GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search

Shangqi Lai*
Monash University
Melbourne, Australia
shangqi.lai@monash.edu

Xingliang Yuan
Monash University
Melbourne, Australia
xingliang.yuan@monash.edu

Shi-Feng Sun*
Monash University
Melbourne, Australia
shifeng.sun@monash.edu

Joseph K. Liu†
Monash University
Melbourne, Australia
joseph.liu@monash.edu

Yuhong Liu
Santa Clara University
Santa Clara, U.S.
yhliu@scu.edu

Dongxi Liu
Data61, CSIRO
Syndey, Australia
dongxi.liu@data61.csiro.au

ABSTRACT

In this paper, 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 of users.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**; **Social network security and privacy**; **Privacy-preserving protocols**;

KEYWORDS

Social Search, Graph Database, Encrypted Query Processing

ACM Reference Format:
Shangqi Lai, Xingliang Yuan, Shi-Feng Sun, Joseph K. Liu, Yuhong Liu, and Dongxi Liu. 2019. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3321705.3329803

*Also with Data61, CSIRO, Melbourne, Australia.
†Corresponding author.

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6784-4/19/07...$15.00.
DOI: https://doi.org/10.1145/3321705.3329803

1 INTRODUCTION

Data breaches in online social networks (OSNs) affect billions of individuals and raise critical privacy concerns across the entire society [27, 36]. Besides, driven by the demands on huge storage and computation resources, OSN service providers utilise public commercial clouds as their back-end data storage [3, 4, 25], which further broadens the attack plane [53]. Therefore, there is an urgent call to improve the control of data confidentiality for cloud providers [35, 38, 64], in particular for current OSN services. The prevailing consensus to prevent data leakage is encryption. However, this approach impairs the functionality of social search, a key enabler for quality OSN services [56]. Social search allows users to search 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 large volume of contents which will be queried for social search related services [22]. 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 [22]), queries of social search contains 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 [10, 44] 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 circuits based framework [44] 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 [20, 29], and shortest distance queries [40, 59, 61, 62]. Unfortunately, the above algorithms are limited for or different from the functionality of social search queries.

Contributions. To bridge the gap, in this paper, 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, and 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 the Facebook's social search system Unicorn [22], 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 [67] for setup (data partition and encryption), Redis [52] as the storage back-end, and uses Apache Thrift [55] to implement the query planner and query processing logic.

Our comprehensive evaluation on the Youtube dataset [41] 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 paper is structured as follows. We discuss related work in Section 2. After that, we describe the system overview in Section 4, and present the encrypted and distributed graph data model and the design of atomic operations in Section 5. In Section 6, we introduce the realisation of privacy-preserving social search queries and their security. Next, we describe our prototype implementation in Section 7, and evaluate the performance in Section 8. We give a conclusion in Section 9.

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 [20] is proposed in the framework of SSE and supports adjacency and neighbouring queries. Some recent work is proposed to support privacy-preserving subgraph queries [15, 19]. 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 oblivious RAM [62], structured encryption [40], or Garbled Circuit [59, 61]. To implement more complicated algorithms, protocols are proposed to use secret sharing and homomorphic encryption for Breadth-first search (BFS) [10], PageRank [63], and approximate eigen-decomposition [54]. We stress that the above work targets on different query functionality other than social search. Note that a recent framework named GarphSC [44] can generate data-oblivious Garbled Circuit (GC) 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 [47–50, 66]. CryptDB [50] 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 [48] proposes a Bloom Filter based index and leverages GC to evaluate arbitrary boolean queries with keywords and ranges. Arx [49] follows the design of CryptDB to support SQL queries, but it uses SSE and GC to reduce the leakage from PPE. Seabed [47] uses additively symmetric homomorphic encryption (ASHE) to perform efficient aggregation over the encrypted data, and develops a schema with padding to mitigate the inference attack [43]. EncKV [66] adapts SSE and ORE 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 [21, 22, 39] (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 form, which are unable to address privacy concerns of sensitive data leakage. Authenticated graph query [26] is proposed to verify the correctness of graph queries, which could be a complementary work to prevent attacks from malicious adversaries.

3 BACKGROUND

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 [22], it is normally represented via a set of adjacency lists. Like [22], we refer to these adjacency lists as *posting lists*.

Formally, the social graph is an edge-labeled and directed graph $G = (V, E)$, where $V = \{v_1, v_2, ...\}$ is the entity set and $E = \{e_1, e_2, ...\}$ 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 $e = (u, v, \textbf{edge-type})$ which consists of its egress, ingress nodes $(u, v \in V)$ 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 $\textbf{edge-type:id}_u$. For example, the user may use $\textbf{friend:id}_i$ to get the posting list of user i's friends.

