SPG: Structure-Private Graph Database via SqueezePIR

Ling Liang*
UC Santa Barbara
Santa Barbara, California, USA
lingliang@ucsb.edu

Jilan Lin*
UC Santa Barbara
Santa Barbara, California, USA
jilan@ucsb.edu

Zheng Qu
UC Santa Barbara
Santa Barbara, California, USA
zhengqu@ucsb.edu

Ishtiyaque Ahmad
UC Santa Barbara
Santa Barbara, California, USA
ishtiyaque@ucsb.edu

Fengbin Tu
UC Santa Barbara
Santa Barbara, California, USA
fengbintu@ucsb.edu

Trinabh Gupta
UC Santa Barbara
Santa Barbara, California, USA
trinabh@ucsb.edu

Yufei Ding
UC Santa Barbara
Santa Barbara, California, USA
yufeiding@ucsb.edu

Yuan Xie
Alibaba Group
Sunnyvale, California, USA
yuanxie@gmail.com

ABSTRACT

Many relational data in our daily life are represented as graphs, making graph application an important workload. Because of the large scale of graph datasets, moving graph data to the cloud becomes a popular option. To keep the confidential and private graph secure from an untrusted cloud server, many cryptographic techniques are leveraged to hide the content of the data. However, protecting only the data content is not enough for a graph database. Because the structural information of the graph can be revealed through the database accessing track.

In this work, we study the graph neural network (GNN), an important graph workload to mine information from a graph database. We find that the server is able to infer which node is processing during the edge retrieving phase and also learn its neighbor indices during GNN's aggregation phase. This leads to the leakage of the information of graph structure data. In this work, we present SPG, a structure-private graph database with SqueezePIR. Our SPG is built on top of Private Information Retrieval (PIR), which securely hides which nodes/neighbors are accessed. In addition, we propose SqueezePIR, a compression technique to overcome the computation overhead of PIR. Based on our evaluation, our SqueezePIR achieves 11.85× speedup on average with less than 2% accuracy loss when compared to the state-of-the-art FastPIR protocol.

PVLDB Reference Format:
Ling Liang, Jilan Lin, Zheng Qu, Ishtiyaque Ahmad, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. SPG: Structure-Private Graph Database via SqueezePIR. PVLDB, 16(7): 1615 - 1628, 2023.
doi:10.14778/3587136.3587138

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/liangling76/SqueezePIR.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587138

1 INTRODUCTION

Graph is a common data structure to represent relational data, such as social networks [39, 56, 59], financial transactions [23, 30, 38], and histories [48]. To help us better understand these data, graph algorithms are wildly used to mine information from graphs. With the success of deep learning, Graph Neural Network (GNN) [36, 41, 58, 71, 72] becomes one of the most important graph mining algorithms. Previous studies have demonstrated the effectiveness of GNN models in many applications, such as recommendations [66, 67], fraud detections [29, 46], and so on.

As the data in real-world graphs are explosively increasing, keeping the graph database locally can be extremely expensive. For example, the industry-level graphs are reported to be TB-level [42, 76]. Meanwhile, for graph datasets used in GNN workloads, each node usually has a node feature that is represented by a vector. These node feature vectors construct a feature table or embedding table, which can also be quite large with a size of TB-level [65, 75]. Therefore, cloud storage and retrieving then become an intriguing role to handle these large-scale graph databases.

However, privacy is the biggest concern when moving the graph to the cloud, as the data can be stolen by a curious server. One important way to protect the data is to apply cryptographic techniques [2, 17, 68], where the client can encrypt the data and upload it to the cloud. When the client wants to fetch a particular edge list or node feature, he/she can retrieve them back and decrypt them locally. Although encryption algorithms can help hide the data content, there is still critical information that can be learned by the server: the graph structure data. Most graph algorithms (including GNN) exhibit an aggregation procedure, meaning to process one vertex requires gathering the information from its neighbors. When the client retrieves the edge list and node features (even in an encrypted format), the server learns which node is processing and its neighbors' ID through the retrieving indices. Thus, the graph structure can be exposed to the server due to the processing nature of GNN models.

The leakage of structured information is vital [10, 24, 60] since it contains personal information in a social network or sensitive data in financial transactions. To avoid this leakage, we propose a structure-private database system for graph applications, namely SPG. Our SPG is built on top of Private Information Retrieval (PIR)

[9, 26, 28, 49], a cryptographic primitive that aims to access a database server obliviously. In other words, PIR allows retrieving an item from the server without revealing which item is retrieved. In SPG, when a graph algorithm tries to fetch the edge list of a node or its neighbors' node features, we apply PIR to hide the indices of the node and its neighbors in the database and thus ensure the privacy of the connectivity information in a graph.

The main challenge of building an SPG system is the expensive execution process of PIR. PIR has been known to have an *"all-for-one"* nature [44]. This means in order to hide the index of an item, we need to touch the entire database (i.e., all items). Otherwise, the server can still learn that the untouched items do not belong to the client's interest. Therefore, applying PIR for edge/node retrieving can be extremely slow. In our experiment, for Reddit [36] database, we observe that under the naive retrieving method, the processing time for one node can take up to 13.5 hours, where 99% of the time is spent on retrieving the node features.

In this work, we present SqueezePIR to accelerate the PIR processing of node feature retrieving. The key insight of SqueezePIR is that GNN algorithms usually demonstrate a certain level of robustness [19, 73], and some noises in the node features are tolerable. Therefore, our SqueezePIR compresses the node feature using low-rank approximation, which allows us to perform much less computation and greatly reduces the execution time.

However, additional homomorphic matrix multiplication is introduced during PIR for a decomposed dataset. We cannot directly apply prior PIR solutions such as FastPIR to the decomposed database. The naive solution to perform the homomorphic matrix multiplication [34, 35] may cause performance degradation even when compared to the original database without compression. Thus, our SqueezePIR protocol minimizes the overhead of reconstructing the retrieved data from approximations, we design a vectorized dataflow that only requires element-wise operations and avoids executing sophisticated matrix multiplications directly. On the other hand, a homomorphic encryption scheme has a limited range for computation, direct decomposition may cause the intermediate data to be out of the boundary during the data recovery. We proposed normalized decomposition to restrict the range of every computation. Based on our evaluation the proposed SqueezePIR achieves 11.85× speedup when compared to the state-of-the-art FastPIR protocol [3].

