbf{s}_j$.
- **Challenge phase:** The adversary $\mathcal{A}$ outputs a message $\mu \in \mathcal{M}$. The challenger runs $\mathsf{HVE.Enc}$ using msk, $\mathbf{x}$ and μ and obtains the ciphertext $\mathbf{c}$, which is given to $\mathcal{A}$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1** and receives $\mathbf{s}_j$ for $q_1 + 1 \leq j \leq q_2$.

Let $r_{\mathcal{A}}$ denote the internal random bits used by $\mathcal{A}$ during the real experiment. We use the term $\mathsf{VIEW}_{\mathcal{A},\mathbf{Real}}$ to denote the ensemble $(\mathcal{M}, \mathbf{c}, \{\mathbf{v}_j\}_{j \in [q_2]}, r_{\mathcal{A}})$, which is essentially the view of the adversary $\mathcal{A}$ in the real experiment described above.

The Simulation Experiment

The simulation experiment for a symmetric-key HVE scheme involves a stateful PPT simulator $\mathcal{S}$ and a stateful PPT adversary $\mathcal{A}$, who interact as follows:

- **Setup phase:** During this phase the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$. In response, the simulator $\mathcal{S}$ provides $\mathcal{A}$ with the message space $\mathcal{M}$. Note that $\mathcal{S}$ does not receive the actual attribute vector $\mathbf{x}$.
- **Query phase 1:** The adversary adaptively chooses predicate vectors $\mathbf{v}_j$, for $j \in [q_1]$. For each such predicate, the simulator $\mathcal{S}$ only receives as input the wildcard pattern $\alpha(\mathbf{v}_j)$ and the decryption pattern $\beta(\mathbf{v}_j, \mathbf{x})$. It responds with the corresponding decryption keys $\mathbf{s}_j$.
- **Challenge phase:** The adversary $\mathcal{A}$ outputs a message $\mu \in \mathcal{M}$. The simulator $\mathcal{S}$ responds with the challenge ciphertext $\mathbf{c}$ corresponding to $\mathbf{x}, \mu$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1** and receives $\mathbf{s}_j$ for $q_1 + 1 \leq j \leq q_2$.

Again, let $r_{\mathcal{A}}$ denote the internal random bits used by $\mathcal{A}$ during the simulation experiment. We use the term $\mathsf{VIEW}_{\mathcal{A},\mathbf{Sim}}$ to denote the ensemble $(\mathcal{M}, \mathbf{c}, \{\mathbf{v}_j\}_{j\in[q_2]}, r_{\mathcal{A}})$, which is essentially the view of the adversary $\mathcal{A}$ in the simulation experiment described above.

We define the advantage of a PPT distinguisher $\mathcal{D}$ in distinguishing the real and simulation experiments described above as $\mathrm{Adv}^{\mathsf{HVE}}_{\mathcal{D},\mathcal{A}}(\lambda) = |\Pr[\mathcal{D}(\mathsf{VIEW}_{\mathcal{A},\mathbf{Real}}) = 1] - \Pr[\mathcal{D}(\mathsf{VIEW}_{\mathcal{A},\mathbf{Sim}}) = 1]|$. A symmetric-key HVE scheme is said to be selectively simulation-secure if for all such PPT distinguishers $\mathcal{D}$ and PPT adversaries $\mathcal{A}$, the function $\mathrm{Adv}^{\mathsf{HVE}}_{\mathcal{D},\mathcal{A}}(\lambda)$ is a negligible in λ.

5.3 Lightweight Symmetric-Key Hidden Vector Encryption

In this section, we propose a novel HVE scheme in the symmetric-key setting, referred to as SHVE, that entirely avoids the use of pairings. Our construction is predicate-only (implying that the payload message is "True" by default) and is amenable to parallel implementations for high performance. At the same time, it guarantees selective simulation-security against probabilistic polynomial-time adversaries for a single ciphertext query and an unbounded number of decryption key queries.

5.3.1 Detailed SHVE Construction

We now present the details of our proposed SHVE construction. Let Σ be a finite set of attributes and $*$ be a wildcard symbol ("don't care" value) not in Σ. Define $\Sigma_* = \Sigma \cup \{*\}$. In our framework, Σ is typically a finite field $\mathbb{Z}_p$, where p is a prime. We define a family of

predicates $\mathcal{P}^{\mathsf{SHVE}} : \Sigma^m \longrightarrow \{0,1\}$ as follows. For each $\mathbf{v} = (v_1, \ldots, v_m) \in \Sigma_*^m$, there exists a predicate $P_{\mathbf{v}}^{\mathsf{SHVE}} \in \mathcal{P}^{\mathsf{SHVE}}$, such that for $\mathbf{x} = (x_1, \ldots, x_m) \in \Sigma^m$, we have:

$$P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = \begin{cases} 1 & \forall\, 1 \leq i \leq m \ (v_i = x_i \text{ or } v_i = *), \\ 0 & \text{otherwise.} \end{cases}$$

In other words, the vector $\mathbf{x}$ matches $\mathbf{v}$ in all the coordinates that are not the wildcard character $*$. The parameter m is referred to as the *width* of the SHVE.

Our construction uses a pseudorandom function (PRF) $F_0 : \{0,1\}^\lambda \times \{0,1\}^{\lambda+\log\lambda} \longrightarrow \{0,1\}^{\lambda+\log\lambda}$ and an IND-CPA secure symmetric encryption scheme $(\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ with both the key-space and the plaintext-space being $\{0,1\}^{\lambda+\log\lambda}$, where λ is a security parameter. The details of the construction are as follows:

- $\mathsf{SHVE.Setup}(1^\lambda)$: On input the security parameter λ, the algorithm uniformly samples $msk \xleftarrow{\$} \{0,1\}^\lambda$. It then defines the payload message space $\mathcal{M} = \{\text{'True'}\}$, and outputs $(msk, \mathcal{M})$.
- $\mathsf{SHVE.KeyGen}(msk, \mathbf{v} \in \Sigma_*^m)$: On input a predicate vector $\mathbf{v} = (v_1, \ldots, v_m)$ and the master secret key msk, we denote by $S = \{l_j \in [m] \mid v_{l_j} \neq *\}$ the set of all locations in $\mathbf{v}$ that do not contain wildcard characters. Let these locations be $l_1 < l_2 < \ldots < l_{|S|}$. The algorithm samples $K \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$ and sets the following:

$$\begin{aligned} d_0 &= \oplus_{j \in [|S|]} \left(F_0(msk, v_{l_j} || l_j)\right) \oplus K, \\ d_1 &= \mathsf{Sym.Enc}\left(K, 0^{\lambda+\log\lambda}\right). \end{aligned}$$

  The algorithm finally outputs the decryption key:

$$\mathbf{s} = (d_0, d_1, S).$$

- $\mathsf{SHVE.Enc}(msk, \mu = \text{'True'}, \mathbf{x} \in \Sigma^m)$: On input a message μ, an index vector $\mathbf{x} = (x_1, \ldots, x_m)$ and the master secret key msk, this algorithm sets $c_l = F_0(msk, x_l || l)$, for each $l \in [m]$, and outputs the ciphertext:

$$\mathbf{c} = (\{c_l\}_{l \in [m]}).$$

- $\mathsf{SHVE.Query}(\mathbf{s}, \mathbf{c})$: The query algorithm takes as input a ciphertext $\mathbf{c}$ and a decryption key $\mathbf{s}$, and parses them as:

$$\begin{aligned} \mathbf{c} &= \left(\{c_l\}_{l \in [m]}\right), \\ \mathbf{s} &= (d_0, d_1, S). \end{aligned}$$

where $S = \{l_1, l_2, \ldots, l_{|S|}\}$. The algorithm computes the following:

$$K' \quad = \quad \left(\oplus_{j \in [|S|]} c_{l_j}\right) \oplus d_0.$$

Next the decryption algorithm computes:

$$\mu' = \mathsf{Sym.Dec}\left(K', d_1\right).$$

If $\mu' = 0^{\lambda + \log \lambda}$, the decryption algorithm outputs 'True' else it outputs $\perp$.

The correctness of the aforementioned scheme may be verified as follows. Let $\mathbf{c} = (\{c_l\}_{l \in [m]})$ be a ciphertext corresponding to an index vector $\mathbf{x} = (x_1, \ldots, x_m)$, and let $\mathbf{s} = (d_0, d_1, S)$ be a decryption key corresponding to predicate vector $\mathbf{v} = (v_1, \ldots, v_m)$. Let $S = \{l_1, l_2, \ldots, l_{|S|}\}$. We consider the following scenarios:

- If $P^{\mathsf{SHVE}}_{\mathbf{v}}(\mathbf{x}) = 1$, we must have $v_{l_j} = x_{l_j}$ for each $j \in [|S|]$. In other words, we have $c_{l_j} = F_0(msk, v_{l_j} || l_j)$ for each $j \in [|S|]$. This now immediately leads to the following relation:

$$\begin{aligned} K' \quad &= \quad \left(\oplus_{j \in [|S|]} c_{l_j}\right) \oplus d_0 = K, \\ \mu' \quad &= \quad \mathsf{Sym.Dec}\left(K, d_1\right) \\ &= \quad 0^{\lambda + \log \lambda}. \end{aligned}$$

- If $P^{\mathsf{SHVE}}_{\mathbf{v}}(\mathbf{x}) = 0$, we must have $v_{l_j} \neq x_{l_j}$, for some $j \in [|S|]$. This in turn implies that for some $j \in [|S|]$, $c_{l_j} \neq F_0(msk, v_{l_j} || l_j)$, and hence, during decryption, $K' \neq K$. This ensures that except with negligible probability, we have $\mu' \neq 0^{\lambda + \log \lambda}$, and the decryption algorithm returns the failure symbol $\perp$.

This establishes the correctness of the SHVE scheme. Quite evidently, in our construction, the key-generation and query algorithms operate only on the secret-key/ciphertext components listed in the subset S, which correspond to the non-wildcard entries in the predicate vector. The speed-up achieved as a result of this property is particularly evident in applications where a majority of the predicate vectors have only sparsely distributed non-wildcard entries. As it turns out, our SSE construction, presented in the following section, presents precisely such an application scenario.

5.3.2 Security of SHVE

We now state the following theorem for the security of our SHVE construction:

Theorem 7. *Our predicate-only* SHVE *construction is selectively simulation-secure in the ideal cipher model as per the security definitions presented in Section 5.2.5.*

Proof. We first show a construction for the simulator $\mathcal{S}$ in the simulation experiment. The simulator models the symmetric encryption scheme $(\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ as an ideal cipher. In particular, the adversary $\mathcal{A}$ either issues encryption queries of the form (κ, μ) or decryption queries of the form (κ, c). The simulator $\mathcal{S}$ maintains a table of the form (κ, μ, c). Upon receipt of an encryption/decryption query, it looks up the table, and either returns an already existing entry or adds a uniformly random entry to the table and returns the same. The simulator operates as follows:

- **Setup phase:** Suppose the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$. The simulator $\mathcal{S}$ sets $\mathcal{M} = \{\text{'True'}\}$ and provides the same to $\mathcal{A}$. It additionally randomly chooses $c_l \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$ for $l \in [m]$.
- **Query phase 1:** The adversary adaptively chooses predicates $P^{\mathsf{SHVE}}_{\mathbf{v}_j}$, for $j \in [q_1]$. For each such predicate, the simulator $\mathcal{S}$ receives the corresponding wildcard pattern $\alpha(\mathbf{v}_j) = (\alpha_{j,1}, \ldots, \alpha_{j,m})$ and the decryption pattern $\beta(\mathbf{v}_j, \mathbf{x})$. $\mathcal{S}$ then does the following:
  - $\mathcal{S}$ computes $S_j = \{l_i \in [m] \mid \alpha_{j,l_i} = 1\}$. Let $S_j = \{l_1, l_2, \ldots, l_{|S_j|}\}$.
  - If $\beta(\mathbf{v}_j, \mathbf{x}) = 1$, it randomly samples $K \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$. Next, for $i \in [|S_j|]$, it sets the following :

    $$d_{j,0} \;=\; \left(\oplus_{i\in[|S_j|]} c_{l_i}\right) \oplus K.$$

    Finally, it sets $d_{j,1} = \mathsf{Sym.Enc}(K, 0^{\lambda+\log\lambda})$. Note that since $(\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ is modelled as an ideal cipher, all the aforementioned $\mathsf{Sym.Enc}$ operations are essentially implemented via table-look-up operations.
  - Otherwise, if $\beta(\mathbf{v}_j, \mathbf{x}) = 0$, the simulator sets $d_{j,0}, d_{j,1} \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$.
  - Finally, the simulator sets the decryption key:

    $$\mathbf{s}_j = (d_{j,0}, d_{j,1}, S_j)\,.$$

    This decryption key is subsequently provided to the adversary $\mathcal{A}$.
- **Challenge phase:** The simulator $\mathcal{S}$ provides $\mathcal{A}$ with the challenge ciphertext $\mathbf{c} = \left(\{c_l\}_{l\in[m]}\right)$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1**, and $\mathcal{S}$ responds with $\mathbf{s}_j$ for $q_1 + 1 \leq j \leq q_2$ as described above.

The indistinguishability of the ciphertext $\mathbf{c}$ and the secret keys $\mathbf{s}_j$ for $j \in [q_2]$ from the real experiment follows directly from the following facts:

- The payload message is 'True' by default in the predicate-only version of the scheme. Now, for each $j \in [q_2]$ such that $P^{\mathsf{SHVE}}_{\mathbf{v}_j}(\mathbf{x}) = 1$, decrypting $\mathbf{c}$ using $\mathbf{s}_j$ returns 'True'. On the other hand, for each $j \in [q_2]$ such that $P^{\mathsf{SHVE}}_{\mathbf{v}_j}(\mathbf{x}) = 0$, decrypting $\mathbf{c}$ using $\mathbf{s}_j$ returns 'True' with only negligible probability.

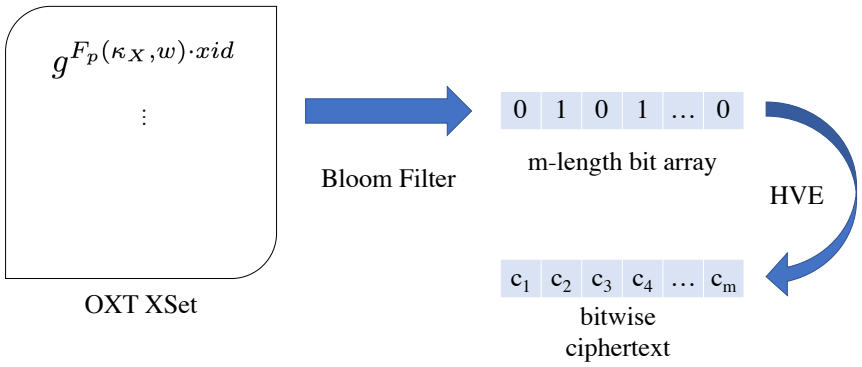


Figure 5.1: An illustration of difference w.r.t XSet in HXT compare to OXT.

- The encryption and decryption outputs of $(\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ are indistinguishable from uniformly random since they are modelled on the ideal cipher model.
- The outputs of the PRF F_0 are indistinguishable from random to a PPT distinguisher $\mathcal{D}$ that can guess the master secret key msk with only negligible probability (Definition 7)

.

□

5.4 HXT Construction

We now give the main construction of our SSE protocol and then prove its correctness and analyse its security.

5.4.1 Hidden Cross Tags (HXT) Protocol

Our SSE protocol HXT uses (for security parameter λ) (*i*) a cyclic group $\mathbb{G}$ with prime order p and generator g, for which the DDH assumption holds (Def. 6), (*ii*) a symmetric-key HVE (see Section 5.2.5), for which we use (for the rest of the chapter) SHVE presented in previous section, (*iii*) a symmetric key encryption scheme Sym with key space $\{0,1\}^\lambda$ (Def. 8), (*iv*) a Bloom filter BF with length m and k hash functions $\{H_j\}_{1\le j\le k}$ (see Section 5.2.4), and finally (*v*) PRFs F with range $\{0,1\}^\lambda$ and F_p with range $\mathbb{Z}_p^*$ (Def. 7).

The HXT protocol consists of two algorithms: SE.EDBSetup and SE.Search.

The setup algorithm SE.EDBSetup (Algorithm 12) gets the security parameter λ and DB and returns the $param$, mk and EDB. The encrypted database EDB has two components: $\mathsf{EDB}(1)$ is TSet generated exactly as in OXT, and $\mathsf{EDB}(2)$, which is shown as the blue part in Fig. 5.1: it is an HVE encryption of a carefully designed Bloom filter BF, which is set up for XSet elements of the form $h(\mathsf{id}, w) = g^{F_p(\kappa_X,w)\cdot \mathsf{xid}}$, for encrypted identifiers $\mathsf{xid} = F_p(\kappa_I, \mathsf{id})$ over all $\mathsf{id} \in \mathsf{DB}(w)$. The setup algorithm writes 1's into BF in positions in set

$$S = \{H_j(h(\mathsf{id}, w))\}_{1\le j\le k},$$

Algorithm 12 SE.EDBSetup

Input: 1^λ, DB
Output: mk, $param$, EDB

function SE.EDBSetup(1^λ, DB)
  Initialise $\mathbf{T} \leftarrow \emptyset$ indexed by keywords W.
  Select key κ_S for PRF F.
  Select keys $\kappa_I,\ \kappa_Z,\ \kappa_X$ for PRF F_p.
  Select hash functions $\{H_j\}_{1\le j\le k}$ for BF.
  Run HVE.Setup(1^λ) to get msk.
  Initialise EDB $\leftarrow \{\}$.
  for $w \in$ W **do**
    Initialise $\mathbf{t} \leftarrow \{\}$.
    Compute $\kappa_e \leftarrow F(\kappa_S, w)$.
    for id $\in$ DB(w) **do**
      Set a counter $c \leftarrow 1$.
      Compute xid $\leftarrow F_p(\kappa_I, \mathsf{id})$.
      Compute $z_w \leftarrow F_p(\kappa_Z, w||c)$; $y_c \leftarrow \mathsf{xid} \cdot z_w^{-1}$.
      Compute $e_c \leftarrow$ Sym.Enc(κ_e, id).
      Append (y_c, e_c) to $\mathbf{t}$ and set $c \leftarrow c+1$.
    end for
    Set $\mathbf{T}[w] \leftarrow \mathbf{t}$.
  end for
  Compute (TSet, κ_T) $\leftarrow$ TSet.Setup($\mathbf{T}$).
  Let EDB(1) = TSet.
  Initialise BF $\leftarrow 0^m$.
  for $w \in$ W **do**
    for id $\in$ DB(w) **do**
      Compute xid $\leftarrow F_p(\kappa_I, \mathsf{id})$.
      for $j = 1 : k$ **do**
        Let $h_j(\mathsf{id}, w) \triangleq H_j\left(g^{F_p(\kappa_X, w)\cdot \mathsf{xid}}\right)$.
        Set BF$[h_j(\mathsf{id}, w)] \leftarrow 1$.
      end for
    end for
  end for
  Compute $\mathbf{c} \leftarrow$ HVE.Enc($msk, \mu =$ 'True', BF).
  Let EDB(2) = $\mathbf{c}$.
  return $param = (\{H_j\}_{j=1}^k)$, $mk = (msk, \kappa_S, \kappa_I, \kappa_Z, \kappa_X, \kappa_T)$, EDB = (EDB(1), EDB(2)).
end function

over all (id, w) pairs with id $\in$ DB(w), and then encrypts BF with HVE.Enc. The new parts of our protocol compared to OXT are coloured blue.