3.2 Oblivious Cross-Tags (OXT) Protocol

Oblivious Cross-Tags (OXT) Protocol [18] is an SSE protocol, which proceeds between client C and server S. It provides an efficient way to perform conjunctive queries in 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 'TSet(w)', is built to point to the set DB(w) of all entity identifiers **id**s associating with w. Each TSet(w) is identified by an indexing term called **stag**(w), and all **id** values in TSet(w) are encrypted via a secret key K_w. Both **stag**(w) and K_w 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 $h(\mathbf{id}, w)$ (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' **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 TSet.Retrieve(TSet, **stag**(w_1)) and retrieve TSet(w_1) from the TSet. In addition, C sends 'intersection tokens' **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, $2 \le i \le n$ (called 'x-terms'). The xtraps allow the server to evaluate the cryptographic hash function of pairs $(\mathbf{id}, w_i)$ without knowing either keyword w_i or **id**. S checks the existence of $h(\mathbf{id}, w_i)$ in XSet and filters the 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 $\{w_i\}_{1 \le i \le n}$ to the client. C finally uses K_w to recover the **id**s of entities.

As mentioned in [18], the security of OXT parameterised by a leakage function $\mathcal{L}_{OXT} = (N, \phi, \bar{s}, \mathrm{SP}, \mathrm{XP}, \mathrm{RP}, \mathrm{IP})$. It depicts what an adversary is allowed to learn about the database and queries via executing OXT protocol. Informally, considering a vector of queries $\mathsf{q} = (\mathsf{s} \wedge \phi(\mathsf{x}_2, ..., \mathsf{x}_n))$, which consists of a vector of s-terms s, a vector of boolean formulas ϕ, 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 $(\mathbf{id}, w)$ pairs.
- ϕ: The boolean formulae that the client wishes to query.
- $\bar{s}$: The repeat pattern in s.
- 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 $(\mathsf{s}, \mathsf{x}_i)$, $2 \le i \le n$.
- IP: The set of result **id** both existing in the posting lists of s$[i]$ and s$[j]$, which is only revealed when two queries q$[i]$, q$[j]$, $i \ne j$ have different s-terms but same x-terms.

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 [8]. 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 [65], and its security model has been formalised in [9]. 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) [2] protocol 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 two-party setting is given in [37].

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 parties 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.

4 SYSTEM OVERVIEW

4.1 System Architecture

As shown in Figure 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 [3] and Instagram [25], 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 [5]. GraphSE^2 aims to improve the protection of data confidentiality at

[1]The scheme proposed in [28] supports disjunctive queries, but it consumes large storage space.

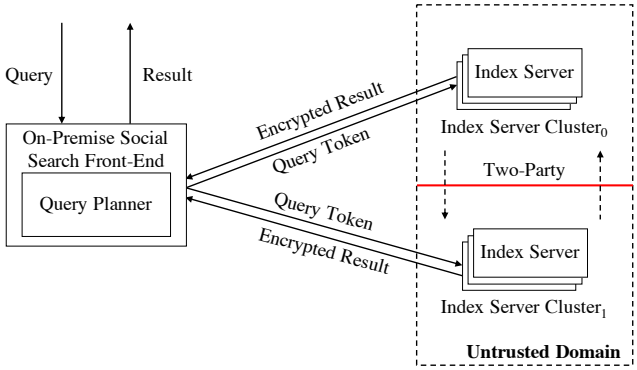


Figure 1: System architecture overview.

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 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 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 [30] and applied in many existing studies [6, 42, 46]. 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 [42].

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 [18]). The second stage requires a combination of different computations to further analyse user contents. For example, collaborative filtering [14] 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 [24, 42]. 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 (GC) 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.

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 **id**s 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

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

Table 1: Supported social search operators in GraphSE^2 and its essential atom operations.

Query operator	Example(from [22])	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**)	√	√	√	√

graph when the data storage back-end of the social search service is deployed at an untrusted domain.

4.4 Query Operators

GraphSE^2 follows a typical plaintext social search system [22] to define the operators (see Table 1).

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 **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 [22], 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 4, 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 [14] 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, **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.

5 THE PROPOSED SYSTEM

We give a list of needed notations in our system construction and security analysis in Table 2. The detailed definitions of preliminaries we used are given in Section 3.

Table 2: Notations and Terminologies

Notation	Meaning
id	the unique identifier of entity
$e_{\mathbf{id}}$	the encrypted entity **id**
t	an indexing term in the form of **edge-type:id**$_u$
DB	an inverted indexed database $\{(t, \{(\text{sort-key}, \mathbf{id})\})\}$
DB(t)	a list of $\{(\text{sort-key}, \mathbf{id})\}$ indexed by t
$\{\mathbf{E}\}$	the encrypted posting list with $(\langle\text{sort-key}\rangle^A, e_{\mathbf{id}})$ pairs
P_i	the i-th party in GraphSE^2 ($i \in \{0, 1\}$)
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

5.1 Encrypted Graph Data Model

To support social search operations in [22] on an encrypted social graph (see Section 3.1 for details), GraphSE^2 creates the OXT index (i.e., TSet and XSet, see Section 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 $\mathcal{ISC}$: $(\mathbf{stag}(\mathbf{edge\text{-}type:id}_u), \{\mathbf{E}\})$. 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 $\mathbf{E}_{\mathbf{id}} = (\langle\text{sort-key}\rangle^A, e_{\mathbf{id}})$, which keeps the encryption $e_{\mathbf{id}}$ 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 $(\mathbf{id}, \mathbf{edge\text{-}type:id}_u)$ pairs to generate an XSet for complex set operations.