We summarized our contribution as follows:

- We propose SPG, which leverages PIR to protect the structure information of a graph database.
- We propose SqueezePIR, which accelerates the PIR processing on the server through approximation.
- We implement SPG and SqueezePIR with SEAL library[57] and evaluate them with various GNN workloads. The results show 11.85× speedup on average with less than 2% accuracy loss when compared to the FastPIR.

2 BACKGROUND

In this section, we introduce the basis of graph neural networks, fully homomorphic encryption, and private information retrieval.

2.1 Graph Neural Network

Graph Neural Network (GNN) applies deep-learning-based algorithms to extract information from graph-structured data, which has demonstrated its effectiveness in different areas such as recommendations system [66, 67] and fraud detections [29, 46]. The input of GNN is a graph. Figure 1(a) shows an example of an undirected graph, which consists of 5 nodes and 5 edges. To store this graph, we use the Compressed Sparse Row (CSR) format, as shown in Figure 1(b). In CSR format, there is a neighbor array and an offset array. The neighbor array keeps the neighbor indices of each node sequential, and the offset array records the starting/ending position of these neighbors accordingly. Moreover, each node in the graph often has a feature vector. The feature vector contains additional information about the node, as shown in Figure 1(c). The length of the feature vector for each node is identical.

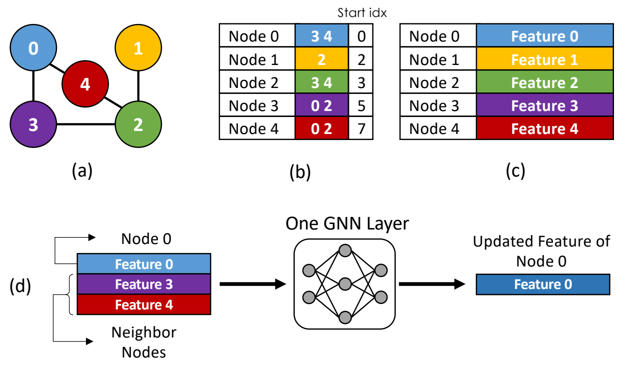


Figure 1: Graph Neural Network Inference: (a) topology of a graph; (b) edge lists of each node; (c) features of each node; (d) inference of one GNN layer.

GNN is an iterative algorithm that learns node/edge information through the connectivity between nodes as shown in Figure 1(d). A GNN model is usually composed of K layers, and each layer contains an *aggregation* function and a *combination* function. For each node, the *aggregation* function takes inputs of the feature vectors from its neighbors, and then the result is used to update its own feature vector in the *combination* function.

Equation 1 expresses the operations of *aggregation* function, where a_v is the aggregated result of node v, k is the k-th layer of the GNN model, h_u is the feature vector of node u, and $\mathcal{N}(v)$ is the neighbor set of node v. Meanwhile, Equation 2 shows the operations of *combination* function, which combines the aggregation result in layer k with the feature vector in layer $k-1$ and generates an updated feature vector.

$$a_v^k = aggregate\ (h_u^{k-1} | u \in \mathcal{N}(v)) \tag{1}$$

$$h_v^k = combine\ (a_v^k, h_v^{k-1}) \tag{2}$$

As the graph data keeps scaling in the real world, the graph structure and feature vectors increase quickly. Moving the data to the cloud for more extensive storage can be a good option.

2.2 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) is a type of encryption scheme that allows generic operations over encrypted data. Modern FHE schemes include BFV [11, 31], BGV [12] and CKKS [20]. These

FHE schemes usually encrypt a vector of raw data into a ciphertext. Thus, the supported operations for these schemes are element-wise computation between vectors and elements rotation for a vector.

Algorithm 1: FHE Encryption/Decryption

```
1 Function Encrypt(V, pk):
      /* Encrypt a vector $V = [v_1, v_2, v_3]$ into ciphertext $C$ with
         public key $pk$.                                          */
2 return C

3 Function Decrypt(C, sk):
      /* Decrypt a ciphertext $C$ into a plain vector
         $V = [v_1, v_2, v_3]$ using the secret key $sk$.          */
4 return V
```

Algorithm 1 shows the encryption and decryption functions in FHE. The `Encrypt` function encrypts a raw vector V into a **ciphertext** C. During encryption, the vector V is termed as **message**. The message is first encoded into a polynomial, called a **plaintext**. Then, the plaintext is encrypted to the **ciphertext**. Usually, the ciphertext C is composed of two vectors that store the coefficients of two polynomials. The degree of polynomials in the ciphertext decides the maximum amount of elements in the original message vector. The encryption phase needs the public key pk. The `Decrypt` function decrypts a ciphertext to the original message vector. The decryption requires a secret key sk that is only known by the client.