The search protocol SE.Search is shown in Algorithm 13, where the first 14 lines generate stag and xtokens similar to OXT. The XSet membership test for conjunctions in OXT is replaced by a HVE token generation and query. Namely, the HVE token token$_c$ for all id$_c$ $\in$ DB(w_1) is generated for a predicate (BF) vector $\mathbf{v}_c$ with 1's in positions in set $S' = \{H_j(h(\mathsf{id}_c, w_i)\}_{1\le j\le k}^{2\le i\le n}$ and wildcards in other positions. [1] Consequently (as the message encrypted by HVE was set to 'True' in SE.EDBSetup) the HVE.Query returns 'True' if $S' \subseteq S$,

[1] Under the "honest-but-curious" assumption, the server following the protocol can not arbitrarily check the membership of Bloom filter. Therefore, the server is not able to check XSet like in OXT.

Algorithm 13 SE.Search

Input: $param$, mk, query $\bar{w} = (w_1 \wedge \cdots \wedge w_n)$ with **sterm** w_1, EDB.
Output: Result R.

1: **function** SE.Search($param, mk, \bar{w}$, EDB)
2: Client's inputs are $(param, mk, \bar{w})$ and server's input is $(param, \mathsf{EDB})$.
3: Client initialises $R \leftarrow \{\}$.
4: Client computes stag $\leftarrow \mathsf{TSet.GetTag}(\kappa_T, w_1)$ and sends stag to the server.
5: Server lets $\mathsf{TSet} = \mathsf{EDB}(1)$.
6: Server computes $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{TSet}, \mathsf{stag})$, sends $|\mathbf{t}|$ to client, and starts accepting xtokens computed by client as follows:
7: **for** $c = 1 : |\mathbf{t}|$ **do**
8: Client computes $\eta_{w_1} \leftarrow F_p(\kappa_Z, w_1||c)$.
9: **for** $\ell = 2 : n$ **do**
10: Client computes $\mathsf{xtoken}[c, \ell] \leftarrow g^{\eta_{w_1} \cdot F_p(\kappa_X, w_\ell)}$.
11: **end for**
12: Client sets $\mathsf{xtoken}[c] \leftarrow (\mathsf{xtoken}[c, 2], \ldots, \mathsf{xtoken}[c, n])$.
13: Client sends $\mathsf{xtoken}[c]$ to server.
14: **end for**
15: Server initialises $\mathcal{E} \leftarrow \{\}$.
16: **for** $c = 1 : |\mathbf{t}|$ **do**
17: Client initialises $\mathbf{v}_c \leftarrow *^m$.
18: Server recovers (y_c, e_c) from the c-th component of $\mathbf{t}$.
19: **for** $\ell = 2 : n$ **do**
20: Server computes $\mathsf{xtag} = \mathsf{xtoken}[c, \ell]^{y_c}$.
21: **for** $j = 1 : k$ **do**
22: Server computes $u_j \leftarrow H_j(\mathsf{xtag})$.
23: Server sends u_j to client.
24: Client sets $\mathbf{v}_c[u_j] \leftarrow 1$.
25: **end for**
26: **end for**
27: Client computes $\mathsf{token}_c \leftarrow \mathsf{HVE.KeyGen}(msk, \mathbf{v}_c)$.
28: Client sends token_c to server.
29: Server lets $\mathbf{c} = \mathsf{EDB}(2)$.
30: Server computes $\mathsf{res}_c \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c, \mathbf{c})$.
31: **if** $\mathsf{res}_c =$ 'True' **then**
32: Server adds e_c to $\mathcal{E}$ (i.e., $\mathcal{E} = \mathcal{E} \cup \{e_c\}$)
33: **end if**
34: **end for**
35: Server sends $\mathcal{E}$ to client.
36: Client computes $\kappa_e \leftarrow F(\kappa_S, w_1)$,
37: Client computes $\mathsf{id}_c \leftarrow \mathsf{Sym.Dec}(\kappa_e, e_c)$, and adds id_c to R for all $e_c \in \mathcal{E}$.
38: **return** R
39: **end function**

i.e. if *all* $n-1$ **xterms** w_i are in the document id_c. Otherwise, HVE.Query returns $\perp$, without revealing KPRP information on which w_i are in id_c. Importantly, in step 27 and 30, the HVE.KenGen and HVE.Query algorithm only uses components of $\mathbf{c}$ in the non-wildcard positions of $\mathbf{v}_c$ and token_c, so search run-time is only proportional to $|\mathsf{DB}(w_1)| \cdot n \cdot k$ (similar to OXT), and not to the size m of the BF. We next show that HXT is correct with the Bloom filter's false positive rate P_e.

Theorem 8. *If the underlying* HVE *scheme and T-set scheme* Γ *are correct, and the* PRF*s* F *and* F_p *are secure, then* HXT *is computationally correct with false positive rate* $P_e \leq (1 - \mathrm{e}^{-k \cdot N/m})^k$.

Proof. Let G_0 denote the original game $\mathbf{Cor}_{\mathcal{A}}^{\mathsf{HXT}}(\lambda)$. We want to show $\Pr[\mathsf{G}_0 = 1] \leq (1 - \mathrm{e}^{-k \cdot N/m})^k + \mathbf{negl}(\lambda)$. We modify G_0 to obtain G_1 by replacing the employed PRFs F and F_p with keys κ_X, κ_I with random functions. From the security of the PRFs against PPT adversaries and the fact that $|\mathsf{DB}|$ is polynomial in λ, we conclude that $\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1] \leq \mathbf{negl}(\lambda)$. We now find an upper bound on $\Pr[\mathsf{G}_1 = 1]$. By correctness of T-Set Γ, we know that the simulated server will retrieve the correct set $\mathsf{DB}(w_1)$ of id's matching the $\mathbf{sterm}$. Also, for all $\mathsf{id} \in \mathsf{DB}(w_1)$, which match the query (i.e $\mathsf{id} \in \mathsf{DB}(w_i)$ for $2 \leq i \leq n$), the encrypted Bloom filter will have 1's in positions $u_j = H_j\left(g^{F_p(\kappa_X, w_i) \cdot F_p(\kappa_I, \mathsf{id})}\right)$, so by correctness of HVE, the client result set S contains the desired result set $\mathsf{DB}(\psi(\mathbf{w}))$. Hence the game can only be won due to false positives, i.e. $\mathsf{id} \in \mathsf{DB}(w_1)$ which does not match the query (i.e $\mathsf{id} \notin \mathsf{DB}(w_i)$ for some $2 \leq i \leq n$) but is still returned by the server. By correctness of HVE, such false positives can happen only if the encrypted Bloom filter will have 1's in positions $u_j = H_j(g^{F_p(\kappa_X, w_i) \cdot F_p(\kappa_I, \mathsf{id})})$ for such $\mathsf{id} \notin \mathsf{DB}(w_i)$. There are two subcases: The first is that $g^{F_p(\kappa_X, w_i) \cdot F_p(\kappa_I, \mathsf{id})} = g^{F_p(\kappa_X, w') \cdot F_p(\kappa_I, \mathsf{id}')}$ for some other $(\mathsf{id}', w') \neq (\mathsf{id}, w)$ and $\mathsf{id}' \in \mathsf{DB}(w')$. This happens with negligible probability $\mathcal{O}(N^4/p)$; The second subcase is that (id, w_i) is a false positive for the Bloom filter, but this happens with probability $\leq (1 - \mathrm{e}^{-k \cdot N/m})^k$ for each $\mathsf{id} \in \mathsf{DB}(w_1)$ and hence by a union bound with overall probability $\leq |\mathsf{DB}(w_1)| \cdot (1 - \mathrm{e}^{-k \cdot N/m})^k$. We conclude that $\Pr[\mathsf{G}_1 = 1] \leq |\mathsf{DB}(w_1)| \cdot (1 - \mathrm{e}^{-k \cdot N/m})^k + \mathbf{negl}(\lambda)$, as required. □

Discussion. Currently, there is no efficient solution to handle the false positives from the Bloom filter. However, we consider this error is negligible under the conjunctive query setting: The false positive rate of getting a wrong result after querying $w_1 \wedge \cdots \wedge w_n$ is ${P_e}^{n-1}$. Also, as shown in [19], XSet also incurs a negligible false positive rate, and it is not considered in OXT. Thus, we still deem the Bloom filter as a memory-efficient and error-free way to keep XSet in HXT.

5.5 Security

A Searchable Symmetric Encryption (SSE) query consists of a Boolean formula ψ and a tuple $\mathbf{w} = (w_1, \ldots, w_n)$ of keywords. Throughout the chapter, we only consider conjunctive queries with $\psi(\mathbf{w}) = w_1 \wedge \cdots \wedge w_n$. Without loss of generality, we assume that w_1 is $\mathbf{sterm}$ and $(w_2, \ldots, w_n)$ are $n - 1$ $\mathbf{xterms}$. For a vector of queries $\mathbf{q} = (\mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$, it consists of a vector s of $\mathbf{sterms}$, and a sequence of vectors $\mathsf{x}_2, \ldots, \mathsf{x}_n$ of $\mathbf{xterms}$.

We define the leakage function of HXT $\mathcal{L}_{\mathsf{HXT}}(\mathsf{DB}, \mathbf{q})$ as a tuple $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{WRP}, \mathsf{IP})$ formed as follows:

- $N = \sum_{i=1}^{d} |\mathsf{W}_i|$ is the total number of appearances of keywords in documents.

- EP is the equality pattern of $\mathsf{s} \in \mathsf{W}^Q$ indicating which queries have the equal $\mathbf{sterms}$. In particular, $\mathsf{EP}[i] = |\{\mathsf{s}[1], \ldots, \mathsf{s}[j]\}|$, where j is the least index for which $\mathsf{s}[j] = \mathsf{s}[i]$. Note that $\mathsf{EP} \in [\omega]^Q$ and it is leaked since the client sends stag corresponding to s to server.
- SP is the size pattern of the queries, i.e. the number of documents matching the $\mathbf{sterm}$ in each query. Formally, $\mathsf{SP} \in [\omega]^Q$ and $\mathsf{SP}[i] = |\mathsf{DB}(\mathsf{s}[i])|$.
- WRP is the whole result pattern, which is an array computed as follows: $\mathsf{WRP}[i] = \mathsf{DB}(\mathsf{s}[i]) \cap_{j=2}^{n} \mathsf{DB}(\mathsf{x}_j[i])$ [2].
- IP is the conditional intersection pattern, which is a 4-dimensional table $\mathsf{IP}[i, j, \alpha, \beta]$ defined as follows:

$$= \begin{cases} \mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j]) & \text{if } i \neq j, \alpha \neq \beta, \\ & \text{and } \mathsf{x}_\alpha[i] = \mathsf{x}_\beta[j], \\ \emptyset & \text{otherwise,} \end{cases}$$

  for $1 \leq i, j \leq Q$ and $1 \leq \alpha, \beta \leq n$.

The overall leakage function $\mathcal{L}$ consists of the leakage from the HXT protocol $\mathcal{L}_{\mathsf{HXT}}$ and the leakage function of T-set $\mathcal{L}_{\mathbf{T}}$.

5.5.1 Leakage Function Comparison

Note that WRP is a new component in our leakage function compared to OXT. In their leakage function, they actually had $\mathsf{KPRP}[i] = \cup_{j=2}^{n} (\mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{x}_j[i]))$, which is denoted by RP in [19] and obviously a lot more than WRP. The following example illustrates the impact of the updated leakage component from KPRP to WRP.

Suppose that a database consists of 6 documents labelled by $\{\mathsf{id}_i\}_{1 \leq i \leq 6}$. Let us assume the following database 'forward index', listing document id's and keywords contained in each:

id	keywords	id	keywords
1	w_1, w_2, w_6, w_7, w_8	4	w_1, w_2, w_3
2	w_2, w_3, w_4, w_5	5	w_1, w_3, w_6
3	w_4, w_5, w_6, w_7	6	w_2, w_3, w_7

Consider the query $w_1 \wedge w_2 \wedge w_3$ for some keywords w_1, w_2, and w_3. By convention, we let w_1 be the least frequent keyword amongst all queried words. The inverted-index listing the document id's containing each of the queried words are $\mathsf{DB}(w_1) = \{\underline{\mathsf{id}_1}, \underline{\mathsf{id}_4}, \underline{\mathsf{id}_5}\}$, $\mathsf{DB}(w_2) = \{\underline{\mathsf{id}_1}, \mathsf{id}_2, \underline{\mathsf{id}_4}, \mathsf{id}_6\}$, and $\mathsf{DB}(w_3) = \{\mathsf{id}_2, \underline{\mathsf{id}_4}, \underline{\mathsf{id}_5}, \mathsf{id}_6\}$.

The KPRP leakage component in OXT is computed as follows: $\mathsf{RP} =$

[2] WRP is defined under the "no false positive" assumption, i.e. Both TSet and Bloom filter do not have a false positive rate. Otherwise, the false positive should be taking into account to define a refined leakage profile $\mathsf{FP\text{-}WRP}$.

Table 5.2: Leakage comparison for query $w_1 \wedge w_2 \wedge w_3$ between KPRP and WRP.

Leakage Component	Leaked Entries
KPRP (from OXT)	$\{(\mathsf{id}_1, w_2), (\mathsf{id}_4, w_2), (\mathsf{id}_4, w_3), (\mathsf{id}_5, w_3)\}$
WRP (from HXT)	$\{(\mathsf{id}_4, w_2), (\mathsf{id}_4, w_3)\}$

$\cup_{j=2}^{3} (\mathsf{DB}(w_1) \cap \mathsf{DB}(w_j)) = \{\mathsf{id}_1, \mathsf{id}_4\} \cup \{\mathsf{id}_4, \mathsf{id}_5\}$. As shown in Table 5.2, the KPRP leakage thus reveals 4 entries of the inverted-index, underlined in the inverted-index above. However, in this chapter, we eliminate the 'partial query' (KPRP) leakage induced by RP, leaving only the *whole result pattern* (WRP) corresponding to the final query result. By definition, we have $\mathsf{WRP} = \cap_{j=1}^{3} \mathsf{DB}(w_j)$. In our example, WRP reveals the exact result of the query, that is $\{\mathsf{id}_4\}$. Table 5.2 shows that the WRP only reveals two entries in the inverted-index above, in contrast to 4 entries revealed by KPRP above.

In fact, [19] has not noticed this leakage in their analysis and simply put RP to be the set of all identifiers matching the i-th query.

5.5.2 Security Analysis of HXT

Here, we show the security of our protocol against both a non-adaptive and an adaptive adversarial server which is assumed to be "honest-but-curious". Similar to [19], we first give a theorem about the security against non-adaptive attacks and later discuss the full security. For the sake of simplicity, we assume in our proof that no false positive happens in our protocol HXT [3] (i.e., no false positive happening in both TSet and BF).

Theorem 9. *Our protocol* HXT *is $\mathcal{L}$-semantically secure against non-adaptive attacks where $\mathcal{L}$ is the leakage function defined as before, assuming that the* DDH *assumption holds in $\mathbb{G}$, that F and F_p are secure* PRF*s, that* HVE *is a selectively simulation-secure protocol, that* $\mathsf{Sym} = (\mathsf{Sym.Enc}, \mathsf{Sym.Dec})$ *is an* IND-CPA *secure symmetric encryption scheme, that Γ is a $\mathcal{L}_{\mathbf{T}}$-secure and computationally correct T-set instantiation, and that no false positive happens in our* HXT*.*

Proof. First of all, we describe that leakage function $\mathcal{L}$, which consists of two components: $\mathcal{L}_{\mathsf{HXT}}$, the leakage from the HXT protocol, and $\mathcal{L}_{\mathbf{T}}$, the T-set leakage function. On input a database DB and a set of search queries $(\mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n) \in \mathsf{W}^n$, the leakage function $\mathcal{L}$ can be computed similarly as in [19]: For every $w \in \mathsf{W}$, it randomly chooses a key $\kappa \in \{0,1\}^\lambda$ and initiates $\mathbf{t}$ as an empty vector; For a counter c, it chooses a non-zero random $y \in \mathbb{Z}_p$ and computes an encryption of constant string $\mathsf{Sym.Enc}(\kappa, 0^\lambda)$ and puts this ciphertext along with y in the c-th component of $\mathbf{t}$; Once the counter reaches its end ($T_w = |\mathbf{T}[w]|$), it puts $\mathbf{t}$ into the w-th entry of $\mathbf{T}$. Then the leakage function is output as $\big((\mathcal{L}_{\mathsf{HXT}}(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)), \mathcal{L}_{\mathbf{T}}(\mathbf{T}, \mathsf{s}), \mathbf{T}[\mathsf{s}]\big)$.

Next, we show the proof of Theorem 9. The proof is structured through a sequence of games. In all games, the adversary supplies a database DB and a list of search queries

[3] Note that the assumption can be relaxed by taking into account the indices introduced due to the false positive.