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 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., [39]) can also be applied to shard the social graph. The design has three advantages in the context of distributed environment. First, it maintains the availability in the event of server failure. Furthermore, the sharding strategy enables

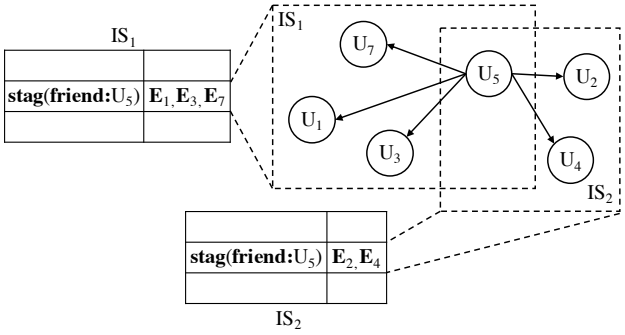


Figure 2: Our encrypted and distributed data model, the arrows indicate the friend relationships between users.

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.

5.3 Atomic Operations

As mentioned in Section 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.

5.3.1 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. GraphSE^2 utilises TSet operations to implement this operation: it generates a search token for the search keyword in the TSet and uses the token to retrieve the encrypted posting list as the return. The detailed algorithm for *Index Access* operation is given in Appendix A.1. *Index Access* operation can be easily extended to run in parallel. More specifically, $\mathcal{SF}$ broadcasts the search token to all $\mathcal{IS}$s. After that, each $\mathcal{IS}$ uses the token 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 [18]. It ensures *Index Access* only leaks the number of edges in the encrypted social graph.

5.3.2 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 OXT protocol to support this atomic operation, but some of the other SSE protocols supporting conjunctive queries (e.g. [33]) 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 $t_1 \wedge \phi(t_2, ..., t_n)$, where t_1 is the 's-term', and $\phi(t_2, ..., t_n)$ is an arbitrary boolean expression [18]. We provide the detailed algorithm for *Set Operations* in Appendix A.1. The Boolean Query algorithm can be utilised to enable set operations of social search queries as shown in Table 1, which will be discussed in the following section.

Security. In cryptographic terms, the OXT protocol is proved to be $\mathcal{L}_{\text{OXT}}$-semantically-secure against adaptive attacks, where $\mathcal{L}_{\text{OXT}}$ is the leakage function defined in [18]. 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 3.2 for more details.

5.3.3 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 for detailed definition) to compute addition and multiplication under 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 [42] 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

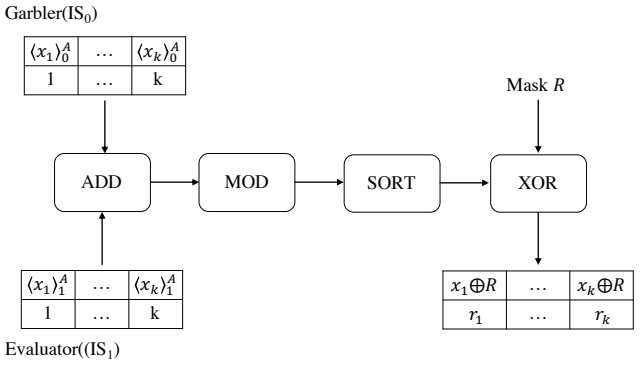


Figure 3: Local sorting process for one pair of garbler and evaluator. The input of the garbler is at the top, and the input/output of the evaluator is on the bottom.

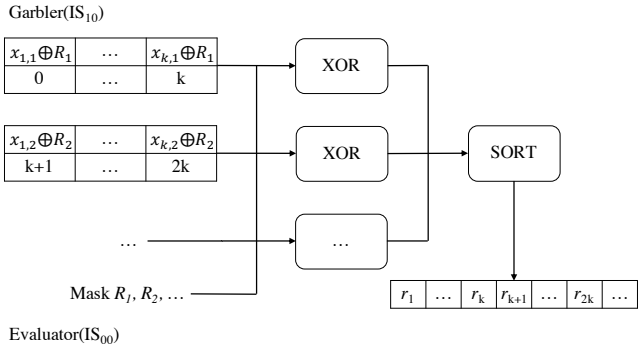


Figure 4: Global sorting process in the coordinators. The input of the garbler is the masked score vector with a payload that indicates the position of the score in vector, and the input of the evaluator is the random masks.

size of $\langle \mathbf{X}\rangle^A$ is s ∗ t and we denote each element in $\langle \mathbf{X}\rangle^A$ as $\langle x_{i,j}\rangle_0^A$, $i = 1, ..., s$ and $j = 1, ..., t$. In addition, we assume each column of $\langle \mathbf{Y}\rangle^A$ has t elements, which are denoted as $\langle y_j\rangle_1^A$, $j = 1, ..., t$. The computation process is listed as follows:

- $\mathcal{IS}_0$ runs a correlated-OT protocol (COT) [2], 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 $COT(r_b, f_b(r_b); \langle y_j\rangle_1^A[b])$ 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 $\Sigma_{j=1}^t \langle\langle x_{i,j}\rangle_0^A \cdot \langle y_j\rangle_1^A\rangle^A$. 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 [51], 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.