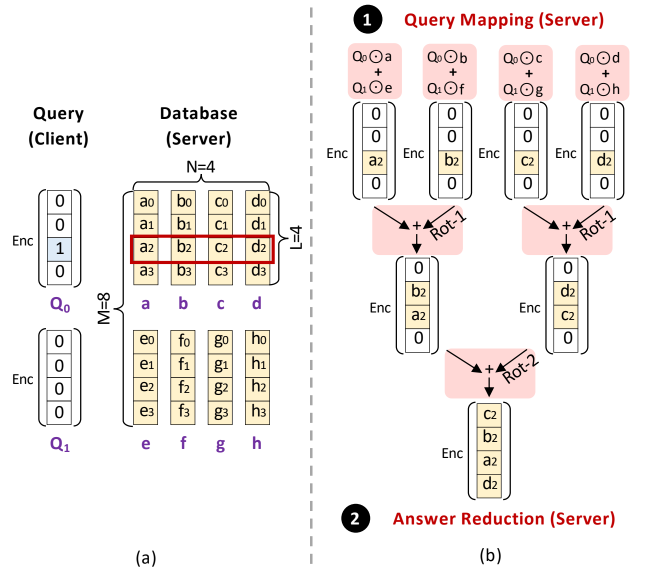
For FHE computations, three types of operations are usually supported: `Hom_Mul`, `Hom_Add`, and `Hom_Rot`. The `Hom_Add` and `Hom_Mul` take two ciphertexts as input and return the encryption of element-wise addition/multiplication. Note that these two functions can also take a plaintext W as input.`Hom_Rot` operates on a single ciphertext. It rotates the elements in the original message vector according to *step*. The sign of *step* denotes the direction of rotation. For example, with $V = [v_1, v_2, v_3]$, rotating V one step to the left ($step = -1$) will result in an encryption of $[v_2, v_3, v_1]$. For different values of steps, FHE requires different rotation keys rk; these keys are generated by the client.

As a remark, for the graph stored on the server, we need to encrypt the content of graph data, i.e. the edge list and feature data for each node. Therefore, in this work, most operations are performed between ciphertexts.

2.3 Private Information Retrieval

Private Information Retrieval (PIR) is a protocol in which a client accesses a database server anonymously, i.e., when the client retrieves a record from the database, PIR protocol ensures that the server does not know which record is retrieved. PIR is widely applied to private keyword search [53, 70], content distribution [33, 49], and anonymous communications [3, 5].

The design of a PIR protocol falls into two lines: information theoretic PIR (IT-PIR) [9, 22, 26, 27] and computational PIR (CPIR) [3, 4, 15, 28, 49]. The IT-PIR replicates the database across multiple non-colluding servers. The client can send different queries to these servers and derive the answer by combining the responses from servers. It has been proven that such schemes are information-theoretic secure against adversarial attacks [22]. On the other hand, CPIR puts the database in a single server and provides the other way around against computationally-bounded adversaries (i.e., the attacker performs limited computations). In this work, we focus on the single-server CPIR because it is more practical than deploying non-colluding servers.



Query (Client)
Database (Server)
N=4
L=4
M=8
Enc
Q_0
Q_1
a b c d
e f g h
1 Query Mapping (Server)
$Q_0 \odot a + Q_1 \odot e$
$Q_0 \odot b + Q_1 \odot f$
$Q_0 \odot c + Q_1 \odot g$
$Q_0 \odot d + Q_1 \odot h$
Rot-1
Rot-2
2 Answer Reduction (Server)
(a)
(b)

Figure 2: FastPIR: (a) The encrypted query generated by the client and the database stored on the server (b) data retrieving on the server.

FastPIR (Query and Database Settings): Currently, one of the most efficient CPIR protocol that based on FHE is FastPIR[3], which is shown in Figure 2. In this example, the database is an 8×4 matrix, and the client wants to retrieve the third record from the server. Assuming the degree of ciphertext is $L = 4$, which means the maximum number of elements in the original message vector is 4. Thus, FastPIR splits the database into two tiles along the column direction, and elements along the column direction are considered to form a vector. Now, the query sent by client is composed of two ciphertexts that correspond to the two tiles in the database. In this example, the third slot in the first ciphertext is set to 1 and other slots are set to 0. The second ciphertext is an all-zero vector. During the retrieving phase, the queries are encrypted by the client and sent to the untrusted database server. The query and the database settings are shown in Figure2(a).

FastPIR (Data Retrieving): Figure 2(b) shows the procedures of how to retrieve a record from the database through FastPIR. In general, data retrieval can be divided into two steps:

① Query Mapping: In this step, the encrypted query first performs `Hom_Mul` with each column of the database. Then, the intermediate results are aggregated together through `Hom_Add`. For example, the result of the first column after query mapping is $Q_0 \odot a + Q_1 \odot e$. We use $\odot$ and + to represnet the `Hom_Mul` and `Hom_Add`.

② Answer Reduction: After the query mapping, each column contains one ciphertext and each of them only has one valid slot, which wastes too much storage space. Thus, FastPIR designs a tree-based reduction method to combine results from different columns into one ciphertext. Specifically, the leaves of the tree are the results of each column after step ① ([0, 0, a_2, 0], [0, 0, b_2, 0], [0, 0, c_2, 0], [0, 0, d_2, 0]). In each level, two nodes that share the same parent will be combined together with Hom_Rot and Hom_Add. For example, in the first level, the first two leaves ([0, 0, a_2, 0], [0, 0, b_2, 0]) share the same parent node. The ciphertext [0, 0, b_2, 0] is rotated by one position left to get [0, b_2, 0, 0], and then added with ciphertext [0, 0, a_2, 0] to get [0, b_2, a_2, 0]. The rotation step for each level l equals 2^{l-1}. In this example, FastPIR follows the recursive tree-based rotation to generate the compact answer [c_2, b_2, a_2, d_2].

Finally, the reduced answer generated on the server will be sent back to the client, and the client can decrypt the result to recover the desired record. Note that the server does not know the content of ciphertexts in the whole process. Also, the server does not know which record is retrieved by the client.

3 STRUCTURE-PRIVATE GRAPH DATABASE

In this section, we present SPG. First, we discuss the database accessing framework for GNN applications. Then, we present the threat model of graph data access when facing an untrusted third-party server. Finally, we introduce our SPG system, built on top of the PIR protocol that enables secure graph data retrieval.

3.1 Accessing Graph Database

As the size of the graph in the real world increased explosively [42, 76], storing the graph data in the cloud is necessary. In this work, we focus on the scenario where both the graph structure data (CSR formatted) and the feature table are stored in the database. Based on Equation 1 and 2, when executing GNN models for graph applications, we need to first retrieve the neighbor indices of a node. Then, the feature vectors of neighbor nodes are retrieved accordingly and aggregated together to complete the *aggregation* phase. Next, we combine the aggregated result with the feature vector of the target node to complete the *combination* phase.