Algorithm 14 Game$_0$

$(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d} \leftarrow \mathsf{DB}$; $\kappa_S, \kappa_I, \kappa_Z, \kappa_X \xleftarrow{\$} \{0,1\}^\lambda$
for $w \in \mathsf{W}$ **do**
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_w}) \leftarrow \mathsf{DB}(w)$; $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  $\mathbf{t} \leftarrow \{\}$; $\kappa_e \leftarrow F(\kappa_S, w)$
  for $c = 1 : T_w$ **do**
    $\mathsf{xid} \leftarrow F_p(\kappa_I, \bar{\mathsf{id}}_{\sigma(c)})$; $e = \mathsf{Sym.Enc}(\kappa_e, \bar{\mathsf{id}}_{\sigma(c)})$
    $z \leftarrow F_p(\kappa_Z, w||c)$; $y \leftarrow \mathsf{xid} \cdot z^{-1} \pmod p$; $\mathbf{t}[c] \leftarrow (y, e)$
  end for
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
end for
$(\mathsf{TSet}, \kappa_T) \leftarrow \mathsf{TSet.Setup}(\mathbf{T})$
for $i = 1 : Q$ **do** $\mathsf{STags}[i] \leftarrow \mathsf{TSet.GenTag}(\kappa_T, \mathsf{s}[i])$ **end for**
$\mathsf{BF} \leftarrow 0^m$
for $w \in \mathsf{W}$ **do**
  $\eta \leftarrow F_p(\kappa_X, w)$
  for $\mathsf{id} \in \mathsf{DB}(w)$ **do**
    $\mathsf{xid} \leftarrow F_p(\kappa_I, \mathsf{id})$
    for $j = 1 : k$ **do**
      $h_j(\mathsf{id}, w) \leftarrow H_j\left(g^{\eta \cdot \mathsf{xid}}\right)$; $\mathsf{BF}[h_j(\mathsf{id}, w)] \leftarrow 1$
    end for
  end for
end for
$\mathbf{c} \leftarrow \mathsf{HVE.Enc}(msk, \mu = \text{'True'}, \mathsf{BF})$; $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\mathcal{E}[i] \leftarrow \{\}$
  for $c = 1 : T$ **do**
    $(y_c, e_c) \leftarrow \mathbf{t}$; $z_c \leftarrow F_p(\kappa_Z, \mathsf{s}[i]||c)$; $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      $\eta_\ell \leftarrow F_p(\kappa_X, \mathsf{x}_\ell[i])$; $\mathsf{xtoken}[c, \ell] \leftarrow g^{z_c \cdot \eta_\ell}$
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j\left(\mathsf{xtoken}[c, \ell]^{y_c}\right)] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathsf{HVE.KeyGen}(msk, \mathbf{v}_c)$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \text{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]$; $\mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

$\mathbf{q} = (\mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ at the beginning, where s and x_i are the list of query sterms and xterms, respectively. The first game Game_0 is designed to have the same distribution as $\mathbf{Real}$, where we neglect all false positives of both TSet and BF for simplifying the proof, and the last one can be easily simulated by an efficient algorithm $\mathcal{S}_{\mathsf{HXT}}$. By showing that the distributions of each two successive games are (computationally) indistinguishable, we get the simulator $\mathcal{S}_{\mathsf{HXT}}$ that meets the requirements of the security definition, thus completing the proof of the theorem. In the following, we use $\Pr[\mathsf{G}_i = 1]$ to denote the probability that Game_i outputs 1.

Game$_0$: this game is slightly modified from the real game to make the analysis easier, the details of which are shown in Algorithm 14. With $(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ as input, the game starts to simulate encrypted database $\mathsf{EDB}(1)$, then it computes a vector of size Q of stags called STags. Particularly $\mathsf{STags}[i] \leftarrow \mathsf{TSet.GenTag}(\kappa_\mathsf{T}, \mathsf{s}[\mathsf{i}])$, for $1 \le i \le Q$. Second, it computes BF similar to that of Algorithm 12 and finally inputs $\mathbf{c}$ into $\mathsf{EDB}(2)$. It finally computes the transcript array $\mathbf{tr}$, with $\mathbf{tr}[i]$ being $(\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[\mathsf{i}], \mathsf{xtoken}[i], \mathsf{token}[i])))$ for $1 \le i \le Q$, by running the last loop of Algorithm 14. Note that the obtained $\mathrm{ResInds}$ in $\mathbf{t}$ is computed by looking up the corresponding id values in $\mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$, instead of decrypting the results returned by SE.Search in the real game. Assuming no false positives happening, the

Algorithm 15 $\mathbf{Game}_1$ and $\boxed{\mathbf{Game}_2}$

$(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d} \leftarrow \mathsf{DB}$; $f_S, f_I, f_Z, f_X \xleftarrow{\$} \mathsf{Func}(\{0,1\}^\lambda, \mathbb{Z}_p^*)$
for $w \in \mathsf{W}$ **do**
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_w}) \leftarrow \mathsf{DB}(w)$; $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  $\mathbf{t} \leftarrow \{\}$; $\kappa_e \xleftarrow{\$} \{0,1\}^\lambda$
  for $c = 1 : T_w$ **do**
    $\mathsf{xid} \leftarrow f_I(\bar{\mathsf{id}}_{\sigma(c)})$; $e = \mathsf{Sym.Enc}(\kappa_e, \bar{\mathsf{id}}_{\sigma(c)})$
    $\boxed{e = \mathsf{Sym.Enc}(\kappa_e, 0^\lambda)}$
    $z \leftarrow f_Z(w||c)$; $y \leftarrow \mathsf{xid} \cdot z^{-1} \pmod p$; $\mathbf{t}[c] \leftarrow (y, e)$
  end for
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
end for
$(\mathsf{TSet}, \kappa_T) \leftarrow \mathsf{TSet.Setup}(\mathbf{T})$
for $i = 1 : Q$ **do** $\mathsf{STags}[i] \leftarrow \mathsf{TSet.GenTag}(\kappa_T, \mathsf{s}[i])$ **end for**
$\mathsf{BF} \leftarrow 0^m$
for $w \in \mathsf{W}$ **do**
  $\eta \leftarrow f_X(w)$
  for $\mathsf{id} \in \mathsf{DB}(w)$ **do**
    $\mathsf{xid} \leftarrow f_I(\mathsf{id})$
    for $j = 1 : k$ **do**
      $h_j(\mathsf{id}, w) \leftarrow H_j\left(g^{\eta \cdot \mathsf{xid}}\right)$; $\mathsf{BF}[h_j(\mathsf{id}, w)] \leftarrow 1$
    end for
  end for
end for
$\mathbf{c} \leftarrow \mathsf{HVE.Enc}(msk, \mu = \text{'True'}, \mathsf{BF})$; $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\mathcal{E}[i] \leftarrow \{\}$
  for $c = 1 : T$ **do**
    $(y_c, e_c) \leftarrow \mathbf{t}$; $z_c \leftarrow f_Z(\mathsf{s}[i]||c)$; $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      $\eta_\ell \leftarrow f_X(\mathsf{x}_\ell[i])$; $\mathsf{xtoken}[c, \ell] \leftarrow g^{z_c \cdot \eta_\ell}$
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j\left(\mathsf{xtoken}[c, \ell]^{y_c}\right)] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathsf{HVE.KeyGen}(msk, \mathbf{v}_c)$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \text{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]$; $\mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

distribution of the explained game is exactly the same as the real game. Therefore, $\Pr[\mathsf{G}_0 = 1] \leq \Pr[\mathsf{Real}_{\mathcal{A}}^{\mathsf{HXT}}(\lambda) = 1] + \mathbf{negl}(\lambda)$.

$\mathbf{Game}_1$: in this game, we replace the PRFs F and F_p with random functions. The details of which are shown in Algorithm 15. Note that since $F(\kappa_S, \cdot)$ is only evaluated on the same input once, its evaluations can be replaced with random selections from the appropriate range. As to $F_p(\kappa_X, \cdot)$, $F_p(\kappa_I, \cdot)$ and $F_p(\kappa_Z, \cdot)$, they are replaced by f_X, f_I and f_Z, respectively. A standard hybrid argument implies that there exist efficient adversaries $\mathcal{B}_{1,1}$ and $\mathcal{B}_{1,2}$ such that $\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1] \leq \mathcal{A}_{F,\mathcal{B}_{1,1}}^{\mathsf{PRF}}(\lambda) + 3\mathcal{A}_{F_p,\mathcal{B}_{1,2}}^{\mathsf{PRF}}(\lambda)$.

$\mathbf{Game}_2$: this game replaces only the encryption of document identifiers with that of constant string 0^λ. In the game, the encryption is operated for polynomial, say $\mathsf{poly}(\lambda)$ times, so by a standard hybrid argument we can see that the indistinguishability between these two games can be reduced to IND-CPA security of the symmetric encryption. That is, there exists an efficient adversary $\mathcal{B}_2$, such that $\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_1 = 1] \leq \mathsf{poly}(\lambda) \cdot \mathcal{A}_{\mathcal{B}_2,\mathsf{Sym}}^{\mathsf{IND-CPA}}(\lambda)$.

$\mathbf{Game}_3$: in this game, BF and xtoken are generated in an alternative but equivalent way,

Algorithm 16 $\mathbf{Game}_3$, $\boxed{\mathbf{Game}_4}$, and $\boxed{\boxed{\mathbf{Game}_5}}$

$(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d} \leftarrow \mathsf{DB}$; $f_S, f_I, f_Z, f_X \xleftarrow{\$} \mathsf{Func}(\{0,1\}^\lambda, \mathbb{Z}_p^*)$
for $w \in \mathsf{W}$ **do**
  for $\mathsf{id} \in \mathsf{DB}(w)$ **do**
    $\eta \leftarrow f_X(w)$; $X[w] \leftarrow g^\eta$; $\mathsf{xid} \leftarrow f_I(\mathsf{id})$; $A[w, \mathsf{id}] \leftarrow X[w]^{\mathsf{xid}}$
    $\boxed{\boxed{A[w, \mathsf{id}] \xleftarrow{\$} \mathbb{G}}}$
  end for
end for
for $w \in \mathsf{W}$ **do**
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_w}) \leftarrow \mathsf{DB}(w)$; $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  $\mathbf{t} \leftarrow \{\}$; $\kappa_e \xleftarrow{\$} \{0,1\}^\lambda$
  for $c = 1 : T_w$ **do**
    $\mathsf{xid} \leftarrow f_I(\bar{\mathsf{id}}_{\sigma(c)})$; $e = \mathsf{Sym.Enc}(\kappa_e, 0^\lambda)$
    $z \leftarrow f_Z(w||c)$; $y \leftarrow \mathsf{xid} \cdot z^{-1} \pmod p$
    $\boxed{y \xleftarrow{\$} \mathbb{Z}_p^*}$
    $\mathbf{t}[c] \leftarrow (y, e)$
  end for
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
  for $u \in \mathsf{W} \setminus \{w\}$ **do**
    for $c = T_w + 1, \ldots, T$ **do**
      $B[w, u, c] \leftarrow X[u]^{f_Z(w||c)}$
      $\boxed{\boxed{B[w, u, c] \xleftarrow{\$} \mathbb{G}}}$
    end for
  end for
end for
$(\mathsf{TSet}, \kappa_T) \leftarrow \mathsf{TSet.Setup}(\mathbf{T})$
for $i = 1 : Q$ **do** $\mathsf{STags}[i] \leftarrow \mathsf{TSet.GenTag}(\kappa_T, \mathsf{s}[i])$ **end for**
$\mathsf{BF} \leftarrow 0^m$
for $w \in \mathsf{W}$ **do**
  for $\mathsf{id} \in \mathsf{DB}(w)$ **do**
    for $j = 1 : k$ **do**
      $h_j(\mathsf{id}, w) \leftarrow H_j\,(A[w, \mathsf{id}])$; $\mathsf{BF}[h_j(\mathsf{id}, w)] \leftarrow 1$
    end for
  end for
end for
$\mathbf{c} \leftarrow \mathsf{HVE.Enc}(msk, \mu = \text{'True'}, \mathsf{BF})$; $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\mathcal{E}[i] \leftarrow \{\}$
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_s}) \leftarrow \mathsf{DB}[\mathsf{s}[i]]$; $\sigma \leftarrow \mathsf{WPerms}[\mathsf{s}[i]]$
  for $c = 1 : T$ **do**
    $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      if $c \le T_s$ **then**
        $(y_c, e_c) \leftarrow \mathbf{t}[c]$; $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c}$
      else
        $\mathsf{xtoken}[c, \ell] \leftarrow B[\mathsf{s}[i], \mathsf{x}_\ell[i], c]$
      end if
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j\,(\mathsf{xtoken}[c, \ell]^{y_c})] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathsf{HVE.KeyGen}(msk, \mathbf{v}_c)$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \text{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]$; $\mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

which is shown in Algorithm 16. Loosely speaking, all possible values $g^{f_X(w) f_I(\mathsf{id})}$ for each identifier $\mathsf{id} \in \mathsf{DB}(w)$ and keyword $w \in \mathsf{W}$ are pre-computed and stored in an array A. Moreover, some xtoken values in transcripts, which correspond to impossible matches, are generated and stored in another array B.

Then arrays A and B are used to compute BF and xtoken. In particular, for a given w and $\mathsf{id} \in \mathsf{DB}(w)$ the element $A[w, \mathsf{id}]$ instead of $g^{f_X(w)f_I(\mathsf{id})}$ is added to BF. Note that $A[w, \mathsf{id}]$ is exactly the value $g^{f_X(w)f_I(\mathsf{id})}$, so BF is the same as in the previous game. In addition, it is easy to see that the transcript $\mathbf{tr}$[i] will be the same only if $\mathsf{xtoken}[i]$ and $\mathsf{token}[i]$ are the same in both games. We note that $\mathsf{token}[i]$ depends on $\mathsf{xtoken}[i]$, so we only focus on the generation of $\mathsf{xtoken}[i]$ array in the following.

In Game_2, the $\mathsf{xtoken}[c, \ell]$ for the ℓ-th $\mathbf{xterm}$ $\mathsf{x}_\ell[i]$ (of the i-th query) and $c \in [T]$ is set to be $g^{f_Z(\mathsf{s}[i]||c)\cdot f_X(\mathsf{x}_\ell[i])}$. In the current game, however, $\mathsf{xtoken}[c, \ell]$ is generated by first looking up $\mathsf{DB}[\mathsf{s}[i]] = (\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_s})$, $\mathsf{WPerms}[\mathsf{s}[i]] = \sigma$ and $\mathbf{t}$, where $\mathbf{t} = (f_I(\bar{\mathsf{id}}_{\sigma(c)})/f_Z(\mathsf{s}[i]||c), e_c)_{c\in[T_s]}$ by the correctness of TSet. Then for $c \in [T_s]$ and $\ell \in [2, n]$, it retrieves (y_c, e_c), such that $y_c = f_I(\bar{\mathsf{id}}_{\sigma(c)})/f_Z(\mathsf{s}[i]||c)$, and sets $\mathsf{xtoken}[c, \ell]$ to be $A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c} = g^{f_Z(\mathsf{s}[i]||c)\cdot f_X(\mathsf{x}_\ell[i])}$. For $c \in [T] \setminus [T_s]$, $\mathsf{xtoken}[c, \ell]$ is set to be $B[\mathsf{s}[i], \mathsf{x}_\ell[i], c] = g^{f_X(\mathsf{x}_\ell[i])\cdot f_Z(\mathsf{s}[i]||c)}$.

It is easy to observe from the above that the $\mathsf{xtoken}[c, \ell]$ is exactly the same as in Game_2. Therefore, we have $\Pr[\mathsf{G}_3 = 1] = \Pr[\mathsf{G}_2 = 1]$.

Game$_4$: this game is almost identical to the previous one, except that the single boxed code in Algorithm 16 is also included: the values y are now drawn randomly from $\mathbb{Z}_p^*$. Due to the modifications made in Game_3, the random function f_Z is chosen during the first steps of the algorithm and never evaluated again later, so z is uniformly and independently distributed. Moreover, since $y = \mathsf{xid} \cdot z^{-1}$, for any $w \in \mathsf{W}$ and $c \in [T_w]$, the value of y is also uniform and independent of the rest of the randomness. Thus replacing y with random values does not affect the distribution of the resulted game, so we have $\Pr[\mathsf{G}_4 = 1] = \Pr[\mathsf{G}_3 = 1]$.

Game$_5$: this game is similar to the previous game, except that it also includes the doubly boxed code in Algorithm 16. That is, all the values of A and B arrays are selected at random from $\mathbb{G}$. Under the DDH assumption, there exists an efficient algorithm $\mathcal{B}_3$ such that $\Pr[\mathsf{G}_5 = 1] - \Pr[\mathsf{G}_4 = 1] \leq \mathcal{A}^{\mathsf{DDH}}_{\mathbb{G},\mathcal{B}_3}(\lambda)$.

To show the indistinguishability between these two games, a simple reduction can be conducted similarly as in [19]. Briefly speaking, the values of X array in G_4 are the g^a values, and the X values are raised to the power of xid when computing A and to the power of $f_Z(w||c)$ when computing B, where xid and $f_Z(w||c)$ act as the b values of the DDH tuple. Thus, A and B in G_4 have values of the form g^{ab}, while in G_5 they are replaced with random values. Differentiating between them can be easily reduced to breaking the DDH assumption, we omit the details here.

Game$_6$: in this game, TSet is generated by using simulator $\mathcal{S}_{\mathbf{T}}$, which is shown in Algorithm 17. The existence of such a simulator is guaranteed by the security notion of T-Sets. In addition, we remove some irrelevant code (some selecting random functions), and other routines remained the same as G_5. Similar to the analysis shown in [19], there exists an efficient algorithm $\mathcal{B}_4$, under the security definition of TSet, such that $\Pr[\mathsf{G}_6 = 1] - \Pr[\mathsf{G}_5 = 1] \leq \mathcal{A}^{\mathsf{TSet}}_{\mathcal{B}_4}(\lambda)$.