5.3.4 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 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 [24] to switch from additive sharing to Yao's sharing. It then adopts the sorting network [7] 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 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 $\mathrm{x} = \{\langle x_i\rangle^A\}_{i=1}^{\mathrm{k}}$, 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 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.
- 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 protocol. 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 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 protocol.
- 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 [9] and OT [2] 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 output. Finally, for the output of GlobalSort(x), only the decode table of rank is sent to the evaluator, which also ensures that the evaluator only learns global rank without knowing the actual ranking score.

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 1). In the following sections, we introduce the operators in GraphSE^2 and their security properties.

6.1 Graph Operators

GraphSE^2 uses atomic operations in Section 5.3 to realise all operators in Table 1. 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 5.3, conjunctive queries with nested queries (e.g., $t_1 \wedge \phi(t_2, ..., t_n)$) are processed by evaluating the boolean expression ϕ 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 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 1, it aims to find the friends of $\mathbf{user}_3$, who are neither $\mathbf{user}_1$'s friends nor $\mathbf{user}_2$'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, **difference** operator excludes the results that satisfy the boolean expression ϕ. Therefore, the s-expression with **difference** operator is represented as $t_1 \wedge \neg\phi(t_2, ..., t_n)$. Comparing with the **and** operator, it returns the results only if the boolean expression ϕ returns false instead of true.

or. The complexity of the original approach for processing disjunctive queries is linear to the size of database [18]. 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 **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 **or** operator. The correctness of the above approach can be easily proved by the set operation: $t_1 \vee t_2 \vee t_3 = (t_1 \backslash (t_2 \vee t_3)) \vee (t_2 \backslash t_3) \vee t_3$.

In general, a disjunctive s-expression with n indexing terms can be rewritten as $n-1$ s-expressions with **difference** operator and 1 s-expressions with **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 $\max(\{|\mathrm{TSet}(t_i)|\}_{i=1}^{n})$.

6.2 Apply Operator

The **apply** is a unique operator in Unicorn [22], 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:id**$_{1,1}$ ... **friend:id**$_{1,N}$) to get the entities that are more than one edge away from the user $\mathbf{id}_1$ in the encrypted graph-structured data. We discuss the detailed implementation of the **apply** in Appendix A.2.

The **apply** operator processes 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 full version [34]).

6.3 Security Analysis

In Section 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 [18, 23] 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 ϕ and a tuple of indexing terms $(t_1, t_2, ..., t_n)$.

Due to the page limit, we provide an overview of the security analysis in Appendix B.1, and the leakage function is given in Appendix B.2, the detailed proof is in the full version [34].

7 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. Specifically, we use the symmetric primitives, i.e., AES-CMAC and AES-CBC, from Bouncy Castle Crypto APIs [58]. In addition, we use a built-in curve from Java Pairing-based Cryptography (JPBC) [16] 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. Oblivious Transfer and Garbled Circuit are implemented by using FlexSC [60]. It implements the extended OTs in [2] 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 [1]. 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 [67] 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 [52] 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 4.1. Thrift thread pool proxy [55] 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 **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 [11] to load the XSet into memory during the startup process. In our prototype, we deploy the Bloom filter from Alexandr Nikitin as it is the fastest Bloom filter implementation for JVM [45]. 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.

Table 3: Statistics of Youtube social network dataset.

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

8 EXPERIMENTAL EVALUATIONS

8.1 Setup

Platform. We deploy the *index server daemons* in a cluster ($\mathcal{ISC}$) with 6 virtual machine instances in the Microsoft Azure platform. All VMs 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 VMs are installed with Ubuntu Server 16.04LTS.

Dataset. We use a Youtube dataset from [41], which is an anonymised Youtube user-to-user links and user group memberships network dataset. The detailed statistical summary is given in Table 3. 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 **id**s. 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 hash function to generate **xtag** 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 **stag** to the $\mathcal{IS}$ as query token. Nonetheless, we still use the PRF value of indexing term and the block counter 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 [22].

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

8.2 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 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 5). The reason is that GraphSE^2 also keeps the number of blocks of each posting list (see Section 7) to accelerate the tuple retrieving process[3]. As shown in Figure 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.

For XSet storage overhead, GraphSE^2 increases it by 17x (1.5GB versus 90MB), mostly due to the fact that the size group element is much larger than a PRF value. But the Bloom Filter successfully saves the memory consumption in runtime, because the size of Bloom Filter only depends on the false positive rate and the number of total elements inside (the number of edges in our system) [11], 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 [22]), and we further choose several variable indexing terms **v** from **friend** edges with selectivity from 1 to 502. Figure 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 TSet(**v**) should be 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 a query related to the moderate users with a tiny latency, and it is also able to reply query about popular users with a slightly bigger but still modest delay.