Table 1: The size of Graph Neural Networks and academic graph databases

GNN Models	Parameter Size (MB)
GraphSaint [71]	1.27
SAGN [58]	8.52
DeeperGCN [41]	0.97
GraphSAGE [36]	0.79
SEAL [72]	0.99

Graph Database	Edge List Size (MB)	Node Feature Size (MB)
Reddit [36]	534.99	617.98
Products [21]	934.23	324.68
Mag [62]	59.20	198.07
Collab [37]	115.17	5.52
Citation2 [62]	1429.67	160.30

Compared with the entire graph, the GNN model used for graph applications is much smaller. As shown in Table1, the size of graphs ranges from hundreds of MB to 1 GB. The industry graph databases [76] can reach TB-level. However, the size of GNN models ranges from 1 MB to 10 MB, which is far less than the graph size. Thus, we assume both the device (local) and server sides can run a given GNN model. In particular, the graph database is accessible by multiple trusted devices. These devices may collect the graph data for different GNN tasks.

3.2 Threat Model

In the graph database system, we assume that every device has the copyright of the database. Also, the devices share an identical secret key that can be used to decrypt a ciphertext. The attacker in the system is the curious-but-honest database server, who wants to steal the content of graph data (raw edge lists and node features) and the structure of a graph (the connection between nodes), however, the server will not modify the content of the retrieved data.

In our threat model, the graph structure can be revealed by accessing history on the server. During the data retrieving, although the server would follow the retrieving requests from the devices and return the correct results, the server also actively logs the retrieving tracks of each device and tries to recover the graph structure. Specifically, the server can identify which node the device is processing by tracking the index of retrieved data in the edge list database. After that, it can infer a node's neighbors by tracking the index of the retrieved data in the node feature database. With multiple rounds of retrieving, the server can potentially reconstruct the graph topology.

The connection information should be kept private for most applications such as finance, health care, and even recommendation systems. An attacker may infringe the copyright of the graph owned by the client. Also, the attacker may further infer user information based on the graph structure information.

3.3 SPG via PIR

Overview: The framework of our structure-private graph database (SPG) is shown in Figure 3. The graph data (edge lists and node features) are kept in the untrusted database server. The client can be a company, or an institute, that owns the graph data. Also, the client can own multiple trusted devices, where these devices are able to access the database individually. To perform GNN for one particular node, the device will first retrieve the edge list of the node. Then, according to the neighbor indices in the edge list, the device further retrieves the feature vectors needed for GNN execution. In our SPG system, the retrieving procedures for both the edge list and node features are protected through the PIR protocol. Therefore, the server will not know which nodes or edges are fetched. Finally, after running a GNN model, the device may update the client for necessary edge and node feature updates. The client will keep receiving updated data from devices and inserting or substituting the new encrypted data on the untrusted server. Since the data stored on the server are encrypted, the server cannot learn graph information during updating.

Data Organization: To facilitate the PIR process, the database should be organized as a "matrix", where one record is a row in the matrix and each record has the same length. This setting ensures that the returned answers are always the same size, and thus the server cannot identify the record based on the answer size. For the feature table in Figure 1(c), it is naturally a matrix since all feature vectors have the same length. However, the edge lists' lengths are different between nodes, since each node has a specific node degree. One naive solution to build the edge list matrix is padding the edge lists for the nodes that have low degrees. However, padding can introduce significant overhead because the edge list for every node needs to be expanded to the maximum degree.

To enable PIR protocol on edge lists, we adopt the CSR format and combine all edge lists together. The edge list array is then reshaped into a matrix. Each row in the edge list matrix may contain edges for multiple nodes, or one node's edge list could span over multiple rows in the edge list matrix. To retrieve an edge list, the device will derive the row ID(s) of edge lists in the database and perform