Algorithm 17 : $\mathbf{Game}_6$ and $\boxed{\mathbf{Game}_7}$

$(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d} \leftarrow \mathsf{DB}$
for $w \in \mathsf{W}$ **and** $\mathsf{id} \in \mathsf{DB}(w)$ **do** $A[w, \mathsf{id}] \xleftarrow{\$} \mathbb{G}$ **end for**
for $w \in \mathsf{s}$ **do** $\mathsf{WPerms}[w] \leftarrow \mathsf{Perm}([T_s])$ **end for**
for $w \in \mathsf{W}$ **do**
  $\mathbf{t} \leftarrow \{\}$; $\kappa_e \xleftarrow{\$} \{0,1\}^\lambda$
  for $c = 1 : T_w$ **do** $e = \mathsf{Sym.Enc}(\kappa_e, 0^\lambda)$; $y \xleftarrow{\$} \mathbb{Z}_p^*$; $\mathbf{t}[c] \leftarrow (y, e)$ **end for**
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
  for $u \in \mathsf{W} \setminus \{w\}$ **do**
    for $c = T_w + 1, \ldots, T$ **do** $B[w, u, c] \xleftarrow{\$} \mathbb{G}$ **end for**
  end for
end for
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_{\mathbf{T}}(\mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$
$\mathsf{BF} \leftarrow 0^m$
for $w \in \mathsf{W}$ **do**
  for $\mathsf{id} \in \mathsf{DB}(w)$ **do**
    for $j = 1 : k$ **do**
      $h_j(\mathsf{id}, w) \leftarrow H_j\,(A[w, \mathsf{id}])\,;\, \mathsf{BF}[h_j(\mathsf{id}, w)] \leftarrow 1$
    end for
  end for
end for
$\mathbf{c} \leftarrow \mathsf{HVE.Enc}(msk, \mu = \text{'True'}, \mathsf{BF})$; $\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
$\boxed{\mathbf{c} \leftarrow \mathcal{L}_{\mathsf{HVE}}(\mu = \text{'True'})}$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\mathcal{E}[i] \leftarrow \{\}$
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_s}) \leftarrow \mathsf{DB}(\mathsf{s}[i])$; $\sigma \leftarrow \mathsf{WPerms}[\mathsf{s}[i]]$
  for $c = 1 : T$ **do**
    $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      if $c \le T_s$ **then**
        $(y_c, e_c) \leftarrow \mathbf{t}[c]$
        if $\bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$ **then** ▷ $\beta(\mathbf{v}_c, \mathsf{BF}) = 1 \Leftarrow P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF}) = 1$
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c}$
        else ▷ $\beta(\mathbf{v}_c, \mathsf{BF}) = 0 \Leftarrow P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF}) = 0$
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c}$
        end if
      else
        $\mathsf{xtoken}[c, \ell] \leftarrow B[\mathsf{s}[i], \mathsf{x}_\ell[i], c]$
      end if
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j\,(\mathsf{xtoken}[c, \ell]^{y_c})] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathsf{HVE.KeyGen}(msk, \mathbf{v}_c)$
    $\boxed{\alpha(\mathbf{v}_c) \leftarrow \{i \in [m] : \mathbf{v}_c[i] \neq 1\}; \beta(\mathbf{v}_c, \mathsf{BF}) \leftarrow P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF})}$
    $\boxed{\mathsf{token}_c[i] \leftarrow \mathcal{S}_{\mathsf{HVE}}(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))}$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \text{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]$; $\mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

$\mathbf{Game}_7$: this game is like the previous one, except that the boxed codes are also included in Algorithm 17. In this game, the second part of EDB (i.e., $\mathsf{EDB}(2) = \mathbf{c}$) and the search tokens $\mathsf{token}[i]$ are generated by running the simulator $\mathcal{S}_{\mathsf{HVE}}$ of HVE. To show the indistinguishability between Game_7 and Game_6, we let $\alpha(\mathbf{v}_c) = [m] \setminus \{H_j[\mathsf{xtoken}[c, \ell]^{y_c}]\}_{\ell \in [2,n]}^{j \in [1,k]}$ and $\beta(\mathbf{v}_c, \mathsf{BF}) = P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF})$. Now we consider the following adversary $\mathcal{B}_5$ against the selective simulation security of HVE. $\mathcal{B}_5$ starts to simulate $\mathrm{Game}_6/\mathrm{Game}_7$ by generating $\mathsf{TSet}, \mathsf{BF}$ and xtoken exactly as in Game_6, and then simulates $\mathbf{c}$ and token with the response from the real/ideal game of HVE. Note that, assuming no false positive happens, it holds that $P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF}) = 1$ *iff*

Algorithm 18 : $\mathbf{Game}_8$

$(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d} \leftarrow \mathsf{DB}$
for $w \in \mathsf{W}$ **and** $\mathsf{id} \in \mathsf{DB}(w)$ **do** $A[w, \mathsf{id}] \xleftarrow{\$} \mathbb{G}$ **end for**
for $w \in \mathsf{s}$ **do** $\mathsf{WPerms}[w] \leftarrow \mathsf{Perm}([T_s])$ **end for**
for $w \in \mathsf{W}$ **do**
  $\mathbf{t} \leftarrow \{\}$; $\kappa_e \xleftarrow{\$} \{0,1\}^\lambda$
  for $c = 1 : T_w$ **do** $e = \mathsf{Sym.Enc}(\kappa_e, 0^\lambda)$; $y \xleftarrow{\$} \mathbb{Z}_p^*$; $\mathbf{t}[c] \leftarrow (y, e)$ **end for**
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
end for
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_{\mathbf{T}}(\mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$
$\mathbf{c} \leftarrow \mathcal{S}_{\mathsf{HVE}}(\mu = \text{'True'})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\mathcal{E}[i] \leftarrow \{\}$
  $(\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_s}) \leftarrow \mathsf{DB}(\mathsf{s}[i])$; $\sigma \leftarrow \mathsf{WPerms}[\mathsf{s}[i]]$
  for $c = 1 : T$ **do**
    $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      if $c \leq T_s$ **then**
        $(y_c, e_c) \leftarrow \mathbf{t}[c]$
        if $\bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c}$
        else if $\exists j \neq i$ and $\nu \in [2, n] : \bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{DB}(\mathsf{s}[j]) \wedge \mathsf{x}_\ell[i] = \mathsf{x}_\nu[j]$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c}$
        else
          $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
        end if
      else
        $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
      end if
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j(\mathsf{xtoken}[c, \ell]^{y_c})] = 1$ **end for**
    end for
    $\alpha(\mathbf{v}_c) \leftarrow \{i \in [m] : \mathbf{v}_c[i] \neq 1\}$; $\beta(\mathbf{v}_c, \mathsf{BF}) \leftarrow P_{\mathbf{v}_c}^{\mathsf{HVE}}(\mathsf{BF})$
    $\mathsf{token}_c[i] \leftarrow \mathcal{S}_{\mathsf{HVE}}(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \text{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]$; $\mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

$\bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$, hence $\mathcal{B}_5$ can derive the input $(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$ of $\mathcal{S}_{\mathsf{HVE}}$ from BF and $\mathsf{xtoken}[c, \ell]$.

By the description of Game_6 and Game_7, we know that the real game of HVE with $\mathcal{B}_5$ perfectly simulates Game_6, while the ideal game with $\mathcal{B}_5$ perfectly simulates Game_7, so we have that $\Pr[\mathsf{G}_7 = 1] - \Pr[\mathsf{G}_6 = 1] \leq \mathcal{A}_{\mathcal{B}_5}^{\mathsf{HVE}}(\lambda)$.

$\mathbf{Game}_8$: To enable the final simulator to work well with its given leakage profile, the way of array A being accessed is changed to an alternative but equivalent way. We note that the array A in Game_7 is only accessed when generating the xtoken, and not ever used for producing $\mathbf{c}$ because the simulator of HVE does not receive the actual BF (as mentioned in the ideal game of HVE). More specifically, in this game, we replace with a random selection the access of array A during the generation of xtoken for the case of $\bar{\mathsf{id}}_{\sigma(c)} \notin \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$, except for the repeated accesses, which does not affect the distribution of xtoken. Note that, a repeated access to the same position of A happens only if it is called during two distinct search queries since computing xtoken for one single query touches only unique position of A. More precisely, for an element indexed by (id, w) to be accessed twice, it must hold that

Algorithm 19 : Simulator $\mathcal{S}_{\mathsf{HXT}}$

for $w \in \hat{\mathbf{x}}$ **and** $\mathsf{id} \in \bigcup_{i=1} (\mathsf{WRP}[i] \cup \bigcup_{j \neq i,\alpha,\beta} \mathsf{IP}[i,j,\alpha,\beta])$ **do**
  $A[\mathsf{id}, w] \xleftarrow{\$} \mathbb{G}$
end for
for $w \in \mathsf{EP}$ **do** $\mathrm{WPerms}[w] \xleftarrow{\$} \mathrm{Perm}([\mathsf{SP}[i]])$ **end for**
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_{\mathbf{T}}(\mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$
$\mathbf{c} \leftarrow \mathcal{S}_{\mathsf{HVE}}(\mu = \text{'True'})$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i]); \sigma \leftarrow \mathsf{WPerms}[\mathsf{EP}[i]]; \mathcal{E}[i] \leftarrow \{\}$
  $R[i] \leftarrow \mathsf{WRP}[i] \cup \bigcup_{j \neq i,\alpha,\beta} \mathsf{IP}[i,j,\alpha,\beta]; T' \leftarrow |R[i]|$
  $(\bar{\mathsf{id}}_1, \bar{\mathsf{id}}_2, \cdots, \bar{\mathsf{id}}_{T'}, \underbrace{\perp, \cdots, \perp}_{\mathsf{SP}[i]-T'}) \leftarrow \mathsf{DB}[\mathsf{EP}[i]]$
  for $c = 1 : T$ **do**
    $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      if $c \leq \mathsf{SP}[i]$ **then**
        $(y_c, e_c) \leftarrow \mathbf{t}[c]$
        if $\bar{\mathsf{id}}_{\sigma(c)} \neq \perp \wedge \bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{WRP}[i]$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \hat{\mathbf{x}}[i, \ell]]^{1/y_c}$
        else if $\bar{\mathsf{id}}_{\sigma(c)} \neq \perp \wedge \bar{\mathsf{id}}_{\sigma(c)} \in \bigcup_{j \neq i,\nu} \mathsf{IP}[i,j,\ell,\nu]$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \hat{\mathbf{x}}[i, \ell]]^{1/y_c}$
        else
          $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
        end if
      else
        $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
      end if
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j(\mathsf{xtoken}[c, \ell]^{y_c})] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathcal{S}_{\mathsf{HVE}}(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \mathrm{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathrm{Res} \leftarrow \mathcal{E}[i]; \mathrm{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathrm{Res}, \mathrm{ResInds}, ((\mathsf{STags}[\mathrm{i}], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

$\mathsf{id} \in \mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j])$ for some $i \neq j$ and $\mathsf{x}_\alpha[i] = \mathsf{x}_\beta[j]$ for some $\alpha, \beta \in [2, n]$. The condition for such a repeated access is exactly captured by the third "if" statement in the last loop of this game (exactly, the IP leakage component). If this condition does not apply, the xtoken is randomly selected from $\mathbb{G}$. Furthermore, it is easy to observe that both token and $\mathbf{c}$ rely heavily on $(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$ which are derived from xtoken, so we have that $\Pr[\mathsf{G}_8 = 1] = \Pr[\mathsf{G}_7 = 1]$.

Simulator: In the following, we present a simulator $\mathcal{S}_{\mathsf{HXT}}$ that takes as input the leakage profile $\mathcal{L}(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ consisting of $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{WRP}, \mathsf{IP}, \mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$[4] and outputs a simulated EDB and $\mathbf{tr}$. By showing that the simulator produces the same distribution as Game_8 and then combining the relations between the games, we get the simulator satisfying the requirements in Theorem 9.

First of all, our simulator $\mathcal{S}_{\mathsf{HXT}}$ will compute a restricted equality pattern of $\mathsf{x} \doteq (\mathsf{x}_2, \ldots, \mathsf{x}_n)$ as below, denoted by $\hat{\mathbf{x}}$. Then it proceeds to produce its final output through Algorithm 19. The restricted equality pattern $\hat{\mathbf{x}}$ can be computed as follows in terms of the leakage IP by defining a table such that $\hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta]$ iff $\mathsf{IP}[t_1, t_2, \alpha, \beta] \neq \emptyset$. The table $\hat{\mathbf{x}}$ describes which

[4]Note that $\mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s})$ and $\mathbf{T}[\mathsf{s}]$ are computed in the same way as [19].

xterms are "known" to be equal by the adversarial server. In particular, we have that

$$\hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta] \implies \mathsf{x}[t_1, \alpha] = \mathsf{x}[t_2, \beta], \text{ and} \tag{5.3}$$

$$(\mathsf{x}[t_1, \alpha] = \mathsf{x}[t_2, \beta]) \wedge (\mathsf{DB}(\mathsf{s}[t_1]) \cap \mathsf{DB}(\mathsf{s}[t_2]) \neq \emptyset)$$

$$\implies \hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta]. \tag{5.4}$$

Taking as input the leakage profile $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{WRP}, \mathsf{IP}, \mathcal{L}_{\mathbf{T}}(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$ and the restricted equality pattern $\hat{\mathbf{x}}$ computed as above, the simulator then works as in Algorithm 19 to generate the $\mathsf{EDB} = (\mathsf{TSet}, \mathbf{c})$ and the transcript $\mathbf{tr}$.

In the simulation, array A is only filled out for positions $w \in \hat{\mathbf{x}}$ and $\mathsf{id} \in \bigcup_{i=1} \big(\mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta]\big)$, which is used to keep the reuse pattern of A during the generation of xtoken. Similarly, the permutations σ's are assigned with respect to EP, the repetition of which captures that of sterms s. When computing the transcript $\mathbf{tr}[i]$ for the i-th query, the simulator sets the "revealed" indices for that query as $R[i] \leftarrow \mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta]$ and puts them in canonical order, calling them $\bar{\mathsf{id}}_1, \bar{\mathsf{id}}_2, \cdots, \bar{\mathsf{id}}_{|R[i]|}$. Since $R[i] \subseteq \mathsf{DB}(\mathsf{s}[i])$, the simulator then pads $R[i]$ up to size $\mathsf{SP}[i]$ by setting $\bar{\mathsf{id}}_k$ for $k \in [|R[i]|, \mathsf{SP}[i]]$ to be dummy symbols $\perp$. After that, the simulator uses $\mathsf{SP}, \mathsf{WRP}, \mathsf{IP}$ to simulate xtoken as described in Algorithm 19.

Next, we show the output of the simulator $\mathcal{S}_{\mathsf{HXT}}$ is identically distributed as that of Game_8. It is clear that the distributions of $\mathbf{t}, (y_c, e_c)$ are identical to Game_8, as $(\mathsf{TSet}, \mathsf{STags})$ are computed exactly in the same way. In addition, the permutations σ's have the same distribution since they are chosen uniformly at random and reused in the same pattern in both cases. Moreover, we can see that identifiers in $\mathsf{DB}(\mathsf{s}[i])/\mathsf{DB}(\mathsf{SP}[i])$ are used in the random order determined by σ, except identifiers not appearing as relevant results are padded with dummy symbols in $\mathsf{DB}(\mathsf{SP}[i])$, and that they follow the same logic in both Game_8 and the simulated game (cf. Algorithm 19): if $\sigma(c)$-th identifier is in either $\mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$ or the set of identifiers containing the sterm of another query with some same xterm, then the corresponding position of A is accessed; otherwise, a random group element is selected. At last, what we need to do is to show the accessed entries from A follow the same repetition in both games, which is analyzed as below.

Suppose that $(\mathsf{id}_1, \mathsf{x}_\ell[i])$ and $(\mathsf{id}_2, \mathsf{x}_\nu[j])$ are any two identifier/key-word pair accessed from A in Game_8. Then the simulator $\mathcal{S}_{\mathsf{HXT}}$ will read the positions $(\mathsf{id}_1, \hat{\mathbf{x}}[i, \ell])$ and $(\mathsf{id}_2, \hat{\mathbf{x}}[j, \nu])$ instead. To show the simulation is identical to Game_8, next we argue that

$$(\mathsf{id}_1, \mathsf{x}_\ell[i]) = (\mathsf{id}_2, \mathsf{x}_\nu[j]) \iff (\mathsf{id}_1, \hat{\mathbf{x}}[i, \ell]) = (\mathsf{id}_2, \hat{\mathbf{x}}[j, \nu]).$$

Obviously, the $\Leftarrow$ direction follows readily from (5.3). As to the other direction, we know

that $\mathsf{id}_1 = \mathsf{id}_2$ are members of the following set

$$\big(\mathsf{WRP}[i] \cup \bigcup_{k \neq i,\alpha,\beta} \mathsf{IP}[i,k,\alpha,\beta]\big) \cap \big(\mathsf{WRP}[j] \cup \bigcup_{k \neq j,\alpha,\beta} \mathsf{IP}[j,k,\alpha,\beta]\big),$$

as the games only use identifiers from these sets when computing xtoken. This indicates that $\mathsf{id} \doteq \mathsf{id}_1 = \mathsf{id}_2$ belongs to $\mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j])$, and thus we can get from (5.4) that $\hat{\mathbf{x}}[i,\ell] = \hat{\mathbf{x}}[j,\nu]$.

Finally, regarding the distributions of output $\mathbf{c}$ and token, they rely heavily on the distribution of xtoken and can be simulated by running $\mathcal{S}_{\mathsf{HVE}}$ with $\alpha(\mathbf{v}_c)$ and $\beta(\mathbf{v}_c, \mathsf{BF})$ as input. Recall that $\alpha(\mathbf{v}_c)$ and $\beta(\mathbf{v}_c, \mathsf{BF})$ can be derived from xtoken and WRP directly. Up to now, we get that $\mathcal{S}_{\mathsf{HXT}}$ perfectly simulates Game_8 with its leakage. □

We now show that our theorem is also valid for adaptive models.

Theorem 10. *Our protocol* HXT *is $\mathcal{L}$-semantically secure against adaptive attacks where $\mathcal{L}$ is the leakage function defined as before, assuming that the* DDH *assumption holds in $\mathbb{G}$, that F and F_p are secure* PRF*s, that* HVE *is a selectively simulation-secure scheme, that* Sym *is an* IND-CPA *secure symmetric encryption scheme, and that Γ is a $\mathcal{L}_{\mathbf{T}}$-secure and computationally correct T-set instantiation.*

Proof. The main idea of proving this theorem, as shown in [19], is similar to that of Theorem 9, except that we need to invoke the adaptive TSet simulator and respond queries adaptively. Roughly speaking, to handle the adaptivity, the simulator with input N chooses N random group elements and then adds them to BF. When simulating the response to each query, the simulator adaptively "assigns" elements of the BF to id-keyword pairs. This is in contrast to the non-adaptive simulator, where it first initialises the A array and then adds the elements to the BF, as determined by the leakage. Currently, the simulator first chooses the elements of the BF, and then uses them or independent elements to initialise A adaptively. □

5.6 Performance Comparison

We first give a list of notations needed in this section for our comparison analysis in Table 5.3.