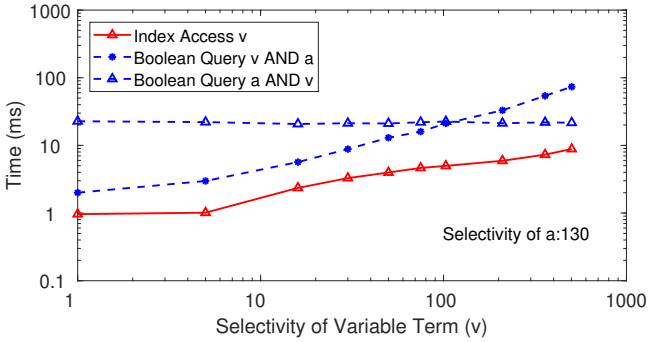


Figure 6: Query delay for two-keyword set queries.

The secure addition and multiplication is 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 8a and 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 Garbled Circuit, we provide a benchmark about the size of the circuit and the corresponding evaluation time. The results are listed in Table 4. 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). 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

[3]If the size of 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.

Table 4: 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

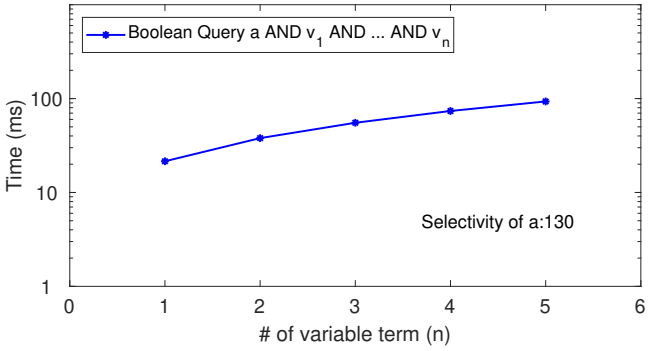
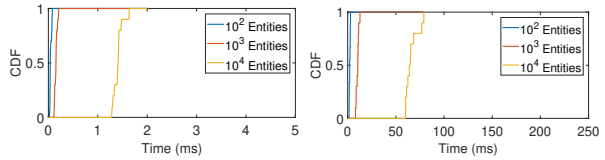


Figure 7: Query delay for multiple-keyword set queries.



(a) Addition (b) Multiplication

Figure 8: The execution time for addition and multiplication operations on two vectors with 10^2, 10^3, 10^4 entities.

terms $\{\mathbf{v_n}\}, \mathbf{n} \in [1, 5]$ as x-term. Figure 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 4. It shows that for an average user with approximately 130 friends, the two-party only requires to transmits 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 9 and Table 5 show the throughput test results for GraphSE^2 and baseline. In all the cases, we group our 6 VM instances into 2 clusters with 3 VMs to fulfil the two-party settings. All VMs 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

Table 5: 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

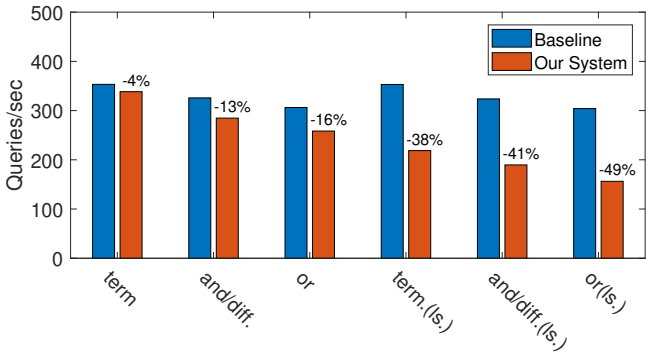


Figure 9: Throughput of different types of Query Operators with 10000 concurrent clients running under GraphSE^2 and baseline, all operators except term have two keywords. Diff. stands for difference operator; ls. stands for locally sorted in $\mathcal{IS}$ after applying the operators.

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 5), 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%).

9 CONCLUSION

This paper 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 mixing protocol with additive sharing and 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.

ACKNOWLEDGMENTS

The work was supported in part by ARC Discovery Project grant DP180102199 and a Microsoft Azure grant.