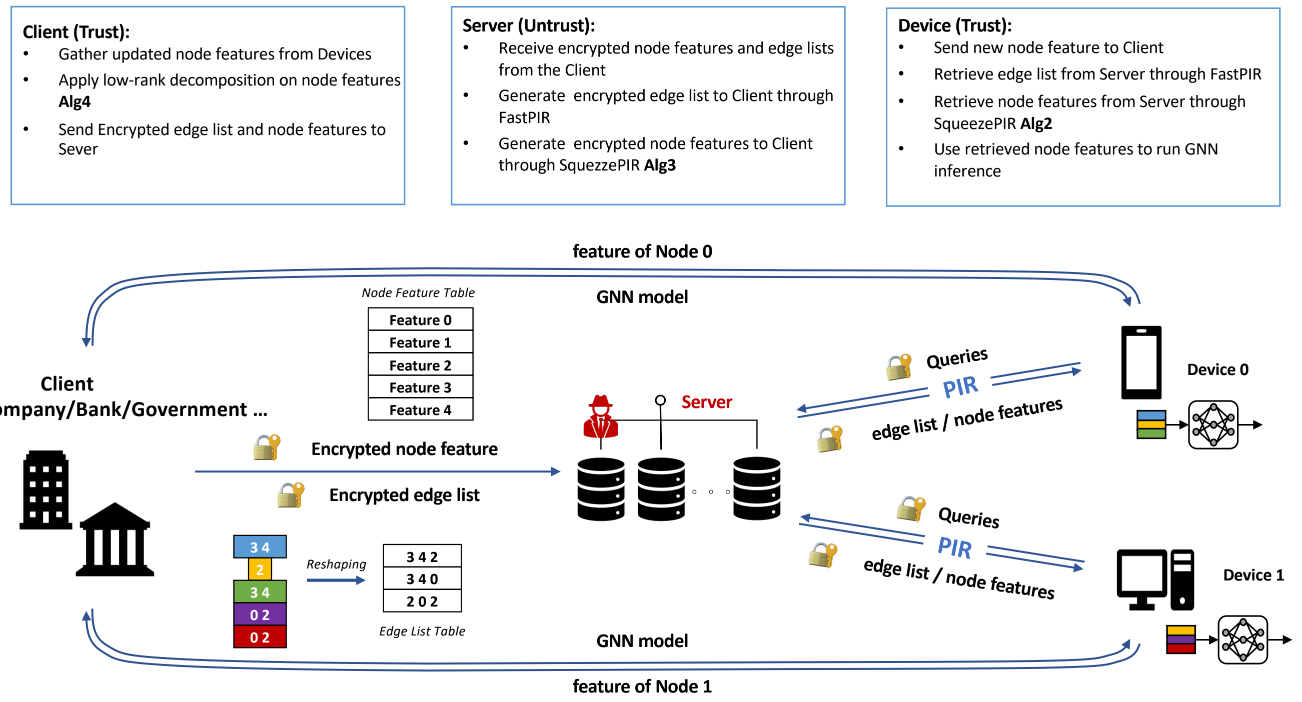

Client (Trust):
Gather updated node features from Devices
Apply low-rank decomposition on node features Alg4
Send Encrypted edge list and node features to Sever
Server (Untrust):
Receive encrypted node features and edge lists from the Client
Generate encrypted edge list to Client through FastPIR
Generate encrypted node features to Client through SqueezePIR Alg3
Device (Trust):
Send new node feature to Client
Retrieve edge list from Server through FastPIR
Retrieve node features from Server through SqueezePIR Alg2
Use retrieved node features to run GNN inference
feature of Node 0
GNN model
Node Feature Table
Feature 0
Feature 1
Feature 2
Feature 3
Feature 4
Client
Company/Bank/Government ...
Encrypted node feature
Encrypted edge list
Server
Queries
PIR
edge list / node features
Device 0
Queries
PIR
edge list / node features
Device 1
Reshaping
342
340
202
Edge List Table
GNN model
feature of Node 1

Figure 3: The framework of a structure-private graph database.

one or multiple PIR queries to fetch them anonymously. The device will keep a copy of the offset in the original array and compute the corresponding row ID(s) based on the edge list. In our experiments, we adopt the average edges per node to evaluate the edge list and feature vectors retrieving time .

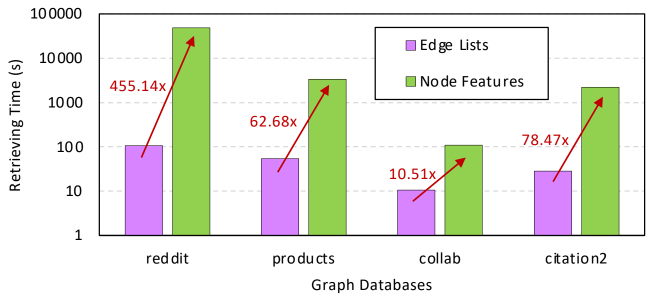


Figure 4: Retrieving time comparison between edge list and neighbor nodes' feature vectors for one node on different graph databases.

Accelerating Feature Retrieving: Performing PIR queries can be extremely expensive. Figure 4 shows the PIR processing time for the two PIR procedures: retrieving edge list and retrieving node features. We adopt FastPIR [3] during the evaluation. From the result, we can find that the node feature retrieving time is 10.51× to 455.14× longer than the edge list retrieving time. The main reason is that for a node, the feature vector size of its neighbor nodes is far larger than the index size of its neighbor nodes. Thus, feature retrieving can be a huge bottleneck in the system. In this work, we mainly focus on how to accelerate the PIR process for node feature retrieving. The key insight is that deep learning-based models are known as tolerable to noises. This means it is not necessary to keep the node features extremely precise. Thus, instead of retrieving the original node features, we trade off GNN accuracy for better performance.

In SPG, we apply different PIR protocols for edge list and node feature retrieving. The FastPIR protocol is used for edge list retrieving, due to the best efficiency in precise data retrieving. For node features, we propose SqueezePIR, which is an approximation-based PIR protocol that performs data retrieving on a decomposed database. SqueezePIR generates answers with controllable errors, and the device can make a trade-off between accuracy and performance.

4 SQUEEZEPIR PROTOCOL

In this section, we present the design of SqueezePIR. We approximate the database with low-rank decomposition, and SqueezePIR greatly reduces the required storage and computation cost after decomposition. The key challenge during the data retrieving is the online data reconstruction from decomposed matrices, which involves additional matrix multiplications compared with FastPIR. However, matrix multiplications with ciphertexts are not efficient.

Our key insight is that the query used for reconstruction is one-hot encoded, meaning only one slot in the query is non-zero. Therefore, we propose a new query mapping scheme through in-place matrix multiplication. In the rest of this section, we first introduce the protocol of SqueezePIR. Then, we discussed the in-place matrix-matrix multiplication scheme in detail, followed by the complexity analysis of our design.