5.6.1 Comparison between HVE Schemes

The performance of the proposed HXT protocol depends on the parameters of the underlying employed HVE scheme. Hence, we first give the comparison between the available HVE schemes and our SHVE scheme in terms of their performance parameters. After showing the performance advantage of proposed SHVE, we carry on to derive and analyse the performance of our HXT and compare it to that of OXT.

Table 5.3: Notations for comparison analysis.

Notation	Meaning
m, m_p	number of multiplications over $\mathbb{G}$ and $\mathbb{Z}_p$
p, e	number of pairings and exponentiations
$\mathsf{e}_{\mathsf{pre}}$	number of preprocessed exponentiations
G, G_T, Z_p	size of an element from $\mathbb{G}$, $\mathbb{G}_T$, and $\mathbb{Z}_p$ resp.
m'	number of non-wildcard elements in a BF
T_{PRF}	time taken to compute a PRF
T_{hash}	time taken to compute a hash of BF
T_{XOR}	time taken to perform an exclusive-or operation over λ
T_{Enc}	time taken to compute a sym. ciphertext
T_{Dec}	time taken to decrypt a sym. ciphertext
T_{TSet}	time taken to set-up TSet

In Table 5.4, we summarise 4 well-known paring-based HVE constructions, as well as the SHVE scheme we proposed and compare them based on their properties including ciphertext, key (token) sizes as well as encryption, query and token generation computational costs when we use them to encrypt a Bloom filter with the length of m, m is also referred to as the width of the HVE here. Note that the second and third schemes were induced from Inner Product Encryption, while the first and the fourth constructions were originally obtained for HVE model. All the presented schemes except ours are pairing-based constructions (over groups $\mathbb{G}$ and $\mathbb{G}_T$ as domain and range of a bilinear function) with different group orders ranging from just 1 prime to product of 3 primes, while our construction is based on symmetric key encryption.

It is clear that the HVE scheme with pairings in [1] can provide the most efficient construction with low complexity encryption, query, and key generation algorithms amongst the pairing-based HVEs. Therefore, the first evaluation aims to compare the runtime efficiency of our SHVE scheme with the IP [1]. To evaluate our scheme, we implement our SHVE with Java, and we choose to use AES-CMAC as our PRF function while AES in CBC mode as the symmetric key encryption scheme. All above symmetric cryptographic primitives are from the Legion of Bouncy Castle [111]. We adopt 128-bit key length for symmetric key encryption scheme because symmetric encryption with 128-bit key offers better security than the Elliptic Curve Cryptography over a curve with a 160-bit prime order group with a smaller key size according to RFC 4492 [160], and, it can perform encryption/decryption efficiently. For comparison, we use the open-source implementation of IP [1] included in Java Pairing based Cryptography [112] library, which is also implemented by Java, and it is constructed on the curve $y^2 = x^3 + x$ over the field $\mathbb{F}_p$ for some prime $p = 3 \mod 4$, the group operations are based on and the 160-bit prime order groups which are generated from above curve. To make the performance of IP [1] consistent with the theoretical analysis from Table 5.4, we add a preprocessing code for Enc., as it has been implemented for KeyGen. and Query, but missed in Enc..

Table 5.4: Different HVE schemes and their properties.

Ref.	$\lvert\mathbb{G}\rvert$	Ciphertext Size	Key Size	Enc. Cost	Query Cost	KeyGen. Cost
BW [37]	p_1p_2	$(2m+1)\mathsf{G}+(1)\mathsf{G_T}$	$(2m+1)\mathsf{G}$	$(6m+2)\mathsf{m}+(8m+2)\mathsf{e}$	$(2m+1)\mathsf{p}$	$(2m+1)\mathsf{p}+(2)\mathsf{m}$
KSW [154]	$p_1p_2p_3$	$2(2m+1)\mathsf{G}+(1)\mathsf{G_T}$	$(2m+1)\mathsf{G}$	$(4m)\mathsf{m}+(2m)\mathsf{m}_p$ $+(6m+1)\mathsf{e}$	$(2m+1)\mathsf{p}$	$(3m+1)\mathsf{m}+(2m)\mathsf{m}_p$ $+(6m+2)\mathsf{e}$
OT [155]	p_1	$2(5m+1)\mathsf{G}+(1)\mathsf{G_T}$	$(11)\mathsf{G}+(m)\mathsf{Z}_p$	$(m+1)\mathsf{e}+(m+1)\mathsf{m}$	$(11)\mathsf{p}+5(m-1)\mathsf{e}$	$(12m+10)\mathsf{m}$
IP [1]	p_1	$(2m+1)\mathsf{G}+(1)\mathsf{G_T}$	$(2m)\mathsf{G}$	$(2m+2)\mathsf{e}+(1)\mathsf{m}$	$(2m)\mathsf{p}+(2m+1)\mathsf{m}$	$(2m)\mathsf{e}+(2m)\mathsf{m}$
SHVE	N/A	$(m)\lambda$	$\mathcal{O}(m')+2\lambda$	$(m)T_{\mathsf{PRF}}$	$(m')T_{\mathrm{XOR}}+T_{\mathsf{Dec}}$	$(m')T_{\mathsf{PRF}}+(m')T_{\mathrm{XOR}}$ $+T_{\mathsf{Enc}}$

Table 5.5: Communication overhead between client and server and their computational costs.

Conjunctive query $\mathbf{q}=(w_1\wedge w_2\wedge\cdots\wedge w_n)$.			
		OXT [19] cost	HXT cost
Computation	**set-up comp. cost**	$NT_{\mathsf{TSet}}+N\mathsf{e}_{\mathsf{pre}}+Nk\mathsf{T}_{\mathsf{hash}}$	$NT_{\mathsf{TSet}}+N\mathsf{e}_{\mathsf{pre}}+Nk\mathsf{T}_{\mathsf{hash}}+(m)T_{\mathsf{PRF}}$
	search common cost (server) xtag comp.&BF match	$\lvert\mathsf{DB}(w_1)\rvert\,((n-1)(k\mathsf{T}_{\mathsf{hash}}+\mathsf{e}))$	
	search additional cost (server) HVE Queries	N/A	$\lvert\mathsf{DB}(w_1)\rvert((m')T_{\mathrm{XOR}}+T_{\mathsf{Dec}})$
	search common cost (client) stag, xtoken comp. & index recover	$\lvert\mathsf{DB}(w_1)\rvert(n\mathsf{T}_{\mathsf{PRF}}+(n-1)\mathsf{e}_{\mathsf{pre}})+\mathsf{T}_{\mathsf{PRF}}+\mathsf{T}_{\mathsf{Dec}}$	
	search additional cost (client) HVE KeyGen	N/A	$\lvert\mathsf{DB}(w_1)\rvert((m')T_{\mathsf{PRF}}+(m')T_{\mathrm{XOR}}+T_{\mathsf{Enc}})$
Storage	**storage size (server)**	$N\lambda+m$	$N\lambda+(m)\lambda$
Communication	**common comm. bandwidth**	$\lvert\mathbf{t}\rvert+\lvert\mathsf{DB}(w_1)\rvert(n-1)\mathsf{G}+\lvert\mathcal{E}\rvert\mathcal{O}(\lambda)$	
	additional comm. bandwidth token_c transmission	N/A	$\lvert\mathsf{DB}(w_1)\rvert(\mathcal{O}(m')+2\lambda)$

Table 5.6: Execution time comparison between IP [1] and the proposed SHVE; The width of HVE: $m = 10000$, no wildcard element.

Scheme	KeyGen. Time (s)	Enc. Time (s)	Query Time (s)
IP [1]	51.154	50.901	119.219
SHVE	0.172	0.162	0.004

It is also critical to distinguish between m (the length of the entire Bloom filter which is at least as large as $36 \times N$) and m'. In the HVE schemes based on inner product encryption, the complexity of key generation and query depends on m, while in the SHVE, the complexity depends only on the number of non-wildcard characters m' in the vector. This has a significant impact on the practicality of the scheme since a dependence of m would mean a query complexity that scales with the size of the database. However, our query complexity (e.g., number of pairings and exponentiations) grows with the size of the result set rather than the database.

It is also critical to distinguish between m (the length of the entire Bloom filter which is at least as large as $36 \times N$) and m'. In the HVE schemes based on inner product encryption, the complexity of key generation and query depends on m, while in the SHVE, the complexity depends only on the number of non-wildcard characters m' in the vector. This has a significant impact on the practicality of the scheme since a dependence of m would mean a query complexity that scales with the size of the database. However, our query complexity (e.g., number of pairings and exponentiations) grows with the size of the result set rather than the database.

All evaluations are performed on a server with Intel Xeon E5 2660 2.20 GHz CPU and 128 GB of DDR3 RAM. The width of HVE m is set to 10000, and the predicate vector $\mathbf{v}$ we used does not have any wildcard element, which means $m = m'$ in our evaluations. We run IP [1] and proposed SHVE scheme three times respectively to obtain the average execution times of key generation, encryption, and query. The results are shown in Table 5.6. Compared with IP [1], the proposed SHVE scheme is 314x faster in Enc., 297x faster in KeyGen., and almost 30000x faster in Query.

We also evaluate the execution time of SHVE with large m to show the efficiency of SHVE. In this evaluate, the width of HVE is varied from 10^5 to 10^8. The result is reported in Table 5.7, and it is consistent with our theoretical performance analysis.

We can see that the SHVE scheme is efficient even if it is running with a large m. For example, it is able to run KeyGen. and Enc. algorithms within 10 minutes and query algorithm within 3 s when $m = 10^8$ (10^8 elements for processing). Note that it needs several hours to run a pairing-based HVE with the same parameter (m) to encrypt and to generate the key, and several days to query.

Table 5.7: The execution time of SHVE with different sizes of predicate vector. The width of HVE: m is from 10^5 to 10^8, no wildcard element.

SHVE width (m)	KeyGen. Time (s)	Enc. Time (s)	Query Time (s)
10^5	0.715	0.676	0.02
10^6	6.402	6.125	0.052
10^7	58.292	56.417	0.403
10^8	581.933	560.636	3.683

5.6.2 Comparison between OXT and HXT

We compare our scheme with that of [19] in terms of computational complexity (of the set-up and search phases), storage size (of the server), and the number of interaction runs and bandwidth for conjunctive query $(w_1 \wedge w_2 \wedge \cdots \wedge w_n)$ with $\mathbf{sterm}\ w_1$. Note that e, m, and p are defined in Table 5.3. The overall comparison is summarized in Table 5.5.

Set-up computational costs

First, we focus on computational cost spent during the set-up phase. Although both our protocol and the OXT share a lot of similarities, the time taken to generate the encrypted database in HXT is mainly contributed from the computation of the $\mathbf{c}$ in addition to TSet, XSet and BF vector computational costs. The computing of $\mathbf{c}$ adds the computational cost of an HVE.Enc that is $(m)T_{\mathsf{PRF}}$ if we employ the SHVE. If we let $T_{\mathsf{TSet}} = T_{\mathsf{PRF}} + T_{\mathsf{Enc}}$, then computing TSet and XSet in OXT sums up to $NT_{\mathsf{TSet}} + N\mathsf{e}_{\mathsf{pre}}$, where the first term is obtained since TSet has N components each filled up using a PRF and an encryption of a symmetric encryption scheme. For the XSet computation, we make N preprocessed exponentiations, in a total of $N\mathsf{e}_{\mathsf{pre}}$. These are shown in the first row computation comparison of Table 5.5. Note that although we do not entirely generate XSet in HXT, we still compute its components to initiate a BF vector. Bloom filter is employed for keeping the XSet in a reasonable storage space for the practical implementation of OXT [19] and generating HVE for HXT, the BF generation has only the evaluations of hashs H_j, for $1 \leq j \leq k$ and kN elements.

Storage size

We now investigate the storage size of HXT and compare it to OXT. The latter stores TSet and the Bloom filter of XSet in EDB, while HXT uses TSet and an encryption of an HVE system. Note that the latter is in fact an HVE ciphertext replacing the Bloom filter of XSet of OXT. The size of TSet equals N, where each component contains a $\mathbb{G}$ element of size $\mathcal{O}(\lambda)$; and the size of Bloom filter is m. Note that m is approximately $1.44kN$ to attain a negligible probability of false positives, which is 35 times smaller than an XSet with a 512-bit base field for $k = 20$ (1024 bits for each element and $1024N$ in total). Therefore, This in total gives $\mathcal{O}(N(\lambda + k))$ as the storage size in OXT. The size of $\mathbf{c}$ depends on the size of the corresponding BF (used to generate it). Therefore, the ciphertext size is $(m)\lambda$. Hence, the overall storage size of HXT is of order $\mathcal{O}(kN\lambda)$.

Search computational costs

During the search phase, the computational costs are divided between the client and the server. The client in both HXT and OXT has to interact with the server once she wants to send stag and xtokens, where she needs to use a preprocessed element to calculate $(n-1)$ exponentiations for each recovered document in $\mathsf{DB}(w_1)$. The overall computational cost till this stage is $T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n-1)(T_{\mathsf{PRF}} + \mathsf{e}_{\mathsf{pre}}) = T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n)T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n-1)\mathsf{e}_{\mathsf{pre}}$. In OXT, the server then performs $|\mathsf{DB}(w_1)|(n-1)\mathsf{e}$ many xtag generation and $|\mathsf{DB}(w_1)|(n-1)kT_{\mathsf{hash}}$ membership test in the BF vector. [5] It finally returns recovered encrypted indices to the client, when she has to perform a decryption of what was stored in TSet. The latter costs client T_{Dec}. In HXT, the server evaluates vectors $\mathbf{v}_c$, for $1 \leq c \leq |\mathsf{DB}(w_1)|$, and sends it to the client. The client consequently computes token_c using HVE.KeyGen. The first one again is endowed by $(n-1)kT_{\mathsf{hash}}$ and $(n-1)\mathsf{e}$ evaluations, while the second component requires an HVE key generation, if m' denotes the number of non-wildcard components of $\mathbf{v}_c$, then the cost of generating token_c is $(m')T_{\mathsf{PRF}} + (m')T_{\mathrm{XOR}} + T_{\mathsf{Enc}}$. Finally at the server-side, the determination of res through an HVE.Query and a token_c is extra compared to OXT, which costs another $(m')T_{\mathrm{XOR}} + T_{\mathsf{Dec}}$, using the above defined notation. The search computational costs discussed above are presented in the second to fifth row of Table 5.5.

Finally, we determine the computational cost overhead on the server and client-side, respectively. For this purpose, we define O^{srvr} as:

$$\frac{|\mathsf{DB}(w_1)|((m')T_{\mathrm{XOR}} + T_{\mathsf{Dec}})}{|\mathsf{DB}(w_1)|((n-1)(kT_{\mathsf{hash}} + \mathsf{e}))}.$$

Based on a micro-benchmark, the computation time of a hash is insignificant, as the computation cost of an exponentiation is comparable to 1300 evaluations of hashs. In addition, the computation time of an XOR operation is 3 times faster than the hash function, 50 times faster than Sym.Dec and 100 times faster than Sym.Enc. If we apply the typical settings from [19]: let $P_e = 10^{-6}$ and $n = 2$, we will have $k = 20$, and m' is equal to the size of $\mathbf{v}_c$, which is $(n-1)k$ (k in above settings), we conclude that HXT only introduces 1% extra cost on the server-side; We further define O^{client} as

$$\frac{|\mathsf{DB}(w_1)|((m')T_{\mathsf{PRF}} + (m')T_{\mathrm{XOR}} + T_{\mathsf{Enc}})}{|\mathsf{DB}(w_1)|(nT_{\mathsf{PRF}} + (n-1)\mathsf{e}_{\mathsf{pre}}) + T_{\mathsf{PRF}} + T_{\mathsf{Dec}}}.$$

Because the preprocessed element performs exponentiation 17 times slower than PRF and PRF is 43 times slower than exclusive-or operation, the overhead of HXT is about 119% on the client-side. It is easy to deduce from the micro-benchmark that the above two ratios are inversely proportional to n, which means the computational overhead of HXT is smaller when the query has a longer keyword list. However, our evaluations on Section 5.7.3 show

[5] Note that server doesn't preprocess xtokens, because preprocessing takes more time than exponentiation directly, so it only suits in the case when the same element is reused many times.

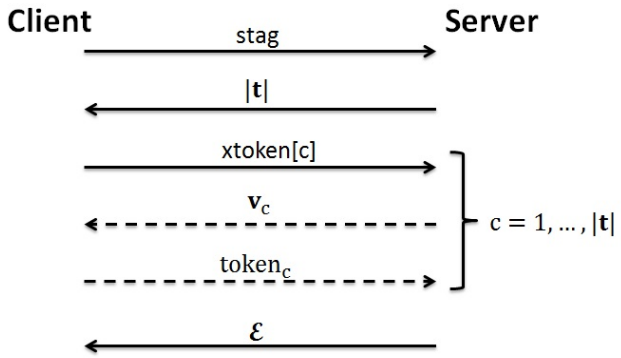


Figure 5.2: All interactions between a server and a client during a search in HXT (all arrows) and OXT (solid arrows only). Since the message flows corresponding to third, forth, and fifth lines are sent in parallel over $c \in [|\mathbf{t}|]$, the HXT protocol only has 6 message flows (or equivalently 3 rounds). This is in contrast to OXT , which has 4 message flows (2 rounds).

that such overhead can be masked by I/O cost on the server-side.