REFERENCES

[1] Apache. 2015. Hadoop. https://hadoop.apache.org[online]. (2015).
[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. 2013. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *ACM CCS'13*.
[3] AWS. 2018. AWS Case Study: Airbnb. https://aws.amazon.com/solutions/case-studies/airbnb/[online]. (2018).
[4] AWS. 2018. AWS Case Study: PIXNET. https://aws.amazon.com/cn/solutions/case-studies/pixnet/[online]. (2018).
[5] AWS. 2018. AWS Outposts: Run AWS Infrastructure On-Premises for A Truly Consistent Hybrid Experience. https://aws.amazon.com/outposts/[online]. (2018).
[6] F. Baldimtsi and O. Ohrimenko. 2015. Sorting and Searching Behind the Curtain. In *FC'15*.
[7] K.E. Batcher. 1968. Sorting Networks and their Applications. In *ACM SJCC'68*.
[8] D. Beaver. 1991. Efficient Multiparty Protocols using Circuit Randomization. In *CRYPTO'91*.
[9] M. Bellare, V.T. Hoang, and P. Rogaway. 2012. Foundations of Garbled Circuits. In *ACM CCS'12*.
[10] M. Blanton, A. Steele, and M. Alisagari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ACM AISACCS'13*.
[11] B.H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[12] R. Bost, B. Minaud, and O. Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *ACM CCS'17*.
[13] R. Bost and Fouque P-A. 2017. Thwarting Leakage Abuse Attacks against Searchable Encryption–A Formal Approach and Applications to Database Padding. Cryptology ePrint Archive, Report 2011/1060. (2017).
[14] J.S. Breese, D. Heckerman, and C. Kadie. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *UAI'98*.
[15] N. Cao, Z. Yang, C. Wang, K. Ren, and W. Lou. 2011. Privacy-Preserving Query over Encrypted Graph-Structured Data in Cloud Computing. In *IEEE ICDCS'11*.
[16] A. De Caro and V. Iovino. 2011. JPBC: Java Pairing Based Cryptography. In *IEEE SCC'11*. 850–855.
[17] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS'15*.
[18] D. Cash, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO'13*.
[19] Z. Chang, L. Zou, and F. Li. 2016. Privacy Preserving Subgraph Matching on Large Graphs in Cloud. In *ACM SIGMOD'16*.
[20] M. Chase and S. Kamara. 2010. Structured Encryption and Controlled Disclosure. In *AISACRYPT'10*.
[21] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang. 2016. Nxgraph: An Efficient Graph Processing System on A Single Machine. In *IEEE ICDE'16*.
[22] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, et al. 2013. Unicorn: A System for Searching the Social Graph. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1150–1161.
[23] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. 2011. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security* 19, 5 (2011), 895–934.
[24] D. Demmler, T. Schneider, and M. Zohner. 2015. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS'15*.
[25] Instagram Engineering. 2018. 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).
[26] M.T. Goodrich, R. Tamassia, and N. Triandopoulos. 2011. Efficient Authenticated Data Structures for Graph Connectivity and Geometric Search Problems. *Algorithmica* 60, 3 (2011), 505–552.
[27] Information is Beautiful. 2018. World's Biggest Data Breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/[online]. (2018).
[28] S. Kamara and T. Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-Linear Complexity. In *EUROCRYPT'17*.
[29] S. Kamara, T. Moataz, and O. Ohrimenko. 2018. Structured Encryption and Leakage Suppression. In *CRYPTO'18*.
[30] S. Kamara, P. Mohassel, and M. Raykova. 2011. Outsourcing Multi-Party Computation. Cryptology ePrint Archive, Report 2011/272. (2011).
[31] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS'16*.
[32] E.M. Kornaropoulos, C. Papamanthou, and R. Tamassia. 2019. Data Recovery on Encrypted Databases with K-Nearest Neighbor Query Leakage. In *IEEE S&P'19*.
[33] S. Lai, S. Patranabis, A. Sakzad, J.K. Liu, D. Mukhopadhyay, R. Steinfeld, et al. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *ACM CCS'18*.
[34] S. Lai, Y. Yuan, S-F. Sun, J.K. Liu, Y. Liu, and D. Liu. 2019. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. https://shangqimonash.github.io/assets/pdf/GraphSE.pdf[online]. (2019).