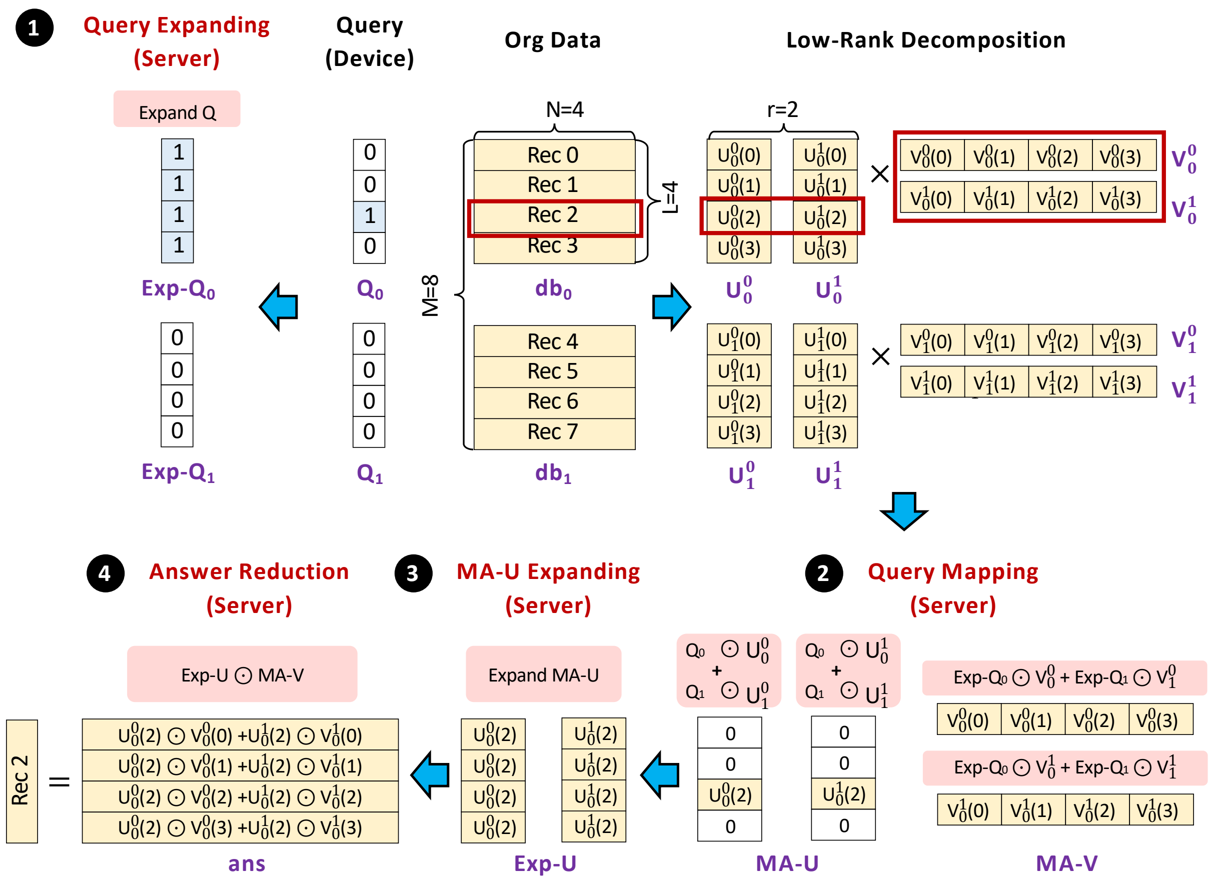


Figure 5: The overview of SqueezePIR's dataflow. The database is splited into tiles, each tile is decomposed into the low-rank form of $U_i \times V_i$. During the data retrieving, the ciphertexts in the queriy firstly performs in-place expansion. Then, the original query Q and the expended query $Exp\text{-}Q$ are mapped to matrix U and V to acquire $MA\text{-}U$ and $MA\text{-}V$, respectively. Next, we apply in-place expansion on $MA\text{-}U$. Finally, the expanded result $Exp\text{-}U$ is mapped to the matrix $MA\text{-}V$ and aggregated as the answer.

4.1 Protocol Design

The SqueezePIR consists of three interfaces: client API, device API, and server API. The client API performs database decomposition and encryption. The device API generates the encrypted queries for data retrieving. The server API executes the PIR protocol to compute the encrypted answers.

Database Decomposition: For a database matrix db, the client first partitions the database into tiles along the column direction and applies low-rank approximation (SVD decomposition) for each tile. The advantage of such partitioning is that for a growing graph with increasing nodes and edges, the client can attach new node features together as a new tile. Then, the decomposition is only performed for the new tiles without changing existing data in the database. The partitioned database tile db_i can be in any shape, but preferably with the same size of ciphertext length, i.e. $L \times L$.

We decompose each tile db_i into two singular matrices U_i and V_i, where $db_i = U_i \times V_i$. Then, we decrease the rank of matrices U_i and V_i to a predefined value. Figure 5 shows an example of database decomposition. Given an database with 8 records and each record with length 4, we split the 8×4 database into two 4×4 tiles: db_0 and db_1. After low-rank decomposition, each block db_i can be recovered by the matrix multiplication between U_i and V_i. By choosing a specific rank, we can approximate the original database db_i with columns from U_i and rows from V_i. Figure 5 shows an example of rank=2, where only the first two columns/rows in U_i/V_i are used to reconstruct the data.

The client then encrypts the two matrices U_i and V_i with the FHE primitive provided in Algorithm 1. To facilitate the computation, U_i is encrypted column-wise and V_i is encrypted row-wise. These ciphertexts are then stored in the server.

Query Generation: When accessing a record in the database, the device generates a query with multiple ciphertexts, and each ciphertext in the query will map to the corresponding tile in the database. By assigning 0 and 1 in the ciphertext slots, the query Q specifies the location of the retrieved record in the database. As shown in Figure 5, to retrieve the third record in the first database tile, the query is composed of two ciphertexts that correspond to the two tiles in the database. The first ciphertext encrypts a one-hot vector that sets the 3^{rd} slot to 1, and the second ciphertext encrypts

Algorithm 2: SqueezePIR Protocol – Device

```
/* Generate query to fetch $t$-th record from $M$ items */
1  Function QueryGen($t, M, L$):
       // $L$ – degree of ciphertext
2      $Q$=[$Q_0, Q_1, Q_2, ..., Q_{M/L-1}$];
3      for $i = 0; i < M/L; ++i$ do
4          $q$ = vector($L$, 0); // an all-zero vector with length $L$
5          if $t/L == i$ then
6              $q[t\%L] = 1$;
7          end
8          $Q_i$ = Encrypt($q$);
9      end
10 return $Q$
/* Decrypt the length-$N$ record from server's answer */
11 Function AnswerDec($ans, N, L$):
12     $record = [rec_0, rec_1, rec_2, ..., rec_{N/L-1}]$;
13     for $j = 0; j < N/L; ++j$ do
14         $rec_j$ = Decrypt($ans_j$);
15     end
16 return $record$
```

an all-zero vector. The detailed query generation in SqueezePIR is shown in Algorithm 2.