Interaction rounds and bandwidth

In our proposed query protocol SE.Search shown in Algorithm 13 (on page 81), the stag and xtoken generations are the same as that in OXT, which accounts for the first round of interaction between the client and server. The bandwidth of this round is $|\mathbf{t}| + |\mathsf{DB}(w_1)|(n-1)\mathsf{G}$. In the second and third rounds of interactions, the server computes $\mathbf{v}_c$ (using the hashs from the BF) for each encrypted index in vector $\mathbf{t}$ retrieved using stag and TSet and sends it to the client. This interaction costs $|\mathsf{DB}(w_1)|\mathcal{O}(m')$ communication overhead. The client then uses the key generation algorithm of the corresponding HVE to form token_c and lets server to use these token_c to check if the result of this query is "True" or not (using the query algorithm of the underlying HVE scheme). This extra interaction round has $|\mathsf{DB}(w_1)|(\mathcal{O}(m')+2\lambda)$ bandwidth. It gathers all encrypted indices e_c that passes the HVE.Query into a set $\mathcal{E}$. This set will be sent to client with bandwidth $|\mathcal{E}|\mathcal{O}(\lambda)$ as the final result, and the client is further responsible for decrypting the recovered indices using her own secret key to the symmetric encryption Sym. Note that one round of interaction between client and server has been added in our HXT compared to OXT, where the server only checks a set membership against XSet rather than employing HVE. All these are summarized in Fig. 5.2.

5.7 Evaluations

5.7.1 Prototype Implementation

We implement a prototype system for evaluating our HXT protocol. To build this prototype, we firstly implement an OXT prototype, because there is no open-source implementation of OXT. Our implementation uses the cryptographic primitives outlined in Section 5.6.1. Bloom filter is an essential part for both OXT and HXT prototypes; we deploy the Bloom fil-

ter from Alexandr Nikitin as it is the fastest Bloom filter implementation for Java [116]. In our OXT prototype, we set the false positive rate to 10^{-6}, and it enables the OXT prototype to keep the Bloom filter of XSet on the RAM of our server.

The OXT prototype consists of two main parts: one for encrypted database generation and the other for database query. Based on the OXT prototype, we implement the proposed SHVE scheme to enable the prototype to perform HXT protocol. By replacing the Bloom filter of XSet to an SHVE ciphertext in EDB generation, and the Bloom filter query to SHVE key generation and query in the database query, the OXT prototype is adapted to an HXT one. All the above programs are implemented by a combination of Java and Scala code, and it has roughly 2000 lines of code.

The implemented prototypes are designed to fulfil the scalability and the query efficiency characteristics of original protocols. To reach these goals, our prototypes are implemented on the distributed platform (i.e. Hadoop [114]). Hadoop is a distributed platform in master-slave structure: It has a master node that manages the resource and monitors the application execution while a group of slave nodes which dedicate their computational resources (e.g. CPU, RAM, disk) to execute the tasks from the master node. Hadoop [114] offers a distributed file system HDFS [161], in addition to a distributed database HBase [162]. HDFS allows our prototypes to store TSet and SHVE in multiple hard drives in different slave nodes, and reach them concurrently. As a result, it avoids the heavy I/O overhead on a single hard drive; HBase provides an efficient in-memory index mechanism over the distributed dataset, which can highly reduce the TSet and SHVE access time.

For scalability, we take steps to further improve the read performance of TSet and HVE on HBase. HBase is a column-based database [162], that is, the data in the same column are stored in the same file. Therefore, we follow tall narrow model [163] to design TSet table to avoid a very long column value because such long value incurs extra overhead (i.e. compaction) while loading them into memory. In the tall narrow model, each tuple list $\mathbf{T}[w]$ is split into fixed-size blocks with a stag_w and a block counter. Because HBase stores data in key lexicographically order, it stores above blocks with the same stag_w into contiguous disk area. Hence, retrieving TSet only has one random access following by the sequential accesses. HVE is stored as key/value pairs on HBase. Because the HVE is a vector the ciphertext, we use the index of vector as the key and the corresponding ciphertext as value. Due to the variety of HVE key, it is difficult to avoid the random access of HVE ciphertext. Therefore, we use the randomised index of HVE ciphertext as the key of HVE, because random keys help to distribute the data into different nodes, which enables the random access in parallel [163].

To accelerate the query phase, we make use of the distributed in-memory computing framework Spark [80]. Spark follows the same data processing flow as MapReduce [164], which distributes the computing tasks and execute them on different slave nodes in parallel. Spark inherits the scalability and fault tolerance of MapReduce [80], but it can execute tasks

Table 5.8: Statistics of the datasets used in the evaluation.

Size	# of documents	Distinct keywords	Distinct (w,id) pairs
2.93GB	$7.8 * 10^5$	$4.0 * 10^6$	$6.2 * 10^7$
8.92GB	$2.7 * 10^6$	$1.0 * 10^7$	$1.6 * 10^8$
60.2GB	$1.6 * 10^7$	$4.3 * 10^7$	$1.4 * 10^9$

in-memory without keeping any intermediate data on disk, it means our prototypes do not have any I/O operation during database query except the TSet and SHVE query.

We deploy our prototypes on a shared Hadoop cluster with 13 slave nodes and one master node. Each node has 2x Intel Xeon CPU E5-2660 2.2GHz (each CPU has 8 cores with dual-thread) and 128GB RAM, in addition, we have another node with the same specification above which is served as edge node and client of our prototypes. All nodes are connected by InfiniBand [165] network technique. The cluster installs CDH 5.2.6 [166], one of the most complete and popular distribution of Hadoop and its peripheral ecosystem (contains Hadoop Yarn 2.5.0, HBase 0.96.8 and Spark 2.0.2). Based on the setup configuration and scheduling policy, we can use at most 416 virtual cores (32 virtual cores in each slave node) and 1248GB RAM (96GB RAM in each slave node), in addition, each virtual core should have at least 2GB. In the real-world scenario, 1 virtual core and 2GB RAM are needed for running the monitor program of a distributed application on Hadoop. As a result, our prototypes can start 415 tasks with 1 virtual core and 3GB RAM concurrently at most. However, our following evaluations show that it is not necessary to use all resources to query the database: 100 concurrent tasks with 1 virtual core and 2GB RAM are sufficient to provide a satisfactory result.

5.7.2 Datasets

We test our implementation on three datasets from Wikimedia Downloads [167]: the original sizes of our datasets are 2.93GB[6], 8.92GB[7] and 60.2GB[8], respectively. A brief summary of the statistical features of the datasets is given in Table 5.8.

The corresponding EDB and Bloom filter size for the above three datasets are 9.3GB and 215MB, 33GB and 575MB, 256GB and 4.76GB. In addition, the HVE size is 28GB, 76GB and 647GB. Fig 5.3 further gives the frequency of keywords according to the number of documents to depict the keyword occurrence distribution of the generated EDBs.

5.7.3 Evaluation Results

Our evaluation aims to verify the following: (i) our implementation in the distributed platform can ensure the efficiency of queries; (ii) the additional query latency introduced in

[6]enwiki-20161220-pages-articles22.xml
[7]enwiki-20161220-pages-articles27.xml
[8]enwiki-20171201-pages-articles.xml

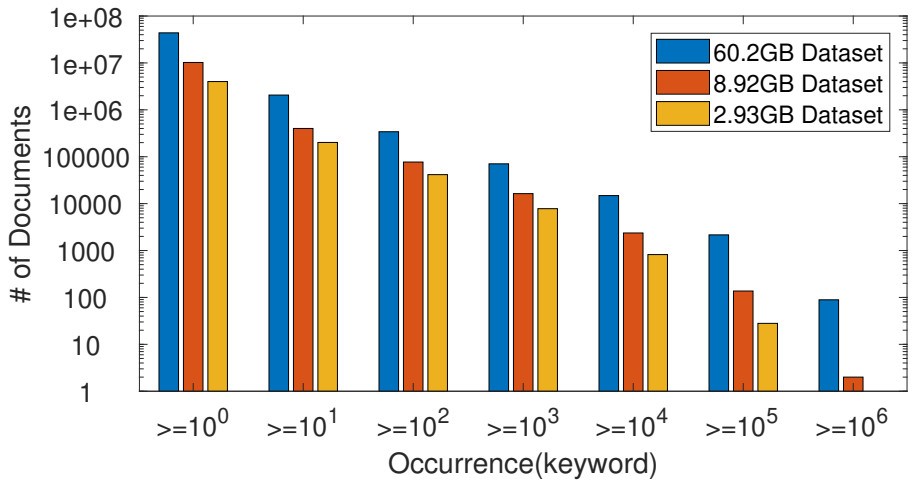


Figure 5.3: The keyword occurrence distribution of three datasets.

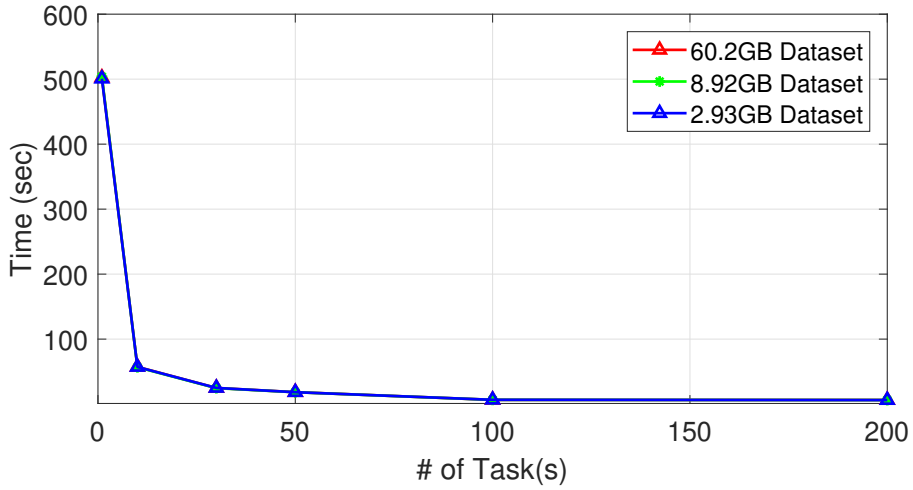


Figure 5.4: HXT server query time when # of parallel tasks increases.

HXT is small; (iii) HXT keeps the scalability property of OXT.

The impact of parallelism

First, we study how distributed computing influences the query efficiency of HXT prototype. We choose a keyword with about 330K matched documents respectively in three datasets, and we use the selected keyword as the **sterm** to perform a two-terms conjunctive query in our HXT prototype. We vary the number of parallel tasks from 1 to 200 before the server start running the query to test the impact of parallelism. As shown in Fig 5.4, we observe that parallelism successfully improved the efficiency of queries by a factor of 100 times on the server-side. In addition, we conclude the impact in three cases: (i) when increasing the number of parallel tasks from 1 to 10, parallelism can highly improve the server-side performance; (ii) when the number of parallel tasks is between 10 to 100, parallelism only can slightly improve the query efficient on server; (iii) after the number of parallel tasks is larger than 100, parallelism does not affect the query efficiency.

The reason is that the computational cost is the dominant cost when the server only has a small fraction of resources is allowed to engage the computation. By increasing the par-

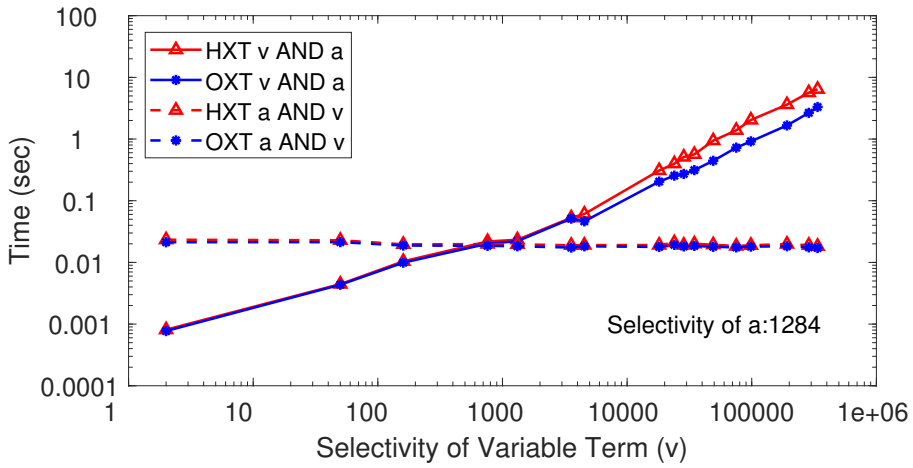


Figure 5.5: Server performance comparison between HXT and OXT in 2.93GB dataset.

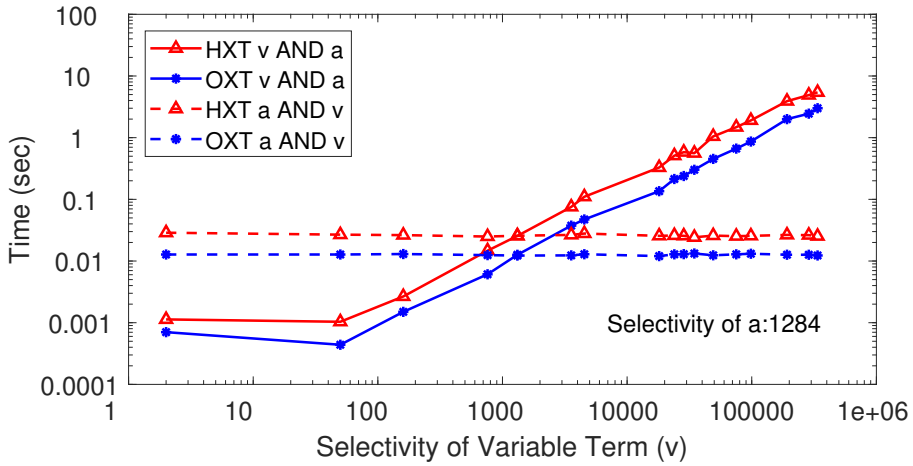


Figure 5.6: Client performance comparison between HXT and OXT in 2.93GB dataset.

allelism factor in the above case, the computation cost of each task can be highly reduced, which yields a significant performance improvement. However, with the increase of the number of parallel tasks, it incurs more communication cost for task scheduling and monitoring between the master and slave nodes of our cluster, and HBase also has I/O limits based on the underlying infrastructure. Therefore, the computation cost is overlapped by the communication cost and I/O latency after we have more than 100 parallel tasks.

Another observation is the query latency highly depends on the selectivity of **sterms**, while it is independent of the size of the dataset. We examine it deeply in the scalability test at the end of this section.

Performance comparison

We use the parallel factor 100 to further investigate the additional overhead in HXT comparing with OXT protocol. Due to the OXT and HXT protocol having the same behaviour when performing the single-keyword search, our evaluation only reports the query performance of the conjunctive query.

We choose a variable term, named **v**, on the 2.93GB dataset. The selectivity of **v** is from 2

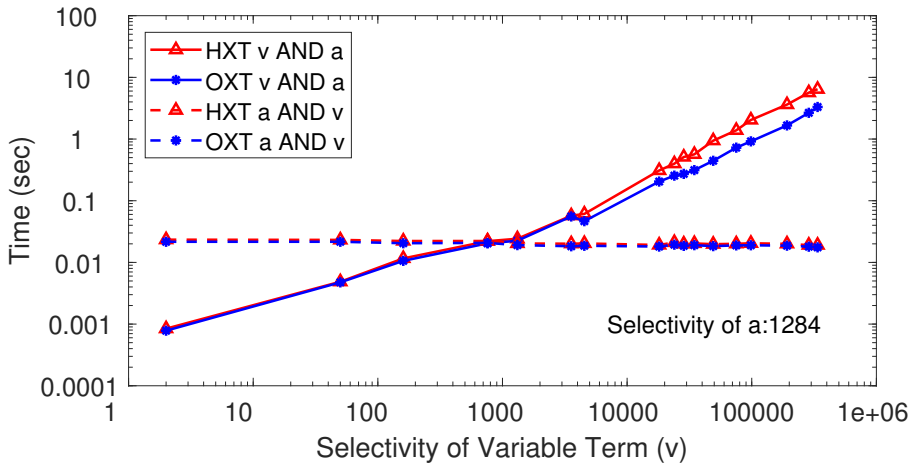


Figure 5.7: Overall query delay comparison between HXT and OXT in 2.93GB dataset.

to 337449 documents. We further choose a fixed term **a** and perform two types of conjunctive queries on the 2.93GB dataset. Fig 5.5 shows the time spent by HXT and OXT during the query on the server-side. The first conjunctive query uses the **v** as the **sterm** and the **a** as the **xterm**. Hence, the OXT server time is linear to the selectivity of **v**, because it needs to do an additional exponentiation for each tuple from the TSet to check against the XSet. When the selectivity of **v** is small, we observe that HXT prototype has $2\% - 8\%$ additional cost comparing with OXT. However, it slows down with the increase of selectivity of **v**. This is because HXT requires to access HBase to get HVE ciphertext, increasing the selectivity also means the server needs to do more HBase access, which increases the load of I/O.

Another conjunctive query uses **a** as the **sterm** and **v** as the **xterm**, the server then runs in a steady constant time regardless of the selectivity of **v**. In the above case, HXT has $2\%-8\%$ overhead against the OXT over time. This also illustrates the importance of choosing the least frequent term as the **sterm**.

The query time on the client-side of HXT and OXT is demonstrated in Fig 5.6. Comparing with the server, the client does not have any I/O operation, so it purely reflects the computation costs, and it fits well with the analysis in Section 5.6.2 as HXT is 2 times slower than OXT.

However, as we design our prototypes to perform queries in parallel, the computation cost of the HVE key generation on the client-side can be overlapped by the xtag generation (it always slower than the xtoken generation because it does not use preprocessed elements), as well as the HBase I/O for loading HVE ciphertext on the server-side. As shown in Fig. 5.7, the overall performance of HXT is not affected by the computation on the client-side.