[35] J. Li, L. Zhang, J. K. Liu, H. Qian, and Z. Dong. 2016. Privacy-Preserving Public Auditing Protocol for Low-Performance End Devices in Cloud. *IEEE Transactions on Information Forensics and Security* 11, 11 (2016), 2572–2583.
[36] K. Liang, J.K. Liu, R. Lu, and D. S. Wong. 2015. Privacy Concerns for Photo Sharing in Online Social Networks. *IEEE Internet Computing* 19, 2 (2015), 58–63.
[37] Y. Lindell and B. Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-party Computation. *Journal of Cryptology* 22, 2 (2009), 161–188.
[38] J. K. Liu, K. Liang, W. Susilo, J. Liu, and Y. Xiang. 2016. Two-Factor Data Security Protection Mechanism for Cloud Storage System. *IEEE Transactions on Computers* 65, 6 (2016), 1992–2004.
[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012).
[40] X. Meng, S. Kamara, K. Nissim, and G. Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *ACM CCS'15*.
[41] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, and B. Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC'07*.
[42] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P'17*.
[43] M. Naveed, S. Kamara, and C.V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM CCS'15*.
[44] K. Nayak, X. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S&P'15*.
[45] A. Nikitin. 2017. Bloom Filter Scala. https://alexandrnikitin.github.io/blog/bloom-filter-for-scala/[online]. (2017).
[46] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. 2013. Privacy-Preserving Matrix Factorization. In *ACM CCS'13*.
[47] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, et al. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX OSDI'16*.
[48] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.G. Choi, et al. 2014. Blind seer: A Scalable Private DBMS. In *IEEE S&P'14*.
[49] R. Poddar, T. Boelter, and R.A. Popa. 2016. Arx: A Strongly Encrypted Database System. Cryptology ePrint Archive, Report 2016/591. (2016).
[50] R.A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP'11*.
[51] P. Pullonen, D. Bogdanov, and T. Schneider. 2012. The Design and Implementation of A Two-Party Protocol Suite for Sharemind 3. http://tubiblio.ulb.tu-darmstadt.de/61259/[online]. (2012).
[52] Redis Labs. 2017. Redis. https://redis.io[online]. (2017).
[53] K. Ren, C. Wang, and Q. Wang. 2012. Security Challenges for the Public Cloud. *IEEE Internet Computing* 16, 1 (2012), 69–73.
[54] S. Sharma, J. Powers, and K. Chen. 2018. PrivateGraph: Privacy-Preserving Spectral Analysis of Encrypted Graphs in the Cloud. *IEEE Transactions on Knowledge and Data Engineering* (2018).
[55] M. Slee, A. Agarwal, and M. Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper* 5, 8 (2007).
[56] D. Sullivan. 2012. 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).
[57] S-F. Sun, X. Yuan, J.K. Liu, R. Steinfeld, A. Sakzad, V. Vo, et al. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *ACM CCS'18*.
[58] The Legion of the Bouncy Castle. 2007. Bouncy Castle Crypto APIs. https://www.bouncycastle.org[online]. (2007).
[59] Q. Wang, K. Ren, M. Du, Q. Li, and A. Mohaisen. 2017. SecGDB: Graph Encryption for Exact Shortest Distance Queries with Efficient Updates. In *FC'17*.
[60] X. Wang. 2018. FlexSC. https://github.com/wangxiao1254/FlexSC[online]. (2018).
[61] D.J. Wu, J. Zimmerman, J. Planul, and J.C. Mitchell. 2016. Privacy-Preserving Shortest Path Computation. ArXiv e-prints, arXiv:1601.02281. (2016).
[62] D. Xie, G. Li, B. Yao, X. Wei, X. Xiao, Y. Gao, et al. 2016. Practical Private Shortest Path Computation Based on Oblivious Storage. In *IEEE ICDE'16*.
[63] P. Xie and E. Xing. 2014. CryptGraph: Privacy Preserving Graph Analytics on Encrypted Graph. ArXiv e-prints, arXiv:1409.5021. (2014).
[64] X. Yang, X. Huang, and J. K. Liu. 2016. Efficient Handover Authentication with User Anonymity and Untraceability for Mobile Cloud Computing. *Future Generation Computer Systems* 62 (2016), 190–195.
[65] A.C. Yao. 1982. Protocols for Secure Computations. In *IEEE SFCS'82*.
[66] X. Yuan, Y. Guo, X. Wang, C. Wang, B. Li, and X. Jia. 2017. EncKV: An Encrypted Key-Value Store with Rich Queries. In *ACM AISACCS'17*.
[67] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *USENIX Workshop HotCloud'10*.
[68] Y. Zhang, J. Katz, and C. Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security'16*.