Naive Answer Generation: For the data retrieving on the server, we first introduce a naive solution that follows the nature computation flow on the decomposed database: After receiving a query Q, the server will map the ciphertexts in the query to every database tile. The answer is produced through Equation 3, where r is the rank used for approximation.

$$answer = \sum_i Q_i \odot U_i^{0:r} \times V_i^{0:r} \qquad (3)$$

If we follow the computation flow in Equation 3, the answer generation can be concluded in the following steps: Firstly, the ciphertext Q_i is mapped to $U_i^{0:r}$ to get $(Q_i \odot U_i^{0:r})$ through element-wise multiplication. This step sets the unwanted records to zeros. Secondly, the resulting $(Q_i \odot U_i^{0:r})$ is multiplied with $V_i^{0:r}$ to generate the answer of a tile. Finally, by adding all tile answers from different tiles we can derive the final answer.

However, in this naive solution, the answer generation process is not efficient due to two reasons: First, the computation in Equation 3 involves matrix multiplications with non-square matrices. Since matrices are represented in ciphertexts and only vectorized operations are supported, matrix multiplication with FHE requires complicated data reshaping and extra ciphertexts for intermediate data. Second, the generated answer after the computation has the size of 4×4, where only the third row contains the desired record and the other three rows are zeros. Therefore, the answer is very sparse, which causes communication inefficiency if we directly send this answer back to the device. To address these issues, we introduce our in-place query mapping scheme in the next sub-section.

4.2 In-place Query Mapping Scheme

We propose an in-place query mapping scheme that performs the answer generation directly using ciphertexts in Q, U, and V, without introducing additional ciphertexts for intermediate data. The detailed workflow is shown in Figure 5. The key idea of the scheme is to leverage the "*all-in-one*" nature for PIR protocols, where only one slot in Q and $Q \odot U$ is non-zero. Thus, we can utilize the remaining slots to store the intermediate result. We summarize the in-place query mapping scheme into four steps:

① Query Expanding: After the low-rank decomposition, we can use ciphertexts in query Q_i to locate the valid slots in matrix U_i. However, in SqueezePIR we also need to select the corresponding matrix V_i among different tiles. Thus, in the first step, we expand the ciphertexts in the original query Q_i to *Exp*-Q_i, such that the valid tile V_i corresponds to an all-one vector. For other tiles, the corresponding ciphertexts *Exp*-Q_j ($j \neq i$) are all-zero vectors. In our example, after the query expanding, the first expanded ciphertext *Exp*-Q_0 is the encryption of an all-one vector that corresponds to the first tile in the database. The detailed expanding method will be explained later.

② Query Mapping: In the second step, we will select the desired record from the database. In SqueezePIR, the encrypted query Q first performs Hom_Mul with each column in matrix U. Next, for each column, the intermediate results for different tiles (i.e., $Q_0 \odot U_0^0$ and $Q_1 \odot U_1^0$ for the first column) are accumulated with Hom_Add to get *MA-U*. Next, SqueezePIR will select the corresponding matrix V_i among different tiles. In this example, we want to select the matrix in the first tile. For each matrix V_i, every row first performs Hom_Mul with the identical ciphertext in the expanded query *Exp*-Q_i. Then, the intermediate results for the same row but different tiles (i.e., Exp-$Q_0 \odot V_0^0$ and Exp-$Q_1 \odot V_1^0$ for the first row) are accumulated together through Hom_Add to get *MA-V*. Since only *Exp*-Q_0 in this example is an all-one vector, V_0 will be selected.

③ *MA-U* Expanding: The third step is a preparation for the final answer accumulation. Because of the "all-for-one" character of PIR, each column in *MA-U* only has one valid slot. The valid slots in *MA-U* can be seen as a vector that needs to perform vector-matrix multiplication with *MA-V* to compute the original record. Specifically, for each row in matrix *MA-V*, every element will multiply with the same valid slot in the corresponding column of $MA - U$. Thus, for each column in *MA-U*, we fill other slots with the valid slot to expand *MA-U* to *Exp-U*. In our example, for each column in *MA-U*, every slot is filled with the value in the third slot after the expansion. With such expansion, each column in the *Exp-U* corresponds to a specific row in *MA-V* that makes the following vector-matrix multiplication fit the homomorphic encryption scheme perfectly.

④ Answer Reduction: In the last step, the intermediate results are combined together to recover the original record. In SqueezePIR, the results after Hom_Mul between columns in *Exp-U* and rows in *MA-V* will be added through Hom_Add to get the final answer.

Recursive Vector Expanding: For step ① and ③ in our query mapping scheme, we want to achieve expanding operation on a one-hot encrypted vector. Specifically, for a one-hot encrypted vector, we want to fill all zero slots in the vector with the same valid value. Thus, the valid slot expanding becomes an overhead, and an efficient algorithm is required. We adopt recursive rotations and additions

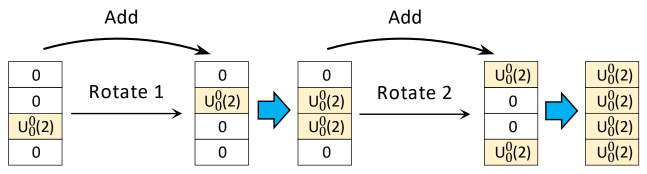


Figure 6: The expanding process for an one-hot ciphertext, where we use recursive rotations and additions.

to limit the expanding overhead. An example is shown in Figure 6. For a one-hot vector with length L, only one slot is valid at the beginning, $log_2 L$ times Hom_Rot and Hom_Add are required to finish the slot expansion. Combined with the expanding function, the detailed algorithm of how to retrieve an entry from the database through SqueezePIR is shown in Algorithm 3. After the *answer* is retrieved from the server side, the device can use the AnswerDec function in Algorithm 2 to decode the desired record.

Table 2: Storage and computation comparison between FastPIR and SqueezePIR.