We further present the query delay comparison of HXT in our three datasets to illustrate the efficiency and scalability of HXT. In this evaluation, we use the execution time of HXT in 2.93 GB dataset as the baseline. Fig. 5.9 shows the efficiency of HXT, as there is only a negligible difference between these execution times of HXT protocol on different sizes of datasets. It also demonstrates the highly scalable property of HXT, because the invariant

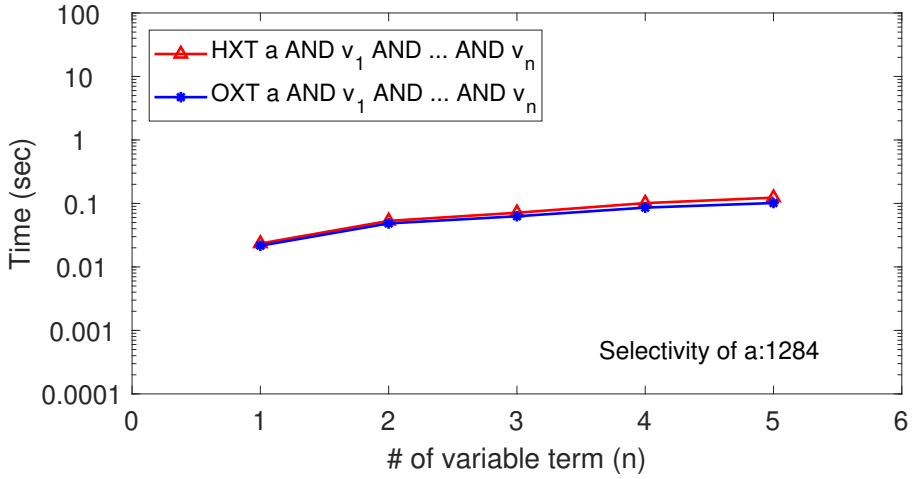


Figure 5.8: Overall query delay comparison between HXT and OXT under multi-keyword setting in 2.93GB dataset.

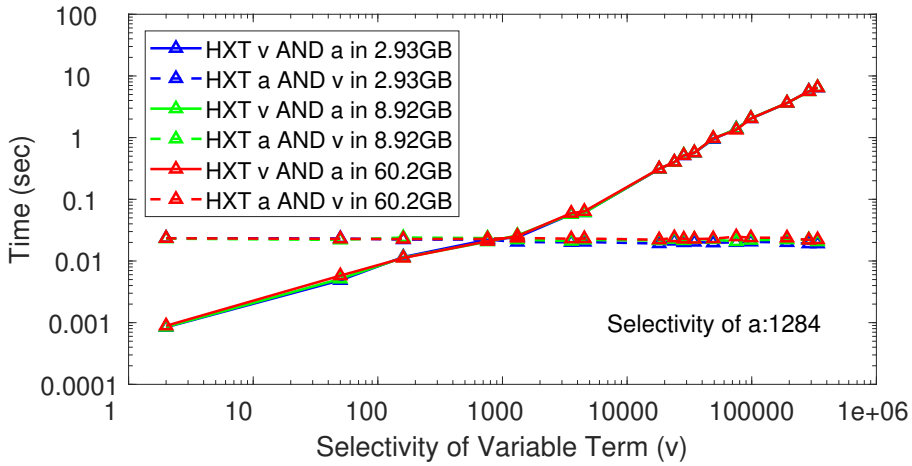


Figure 5.9: Overall query delay comparison of HXT for different sizes of datasets.

query delay implies that the delay is independent to the size of the dataset, even if the encrypted dataset is larger than the size of RAM.

The last evaluation in this part aims to compare the performances of HXT and OXT for querying multiple keywords. In this evaluation, the **sterm** is identical to the fixed term **a** in previous two-keyword evaluation, but we introduce more variable terms $\{\mathbf{v_n}\}, \mathbf{n} \in [1, 5]$ as **xterms** in the conjunctive query. As shown in Fig. 5.8, the query delay increases if the query has more keywords. The underlying reason is that the size of m' is proportional to the size of the keyword list, and the HXT server is required to have more I/O operations with the increasing size of m'. However, such delay is affordable: the HXT prototype has only 8% and 21% additional cost when the query has two and six keywords, respectively.

Communication overhead comparison.

We use the test query with various selectivities in Section 5.7.3 (i.e. variable term **v** AND fixed term **a**) to compare the communication overhead of OXT and HXT. The evaluate is conducted in 2.93GB dataset. However, the communication overhead is identical for different

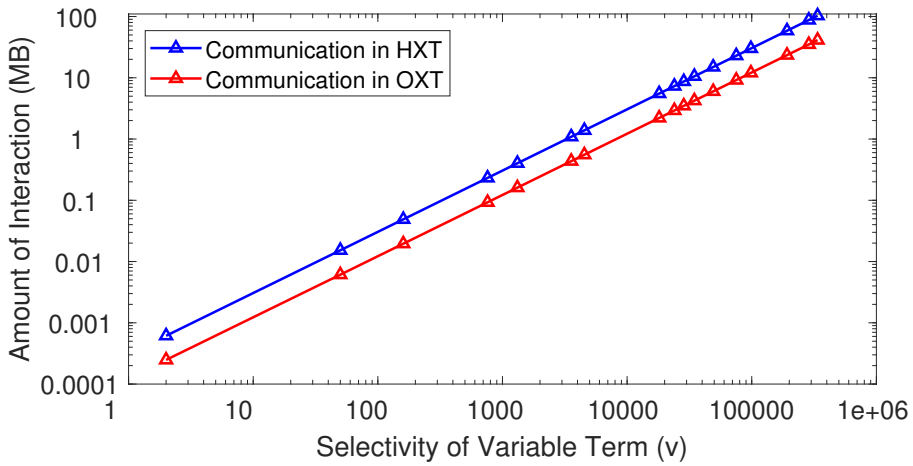


Figure 5.10: Bandwidth communication comparison of HXT and OXT in 2.93GB dataset.

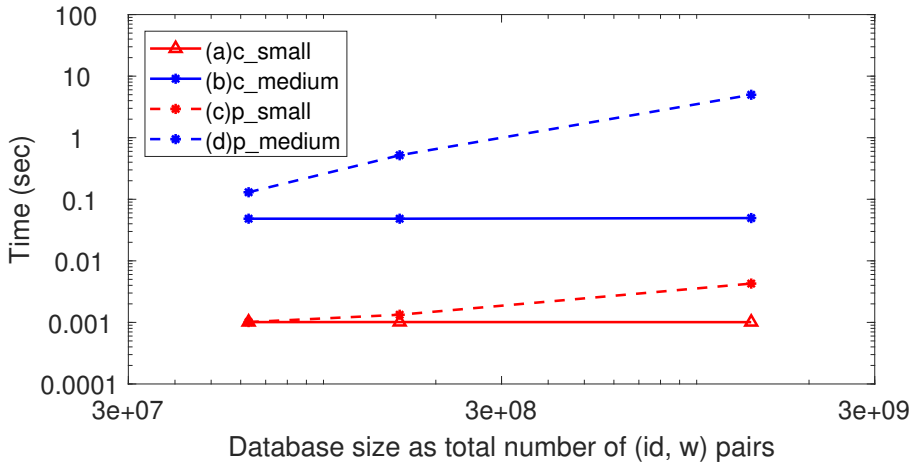


Figure 5.11: HXT scalability test in various dataset, the test is running in four cases: (a) constant small (10) result set; (b) constant medium-size (10000) result set; (c) proportional small result set; (d) proportional medium result set.

datasets, because it only correlates with the selectivity of **v** (see Table 5.5).

The evaluation result (see Fig. 5.10) shows that HXT needs 1.5 times more in communications to transmit the token for HVE, since OXT only requires the client to transmit the xtoken. Such communication overhead is moderate in our test system because the client only sends 100MB tokens at most for a query with about 330K documents, which can be handled within several milliseconds. However, it introduces an extra delay for the queries if the selectivity of **v** is large and the network bandwidth is limited.

Scalability of HXT

We leverage three EDBs we generated from Wikimedia Downloads [167] to demonstrate the scalability of the HXT protocol. We follow the same evaluation method in [19] to inject artificial query terms to randomly selected documents. Fig. 5.11 shows that the implementation of HXT preserves the scalability of OXT protocol even when the database sizes exceed the memory we assigned for database query. The query time of HXT is independent of the size of the database. Instead, it is linear to the size of the result set.

5.8 Conclusion

In this chapter, we propose a new searchable encryption protocol to obtain better security than to the current existing OXT protocol at the cost of slightly increasing the communication and computation overhead.

In particular, we have introduced the hidden cross-tags (HXT) protocol by employing Bloom filters (BF) and newly introduced lightweight symmetric hidden vector encryption (SHVE). It is similar to OXT [19] except that we replace the XSet by an SHVE encryption of BF. The search algorithm re-constructs the search tokens and performs SHVE query algorithm instead of an XSet membership test. It has been shown that our SHVE is selectively simulation-secure, and our HXT is computationally correct, semantically secure against selective adversaries. Implementation and experiments have been conducted to compare the efficiency of the SHVE with those pairing-based HVEs and HXT versus OXT accordingly.

Some possible further research directions are: (*i*) to establish a protocol achieving a better security (by even removing WRP from the leakage profile) robust to the recent attacks [36], while supporting Boolean queries, (*ii*) to apply HXT to other types of queries including rich queries [151, 168], (*iii*) to employ HXT in dynamic SSE scheme with forward/backward security [107].

Chapter 6

Future Directions

This chapter presents some possible research directions and our on-going projects related to the social search system and the underlying cryptographic primitives.

- **Verifiable social search system.** Although this thesis focuses on the data confidentiality in social search systems, it is also interesting to build a system which allows the user to verify the correctness of query results. Such a system can effectively address the attack from malicious adversaries and offer a strong guarantee to the correctness of social search functionality.
- **Machine learning over the encrypted social graph.** Machine learning algorithms are an important component in OSN, and they also perform analysis based on users' sensitive data. Thus, designing privacy-preserving machine learning algorithms with a proper result disclosure control is a potential future direction.
- **Dynamic searchable encryption schemes.** The trend of searchable encryption research is towards the design of dynamic searchable encryption schemes, and new security model (forward and backward security [107]) is proposed to address the threat under the dynamic setting. There are some interesting topics under the above new context:
  - **Hardware-assisted dynamic SSE.** The new security requirement incurs the degradation of performance. To preserve the efficiency of SSE schemes, a promising research direction is to investigate how to use trusted hardware (i.e., Intel SGX [65]) to assist the protocol processing. More specifically, the trusted hardware offers an isolated environment named *enclave* to store and execute the code and data securely. This can serve as an efficient alternative for the cryptographic primitives that incur heavy computation and communication costs.
    Recently, some dynamic SSE schemes based on SGX [169, 170] have been proposed, whereas they cannot handle large dataset. The reason is that those schemes utilise fail to consider the limitation of SGX. Particularly, SGX has a limited memory space (128 MB) to support the application. If an application requires more than 128 MB memory, the extra memory is stored outside, and the CPU will load it on demand via paging, which is a very costly procedure. Meanwhile, there is a considerable communication cost between the enclave and outside (server). In [171], we devise a new SGX-assisted dynamic SSE protocol to remedy the above issue: Firstly, we integrate the batch processing to reduce the I/O cost between the server and enclave. Additionally, we employ Bloom filter to compress the database state stored within the enclave. Our results show that the proposed scheme is $2\times$ faster than the schemes in [170] since it avoids paging. Also, the

number of I/O operations is decreased by $30\times$ at most, and the insertion and deletion operation is $2\times$ faster due to the reduced I/O cost.

  - **Dynamic SSE with rich functionalities.** The existing dynamic schemes are only able to answer single keyword queries. Another possible direction is to build a practical dynamic SSE construction to support rich functionalities (e.g., boolean queries, range queries). This can improve the practicality of dynamic schemes since the new scheme supports more real-world use cases.

- **Applications based on SHVE.** As shown in Chapter 5, the SHVE scheme is a computation and communication-efficient primitive that enables secure membership testing. Since membership testing is an important building block in real-world applications, a potential research direction is to build secure applications based on SHVE.
  In our recent work [172], we show that SHVE can be an effective solution for secure pattern matching in network traffic. We observe that the existing privacy-preserving pattern matching protocols rely on some heavy building blocks such as sliding window tokenisation [173] and public-key cryptographic primitives [174]. Thus, they involve a noticeable computation and communication overhead and are far from practical. Compared to the previous protocols, SHVE does not require to tokenise the string via the sliding window, because SHVE examines the pattern in byte-wise instead of checking it as a whole. Also, the HVE scheme can be very efficient since it only relies on symmetric-key primitives. The preliminary results show that the SHVE-based pattern matching protocol saves 94% in bandwidth consumption comparing to the previous work and still achieves a microsecond-level matching delay.

Chapter 7

Conclusion

This thesis shows how to build a practical system to protect data confidentiality on OSN while preserving the functionality of social search. To achieve this goal, this thesis starts by identifying the essential operations (aka atomic operations) for social search. It provides customised cryptographic tools that offer strong security guarantees and efficiently process queries on structured social graph data, as well as queries with computation. In addition, this thesis introduces an encrypted graph model with parallel processing support, which allows the proposed system to scale up for large OSN in the real world. This thesis designs and realises GraphSE^2 for privacy-preserving social search query processing, as well as a privacy-preserving outlier detection application to show the usability of GraphSE^2. For an average user (130 friends), GraphSE^2 can efficiently process his/her queries within 1 s. Furthermore, outlier detection is implemented as an application upon GraphSE^2, the performance overheads for the application is modest (around 200 ms to answer query). GraphSE^2 is a highly-scalable system that can support billions of users in the real-world deployment. It allows OSN providers to store the encrypted data in the public cloud while providing a satisfied OSN services to their users. As a result, OSN providers could embrace the advantage of using a public cloud as the back-end since it reduces the maintenance cost. Also, the proposed system mitigates the risk of data breach and helps OSN providers to avoid the financial loss and legal penalty in consequence.

This thesis also presents a cryptographic scheme HXT, which is dedicated to improving the security of conjunctive queries. HXT successfully eliminate the so call 'KPRP' leakage, which can be exploited to recover all the encrypted IDs with 100% accuracy after several conjunctive queries on the encrypted database. Furthermore, the evaluations indicate the practicality of the proposed system and cryptographic primitives: HXT improves the security level of conjunctive queries with only 10% extra query delay compared to OXT. HXT is a practical alternative to OXT since it blocks the attacks on conjunctive queries while retaining the functionality and efficiency of OXT.

References

[1] V. Iovino and G. Persiano. Hidden-Vector Encryption with Groups of Prime Order. In *Proceedings of the 2nd International Conference on Pairing-Based Cryptography*, pages 75–88, 2008.

[2] Statista. Number of Monthly Active Facebook Users Worldwide as of 3rd Quarter 2017. `https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/` [online], 2017.

[3] Internet World Stats. World Internet Usage and Population Statistics. `http://www.internetworldstats.com/stats.htm` [online], 2017.

[4] R. Gross and A. Acquisti. Information Revelation and Privacy in Online Social Networks. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 71–80, 2005.

[5] UpGuard. Block Buster: How A Private Intelligence Platform Leaked 48 Million Personal Data Records. `https://www.upguard.com/breaches/s3-localblox` [online], 2018.

[6] C. Cadwalladr and E. Graham-Harrison. Facebook-Cambridge Analytica Data Scandal. `https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election` [online], 2018.

[7] Amazon Web Services. AWS Case Study: Airbnb. `https://aws.amazon.com/solutions/case-studies/airbnb` [online], 2018.

[8] Amazon Web Services. AWS Case Study: PIXNET. `https://aws.amazon.com/solutions/case-studies/pixnet` [online], 2018.

[9] Instagram Engineering. What Powers Instagram: Hundreds of Instances, Dozens of Technologies. `https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad` [online], 2018.

[10] K. Ren, C. Wang, and Q. Wang. Security Challenges for the Public Cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.

[11] G. Brown, T. Howe, M. Ihbe, A. Prakash, and K. Borders. Social Networks and Context-Aware Spam. In *Proceedings of the 2008 ACM Conference on Computer-Supported Cooperative Work*, pages 403–412, 2008.

[12] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[13] Facebook. Facebook Graph Search. `https://www.facebook.com/graphsearcher/` [online], 2013.

[14] LinkedIn Engineering. IndexTank Social Search Engine and Services. `https://engineering.linkedin.com/open-source/indextank-now-open-source` [online], 2011.

[15] R.A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[16] V. Pappas et al. Blind seer: A Scalable Private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374, 2014.

[17] A. Papadimitriou et al. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Conference on Operating Systems Design and Implementation*, pages 587–602, 2016.

[18] X. Yuan, Y. Guo, X. Wang, C. Wang, B. Li, and X. Jia. EncKV: An Encrypted Key-Value

Store with Rich Queries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 423–435, 2017.

[19] D. Cash, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013*, pages 353–373, 2013.

[20] S. Kamara and T. Moataz. Boolean Searchable Symmetric Encryption with Worst-Case Sub-Linear Complexity. In *Advances in Cryptology – EUROCRYPT 2017*, pages 94–124, 2017.

[21] X. Meng, S. Kamara, K. Nissim, and G. Kollios. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 504–517, 2015.

[22] D.J. Wu, J. Zimmerman, J. Planul, and J.C. Mitchell. Privacy-Preserving Shortest Path Computation. ArXiv e-prints, arXiv:1601.02281, 2016.

[23] M. Chase and S. Kamara. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology – ASIACRYPT 2010*, pages 577–594, 2010.

[24] K. Nayak, X. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394, 2015.

[25] E. Boyle, K-M. Chung, and R. Pass. Oblivious Parallel RAM and Applications. In *TCC'16*, 2016.

[26] C. Gentry, A. Sahai, and B. Water. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, pages 75–92, 2013.

[27] E. Stefanov et al. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–310, 2013.

[28] S. Lai, X. Yuan, S-F. Sun, J.K. Liu, Y. Liu, and D. Liu. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 41–54, 2019.

[29] M. Curtiss et al. Unicorn: A System for Searching the Social Graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[30] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[31] D. Demmler, T. Schneider, and M. Zohner. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*, 2015.

[32] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, 2017.

[33] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 29–42, 2007.

[34] L.H. Ungar and D.P. Foster. Clustering Methods for Collaborative Filtering. In *AAAI Workshop on recommendation Systems*, volume 1, pages 114–129, 1998.

[35] Z. Miller, B. Dickinson, W. Deitrick, W. Hu, and A.H. Wang. Twitter Spammer Detection using Data Stream Clustering. *Information Sciences*, 260:64–73, 2014.

[36] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium*, pages 707–720, 2016.

[37] D. Boneh and B. Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *Theory of Cryptography*, pages 535–554, 2007.

[38] S. Buchegger, D. Schiöberg, L-H. Vu, and A. Datta. PeerSoN: P2P Social Networking: Early Experiences and Insights. In *Proceedings of the 2nd ACM EuroSys Workshop on Social Network Systems*, pages 46–52, 2009.