A ALGORITHMS FOR ATOMIC OPERATIONS AND QUERY

A.1 Index Access and Set Operations

Algorithm 1 Index Access

Input: TSet, Indexing Term t
Output: Encrypted Result TSet(t)
1: **function** IndexAccess(TSet, t)
2: $\mathcal{SF}$ inputs indexing term t, and $\mathcal{IS}$ inputs TSet;
3: $\mathcal{SF}$ computes $\tau \leftarrow$ **stag**(t);
4: $\mathcal{SF}$ sends τ to $\mathcal{IS}$;
5: $\mathcal{IS}$ computes TSet(t) $\leftarrow$ TSet.Retrieve(TSet, τ);
6: **return** TSet(t);
7: **end function**

Algorithm 1 outlines the searching procedure using TSet operations. On receiving the search keyword, $\mathcal{SF}$ firstly generates a search token τ, which is **stag**(t) of the indexing term t. $\mathcal{IS}$ can use τ to search TSet and get the encrypted posting list TSet(t) as the return.

Algorithm 2 Boolean Query

Input: TSet, XSet, Query $(t_1 \wedge \phi(t_2, ..., t_n))$ with s-term t_1
Output: Encrypted Result R
1: **function** BooleanQuery(TSet, XSet, $\bar{t}, \phi$)($\bar{t}$ is the indexing term list $(t_1, ..., t_n)$, and ϕ is an arbitrary boolean expression)
2: $\mathcal{SF}$ inputs indexing term $\bar{t}, \phi$, and $\mathcal{IS}$ inputs TSet, XSet;
3: $\mathcal{SF}$ initialise a boolean expression $\hat{\phi}(v_2, ..., v_n)$ from ϕ and sends it to $\mathcal{IS}$;
4: $\mathcal{SF}, \mathcal{IS}$ runs TSet(t_1) $\leftarrow$ IndexAccess(TSet, t_1);
5: $\mathcal{IS}$ parses TSet(t_1) to {**E**};
6: **for** $l = 2 : n$ **do**
7: $\mathcal{SF}$ computes **xtoken**(t_1, t_l);
8: $\mathcal{SF}$ sends **xtoken**(t_1, t_l) to $\mathcal{IS}$;
9: **end for**
10: $\mathcal{IS}$ initialises $R \leftarrow \{\}$;
11: **for** $c = 1 : |\{\mathbf{E}\}|$ **do**
12: **for** $l = 2 : n$ **do**
13: $\mathcal{IS}$ uses **xtoken**(t_1, t_l) to compute $h(\mathbf{id}_c, t_l)$;
14: $\mathcal{IS}$ lets $v_l = (h(\mathbf{id}_c, t_l) \in$ XSet);
15: **end for**
16: **if** $\hat{\phi}(v_2, ..., v_n) =$ '$True$' **then**
17: $\mathcal{IS}$ adds $\mathbf{E}_c$ in R;
18: **end if**
19: **end for**
20: **return** R;
21: **end function**

As shown in Algorithm 2, the extended protocol follows the basic steps of OXT 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 $\phi(t_2, ..., t_n)$. Specifically, $\mathcal{SF}$ substitutes all indexing terms t_i to boolean variables v_i ($i = 2, ..., n$) and generates a boolean function $\hat{\phi}(v_2, ..., v_n)$. $\mathcal{SF}$ then sends $\hat{\phi}(v_2, ..., v_n)$ to $\mathcal{IS}$. $\mathcal{IS}$ sets the value of v_i to the truth values of $h(\mathbf{id}_c, t_i) \in$ XSet. Then, it evaluates $\hat{\phi}(v_2, ..., v_n)$ and returns $\mathbf{E}_c$ as a result if $\hat{\phi}$ outputs true.

A.2 Apply Operator

Algorithm 3 Apply

Input: Outer Query Structure QS_O, Nested Query Structure QS_N, Array of **id**
Output: Encrypted Result O
1: **function** Apply(QS_O, QS_N, **id**)(in $\mathcal{SF}$)
2: **for** $i = 1 : QS_N.s.size$ **do**
3: $term \leftarrow QS_N.prefix||\mathbf{id}[i]$;
4: $QS_N.s[i] \leftarrow term$;
5: **end for**
6: $o \leftarrow$ Search($QS_N.s, QS_N.filter$);
7: **for** $i = 1 : o.size$ **do**
8: $term \leftarrow QS_O.prefix||o[i].id$;
9: $QS_O.s[i] \leftarrow term$;
10: **end for**
11: $O \leftarrow$ Search($QS_O.s, QS_O.filter$);
12: **return** O;
13: **end function**
Input: Query q, Result Filter f
Output: Encrypted Result r
1: **function** Search(q, f)(in $\mathcal{IS}$)
2: $r \leftarrow f(Execute(q))$;
3: **return** r
4: **end function**

The complete procedure of **apply** operator is shown in Algorithm 3. GraphSE^2 defines a query structure to construct **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** operator, $\mathcal{SF}$ 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.

B SECURITY ANALYSIS

B.1 Overview

The main idea of analysing the security of GraphSE^2 is similar to that in SSE scheme [18, 23]. 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 **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.

Due to the page limit, we only present the leakage function of GraphSE^2 in Appendix B.2. The formal security proof is given in the full version [34].

Discussion. Note that there exist some emerging threats against the building blocks of GraphSE^2. Regarding SSE, leakage-abuse attacks [17, 68] can help an attacker to explore the information learned during queries. To mitigate them, recent studies on padding countermeasures [13, 17] and forward/backward privacy [12, 57] 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 [31, 32] demonstrates that the underling 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.

B.2 Leakage Function

Recall the security definition from [18, 23]: The security of SSE is parameterised by a leakage function $\mathcal{L}_{\text{SSE}}$, which depicts the scope of information about data and queries that the adversary is allowed to learn through the interaction with server.

Therefore, we start by giving the leakage function of query operators in GraphSE^2. As shown in Section 5.3, all operators in GraphSE^2 inherit the leakage of OXT [18] (i.e., TSet leakage $\mathcal{L}_{\mathbf{T}}$ and OXT leakage $\mathcal{L}_{\text{OXT}}$).

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 $\mathcal{L}_{\text{R}}$ to capture this new leakage. In particular, $\mathcal{L}_{\text{R}}$ 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 $r_{\mathbf{id}}$ for every $e_{\mathbf{id}}$, where $r_{\mathbf{id}} \in \mathcal{N}$ is the rank of $e_{\mathbf{id}}$.

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 $\mathcal{L}_{\text{R}}$.