[39] L.A. Cutillo, R. Molva, and T. Strufe. Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[40] S. Nilizadeh, S. Jahid, P. Mittal, N. Borisov, and A. Kapadia. Cachet: A Decentralized Architecture for Privacy Preserving Social Networking with Caching. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 337–348, 2012.

[41] Diaspora Foundation. The Diaspora* Project. `https://diasporafoundation.org` [online], 2010.

[42] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L.P. Cox. Confidant: Protecting OSN Data without Locking It Up. In *Proceedings of the 12th International Middleware Conference*, pages 60–79, 2011.

[43] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. In *Proceedings of the 1st Workshop on Online Social Networks*, pages 49–54, 2008.

[44] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better Privacy for Social Networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 169–180, 2009.

[45] J. Domingo-Ferrer, A. Viejo, F. Sebé, and Ú. González-Nicolás. Privacy homomorphisms for social networks with private relationships. *Computer Networks*, 52(15):3007–3016, 2008.

[46] E. De Cristofaro, C. Soriente, G. Tsudik, and A.'Williams. Hummingbird: Privacy at The Time of Twitter. In *2012 IEEE Symposium on Security and Privacy*, pages 285–299, 2012.

[47] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-Diversity: Privacy Beyond k-Anonymity. In *IEEE 22nd International Conference on Data Engineering*, pages 24–35, 2006.

[48] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An Online Social Network with User-Defined Privacy. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, pages 135–146, 2009.

[49] J. Sun, X. Zhu, and Y. Fang. A Privacy-Preserving Scheme for Online Social Networks with Efficient Revocation. In *Proceedings of the 29th Conference on Information Communications*, pages 2516–2524, 2010.

[50] S. Jahid, P. Mittal, and N. Borisov. EASiER: Encryption-based Access Control in Social Networks with Efficient Revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 411–415, 2011.

[51] D.X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[52] E-J. Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[53] A. Boldyreva and N. Chenette. Efficient Fuzzy Search on Encrypted Data. In *International Workshop on Fast Software Encryption*, pages 613–633, 2014.

[54] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 253–262, 2010.

[55] F. Kerschbaum and A. Tueno. An Efficiently Searchable Encrypted Data Structure for Range Queries. In *European Symposium on Research in Computer Security*, pages 344–364, 2019.

[56] C. Zuo, S-F. Sun, J.K. Liu, J. Shao, and J. Pieprzyk. Dynamic Searchable Symmetric En-

cryption Schemes Supporting Range Queries with Forward (and Backward) Security. In *European Symposium on Research in Computer Security*, pages 228–246, 2018.

[57] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, 2016.

[58] P. Mukherjee and D. Wichs. Two Round Multiparty Computation via Multi-Key FHE. In *Advances in Cryptology — EUROCRYPT 2016*, pages 735–763, 2016.

[59] Y. Lindell, B. Pinkas, N.P. Smart, and A. Yanai. Efficient Constant-Round Multi-party Computation Combining BMR and SPDZ. *Journal of Cryptology*, 32(3):1026–1069, 2019.

[60] MySQL. MySQL Enterprise Encryption. `https://www.mysql.com/products/enterprise/encryption.html` [online], 2019.

[61] Oracle. SQL Server Encryption. `https://docs.oracle.com/cd/B19306_01/network.102/b14268/asotrans.htm#ASOAG600` [online], 2019.

[62] Mircosoft. Transparent Data Encryption. `https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/sql-server-encryption?view=sql-server-ver15` [online], 2019.

[63] R. Poddar, T. Boelter, and R.A. Popa. Arx: A Strongly Encrypted Database System. Cryptology ePrint Archive, Report 2016/591, 2016.

[64] P. Xie and E. Xing. CryptGraph: Privacy Preserving Graph Analytics on Encrypted Graph. ArXiv e-prints, arXiv:1409.5021, 2014.

[65] Intel. Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx` [online], 2019.

[66] ARM. ARM TrustZone Technology. `https://developer.arm.com/ip-products/security-ip/trustzone` [online], 2019.

[67] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in The Cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, 2015.

[68] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A-R. Sadeghi. HardIDX: Practical and Secure Index with SGX. In *Data and Applications Security and Privacy XXXI*, pages 386–408, 2017.

[69] W. Zheng, A. Dave, J.G. Beekman, R.A. Popa, J.E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 283–298, 2017.

[70] K. Liang, J.K. Liu, R. Lu, and D. S. Wong. Privacy Concerns for Photo Sharing in Online Social Networks. *IEEE Internet Computing*, 19(2):58–63, 2015.

[71] Information is Beautiful. World's Biggest Data Breaches. `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks` [online], 2018.

[72] X. Yang, X. Huang, and J. K. Liu. Efficient Handover Authentication with User Anonymity and Untraceability for Mobile Cloud Computing. *Future Generation Computer Systems*, 62:190–195, 2016.

[73] J. K. Liu, K. Liang, W. Susilo, J. Liu, and Y. Xiang. Two-Factor Data Security Protection Mechanism for Cloud Storage System. *IEEE Transactions on Computers*, 65(6):1992–2004, 2016.

[74] J. Li, L. Zhang, J. K. Liu, H. Qian, and Z. Dong. Privacy-Preserving Public Auditing Protocol for Low-Performance End Devices in Cloud. *IEEE Transactions on Information Forensics and Security*, 11(11):2572–2583, 2016.

[75] D. Sullivan. Google's Results Get More Personal With "Search Plus Your World". `https://searchengineland.com/googles-results-get-more-personal-with-search-plus-your-world-107285` [online], 2012.

[76] M. Blanton, A. Steele, and M. Alisagari. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 207–218, 2013.

[77] S. Kamara, T. Moataz, and O. Ohrimenko. Structured Encryption and Leakage Suppression. In *Advances in Cryptology – CRYPTO 2018*, pages 339–370, 2018.

[78] D. Xie et al. Practical Private Shortest Path Computation Based on Oblivious Storage. In *2016 IEEE 32nd International Conference on Data Engineering*, pages 361–372, 2016.

[79] Q. Wang, K. Ren, M. Du, Q. Li, and A. Mohaisen. SecGDB: Graph Encryption for Exact Shortest Distance Queries with Efficient Updates. In *Financial Cryptography and Data Security*, pages 79–97, 2017.

[80] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95–101, 2010.

[81] Redis Labs. Redis. `https://redis.io` [online], 2017.

[82] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper*, 5(8), 2007.

[83] N. Cao, Z. Yang, C. Wang, K. Ren, and W. Lou. Privacy-Preserving Query over Encrypted Graph-Structured Data in Cloud Computing. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, pages 393–402, 2011.

[84] Z. Chang, L. Zou, and F. Li. Privacy Preserving Subgraph Matching on Large Graphs in Cloud. In *Proceedings of the 2016 International Conference on Management of Data*, pages 199–213, 2016.

[85] S. Sharma, J. Powers, and K. Chen. PrivateGraph: Privacy-Preserving Spectral Analysis of Encrypted Graphs in the Cloud. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):981–995, 2018.

[86] M. Naveed, S. Kamara, and C.V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655, 2015.

[87] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8), 2012.

[88] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang. Nxgraph: An Efficient Graph Processing System on A Single Machine. In *2016 IEEE 32nd International Conference on Data Engineering*, pages 409–420, 2016.

[89] M.T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient Authenticated Data Structures for Graph Connectivity and Geometric Search Problems. *Algorithmica*, 60(3):505–552, 2011.

[90] D. Beaver. Efficient Multiparty Protocols using Circuit Randomization. In *Advances in Cryptology – CRYPTO '91*, pages 420–432, 1991.

[91] A.C. Yao. Protocols for Secure Computations. In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[92] M. Bellare, V.T. Hoang, and P. Rogaway. Foundations of Garbled Circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 784–796, 2012.

[93] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 535–548, 2013.

[94] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-party Computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[95] Amazon Web Services. AWS Outposts: Run AWS Infrastructure On-Premises for A Truly Consistent Hybrid Experience. `https://aws.amazon.com/outposts` [on-

line], 2018.

[96] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing Multi-Party Computation. Cryptology ePrint Archive, Report 2011/272, 2011.

[97] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-Preserving Matrix Factorization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 801–812, 2013.

[98] F. Baldimtsi and O. Ohrimenko. Sorting and Searching Behind the Curtain. In *Financial Cryptography and Data Security*, pages 127–146, 2015.

[99] J.S. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 43–52, 1998.

[100] S. Lai et al. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 745–762, 2018.

[101] P. Pullonen, D. Bogdanov, and T. Schneider. The Design and Implementation of A Two-Party Protocol Suite for Sharemind 3. `http://tubiblio.ulb.tu-darmstadt.de/61259` [online], 2012.

[102] K.E. Batcher. Sorting Networks and their Applications. In *Proceedings of the Spring Joint Computer Conference*, pages 307–314, 1968.

[103] M. McPherson, L. Smith-Lovin, and J.M. Cook. Birds of a Feather: Homophily in Social Networks. *Annual Review of Sociology*, 27(1):415–444, 2001.

[104] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[105] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 668–679, 2015.

[106] R. Bost and Fouque P-A. Thwarting Leakage Abuse Attacks against Searchable Encryption–A Formal Approach and Applications to Database Padding. Cryptology ePrint Archive, Report 2011/1060, 2017.

[107] R. Bost, B. Minaud, and O. Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482, 2017.

[108] S-F. Sun et al. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 763–780, 2018.

[109] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[110] E.M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data Recovery on Encrypted Databases with K-Nearest Neighbor Query Leakage. In *2019 IEEE Symposium on Security and Privacy*, pages 245–262, 2019.

[111] The Legion of the Bouncy Castle. Bouncy Castle Crypto APIs. `https://www.bouncycastle.org` [online], 2007.

[112] A. De Caro and V. Iovino. JPBC: Java Pairing Based Cryptography. In *2011 IEEE Symposium on Computers and Communications*, pages 850–855, 2011.

[113] X. Wang. FlexSC. `https://github.com/wangxiao1254/FlexSC` [online], 2018.

[114] Apache. Hadoop. `https://hadoop.apache.org` [online], 2015.

[115] B.H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[116] A. Nikitin. Bloom Filter Scala. `https://alexandrnikitin.github.io/blog/`

bloom-filter-for-scala [online], 2017.
[117] S. Sadik and L. Gruenwald. Research Issues in Outlier Detection for Data Streams. *ACM SIGKDD Explorations Newsletter*, 15(1):33–40, 2014.
[118] X. Yuan, X. Wang, J. Lin, and C. Wang. Privacy-Preserving Deep Packet Inspection in Outsourced Middleboxes. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.
[119] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3), 2009.
[120] K. Fu and W. Xu. Risks of Trusting the Physics of Sensors. *Communications of the ACM*, 61(2):20–23, 2018.
[121] M. Salehi, C. Leckie, J. C. Bezdek, T. Vaithianathan, and X. Zhang. Fast Memory Efficient Local Outlier Detection in Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3246–3260, 2016.
[122] M. Gupta, J. Gao, C.C. Aggarwal, and J. Han. Outlier Detection for Temporal Data: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014.
[123] M.O. Rabin. How To Exchange Secrets with Oblivious Transfer. Cryptology ePrint Archive, Report 2005/187, 2005.
[124] F. Angiulli and F. Fassetti. Detecting Distance-Based Outliers in Streams of Data. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 811–820, 2007.
[125] M. Kontaki, A. Gounaris, A.N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous Monitoring of Distance-Based Outliers over Data Streams. In *2011 IEEE 27th International Conference on Data Engineering*, pages 135–146, 2011.
[126] K. Bhaduri, M.D. Stefanski, and A.N. Srivastava. Privacy-Preserving Outlier Detection through Random Nonlinear Data Distortion. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 41(1):260–272, 2011.
[127] J. Böhler, D. Bernau, and F. Kerschbaum. Privacy-Preserving Outlier Detection for Data Streams. In *Data and Applications Security and Privacy XXXI*, pages 225–238, 2017.
[128] S.M. Erfani, Y.W. Law, S. Karunasekera, A.C. Leckie, and M. Palaniswami. Privacy-Preserving Collaborative Anomaly Detection for Participatory Sensing. In *Advances in Knowledge Discovery and Data Mining*, pages 581–593, 2014.
[129] A. Alabdulatif, H. Kumarage, I. Khalil, and X. Yi. Privacy-Preserving Anomaly Detection in Cloud with Lightweight Homomorphic Encryption. *Journal of Computer and System Sciences*, 90:28–45, 2017.
[130] L. Li, L. Huang, W. Yang, X. Yao, and A. Liu. Privacy-Preserving LOF Outlier Detection. *Knowledge and Information Systems*, 42(3):579–597, 2015.
[131] J. Vaidya and C. Clifton. Privacy-Preserving Outlier Detection. In *Proceedings of 4th IEEE International Conference on Data Mining*, pages 233–240, 2004.
[132] L. Cao et al. Scalable Distance-Based Outlier Detection over High-Volume Data Streams. In *2014 IEEE 30th International Conference on Data Engineering*, pages 76–87, 2014.
[133] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. In *2017 IEEE Symposium on Security and Privacy*, pages 655–672, 2017.
[134] L. Tran, L. Fan, and C. Shahabi. Distance-Based Outlier Detection in Data Streams. *Proceedings of the VLDB Endowment*, 9(12):1089–1100, 2016.
[135] AgentVi. innoVi Enterprise. https://www.agentvi.com/products/innovi/innovi-enterprise/ [online], 2018.
[136] Microsoft. Tracking a Building's Vital Signs to Keep it Safe and Healthy. https://customers.microsoft.com/en-us/story/tracking-a-buildings-vital

-signs-to-keep-it-safe-and-h [online], 2016.
[137] Amazon. Amazon Kinesis. https://aws.amazon.com/kinesis/ [online], 2018.
[138] Microsoft. Time Series Anomaly Detection. https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/time-series-anomaly-detection#how-to-configure-time-series-anomaly-detection [online], 2018.
[139] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *2013 IEEE Symposium on Security and Privacy*, pages 334–348, 2013.
[140] P.K. Chan and M.V. Mahoney. Modeling Multiple Time Series for Anomaly Detection. In *Proceedings of the 5th IEEE International Conference on Data Mining*, pages 90–97, 2005.
[141] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. *ACM SIGMOD Record*, 29(2):427–438, 2000.
[142] M. Luby and C. Rackoff. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM Journal on Computing*, 17:373–386, 1988.
[143] R. Bost, R.A. Popa, S. Tu, and S. Goldwasser. Machine Learning Classification over Encrypted Data. In *NDSS*, 2015.
[144] D. Dua and C. Graff. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml [online], 2017.
[145] C-K. Chu, W.T. Zhu, J. Han, J.K. Liu, J. Xu, and J. Zhou. Security Concerns in Popular Cloud Storage Services. *IEEE Pervasive Computing*, 12(4):50–57, 2013.
[146] K. Liang, C. Su, J. Chen, and J.K. Liu. Efficient Multi-Function Data Sharing and Searching Mechanism for Cloud-Based Encrypted Data. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 83–94, 2016.
[147] J.K. Liu, M.H. Au, W. Susilo, K. Liang, R. Lu, and B. Srinivasan. Secure Sharing and Searching for Real-time Video Data in Mobile Cloud. *IEEE Network*, 29(2):46–50, 2015.
[148] D.X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
[149] E. Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003.
[150] D. Cash et al. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.
[151] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M-C. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *European Symposium on Research in Computer Security*, pages 123–145, 2015.
[152] S. Sun, J.K. Liu, A. Sakzad, R. Steinfeld, and T.H. Yuen. An Efficient Non-interactive Multi-client Searchable Encryption with Support for Boolean Queries. In *European Symposium on Research in Computer Security*, pages 154–172, 2016.
[153] M.S. Islam, M. Kuzu, and M. Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*, 2012.
[154] J. Katz, A. Sahai, and B. Waters. Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. *Journal of cryptology*, 26(2):191–224, 2013.
[155] T. Okamoto and K. Takashima. Adaptively Attribute-Hiding (Hierarchical) Inner Product Encryption. In *Advances in Cryptology – EUROCRYPT 2012*, pages 591–608, 2012.
[156] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. In *Advances in Cryptology – EUROCRYPT 2004*, pages 506–522, 2004.
[157] R. Cramer and V. Shoup. Signature Schemes Based on the Strong RSA Assumption. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 46–51, 1999.
[158] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC

Press, 2014.
[159] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet mathematics*, 1(4):485–509, 2004.
[160] S. Blake-Wilson, N. Bolyard, V.Gupta, C. Hawk, and B. Moeller. RFC4492: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). *RFC4492, Internet Engineering Task Force*, 2006.
[161] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.
[162] Apache. Hbase. `https://hbase.apache.org` [online], 2015.
[163] L. George. Advanced HBase Schema Design. Presented at Hadoop World, 2011.
[164] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[165] IBTA. InfiniBand Specification. `http://www.infinibandta.org/` [online], 2017.
[166] Cloudera. CDH Overview. `https://www.cloudera.com/documentation/enterprise/5-2-x/topics/cdh_intro.html` [online], 2018.
[167] Wikimedia Foundation. Wikimedia downloads. `https://dumps.wikimedia.org` [online], 2017.
[168] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M.N. Garofalakis. Practical Private Range Search Revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198, 2016.
[169] P. Mishra, R. Poddar, J. chen, A. Chiesa, and R.A. Popa. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy*, pages 279–296, 2018.
[170] G. Amjad, S. Kamara, and T. Moataz. Forward and Backward Private Searchable Encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.
[171] V. Vo, S. Lai, X. Yuan, S-F. Sun, S. Nepal, and J.K. Liu. Accelerating Forward and Backward Private Searchable Encryption Using Trusted Execution. ArXiv e-prints, arXiv:2001.03743, 2020.
[172] S. Lai et al. Towards Practical Encrypted Network Traffic Pattern Matching for Secure Middleboxes. ArXiv e-prints, arXiv:2001.01848, 2020.
[173] J. Sherry, C. Lan, R.A. Popa, and S. Ratnasamy. Blindbox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 213–226, 2015.
[174] N. Desmoulins, P-A. Fouque, C. Onete, and O. Sanders. Pattern Matching on Encrypted Streams. In *Advances in Cryptology – ASIACRYPT 2018*, pages 121–148, 2018.