	Storage	Hom_Mul	Hom_Rot
FastPIR	MN	MN	N
SqueezePIR	2rMN/L	2rMN/L+rN	(M/L + rN/L)log_2L

4.3 Complexity Analysis

In this subsection, we analyze the storage consumption and computation complexity of our SqueezePIR. Assume the database has been already tiled into $L \times L$ squares, i.e., $db = \{db_{i,j}\}$. By taking advantage of low-rank decomposition, SqueezePIR reduces the storage cost from MN to $2rMN/L$ as shown in Figure 5. Here, M and N are the rows and columns of the original database, r is the pre-defined rank after the low-rank decomposition. From the computation perspective, Hom_Mul and Hom_Rot occupies most of the computation resources. Since low-rank decomposition can reduce the database size dramatically, the amount of Hom_Mul in SqueezePIR is $2rMN/L+rN$ (step ② and ④). In SqueezePIR, the vector expanding requires recursive rotations in step ① and ③, which takes $(M/L + rN/L)log_2 L$ times Hom_Rot. A detailed comparison of storage and computation cost between FastPIR and SqueezePIR is shown in Table 2.

5 DATABASE DECOMPOSITION

Currently, most homomorphic encryption schemes have a limited range of computations. Once the result exceeds the limit, the resulting ciphertext cannot be decrypted correctly. We need to guarantee that every intermediate data during computation is inside the boundary. However, analyzing the boundary during the answer computation for the entire graph database is inefficient. Thus, we first determine that the computation overflow can occur during the database reconstruction. Then, we propose a normalized decomposition framework to restrict the range of every computation. As a result, we only need to set the homomorphic encryption security parameters once for arbitrary databases.

Algorithm 3: SqueezePIR Protocol – Server

```
/* Expand the one-hot ciphertext c in-place                    */
Function CipherExpand(c):
    L = c.length;
    for i = 1; i < L; i *= 2 do
        c += HomRot(c, i);                    // Recursive expanding
    end
return c
/* Generate answer from the $M \times N$ database              */
Function AnswerGen($U, V, Q, M, N, r, L$):
    // r – number of ranks, L – Length of ciphertext
    // database is tiled into $L \times L$ squares: $db = \{db_{i,j}\}$
    // $U_{i,j}$, $V_{i,j}$ correspond to $db_{i,j}$
    // $U_{i,j} = [U^0_{i,j}, U^1_{i,j}, ..., U^{r-1}_{i,j}]$, $V_{i,j} = [V^0_{i,j}, V^1_{i,j}, ..., V^{r-1}_{i,j}]^T$
    $ans = [ans_0, ans_1, ..., ans_{N/L-1}]$;
    for $i = 0; i < M/L; ++i$ do
        $Exp\text{-}Q_i$ = CipherExpand($Q_i$);          // Step 1
    end
    for $j = 0; j < N/L; ++j$ do
        $ans_j$ = Encode(vector($L$, 0));
        for $k = 0; k < r; ++k$ do
            $MA\text{-}U$ = Encode(vector($L$, 0));
            $MA\text{-}V$ = Encode(vector($L$, 0));
            for $i = 0; i < M/L; ++i$ do
                $MA\text{-}U \mathrel{+}= Q_i \odot U^k_{i,j}$ ;              // Step 2
                $MA\text{-}V \mathrel{+}= Exp\text{-}Q_i \odot V^k_{i,j}$ ;   // Step 2
            end
            $Exp\text{-}U$ = CipherExpand($MA\text{-}U$) ;       // Step 3
            $ans_j \mathrel{+}= Exp\text{-}U \odot MA\text{-}U$ ;   // Step 4
        end
    end
return $ans$
```

5.1 Operation Analysis in SqueezePIR

First, we analyze the computations in SqueezePIR to determine which operations can cause the result exceeding the limit.

According to the dataflow in Figure 5, step ① and ③ apply recursive vector expanding on a one-hot encrypted vector. The range of intermediate results will not change during the expansion. In step ②, the server computes $\sum_i Q_i \odot U^k_{i,j}$ and $\sum_i Exp\text{-}Q_i \odot V^k_{i,j}$. Because there is only one one-hot ciphertext in the original query Q and one all-one ciphertext in the expanded query $Exp\text{-}Q$, all results produced in step ② will not exceed the boundary of the data in matrices U and V.

However, after the step ④, the result may exceed the original data range. Because each element in the ans_i is computed by vector-matrix multiplication between the non-zero row in $MA\text{-}U$ and matrix $MA\text{-}V$. Since we do not have constraints on $MA\text{-}U$ and $MA\text{-}V$, the intermediate results may exceed the original boundary of data in U and V. In order to overcome the boundary exceeding problem, we need to make sure that the intermediate results are bounded during the low-rank approximation.

5.2 Data Range in SVD

We adopt singular value decomposition (SVD) for the low-rank decomposition. Since the reconstruction can cause overflow issues based on our analysis, we propose a method to bound the norm of intermediate results to [0,1].

Ordinary SVD: In SqueezePIR, each database tile, namely a $L \times L$ square matrix X is first decomposed to $X = A\Lambda B$ through SVD, where A and B^* are unitary matrices. Λ is a diagonal matrix that stores the singular values of X. Usually, the elements in Λ are arranged in descending order, and we can reformulate X as $X = A'B'$ where $A' = A\sqrt{\Lambda}$ and $B' = \sqrt{\Lambda}B$. For each element in the original matrix, its value can be computed with

$$x_{\alpha,\beta} = \sum_{\gamma=0}^{L} a'_{\alpha,\gamma} * b'_{\gamma,\beta} \tag{4}$$

where $a'_{\alpha,\gamma} = a_{\alpha,\gamma}\sqrt{\lambda_\gamma}$ and $b'_{\gamma,\beta} = \sqrt{\lambda_\gamma} b_{\gamma,\beta}$. We want to bound the intermediate results during the computation of arbitrary elements in X. We use $x_{\alpha,\beta}[:k]$ to represent the intermediate result, which can be formulated as

$$\begin{aligned} x_{\alpha,\beta}[:k] &= \sum_{\gamma=0}^{k} a'_{\alpha,\gamma} * b'_{\gamma,\beta} \\ &= \sum_{\gamma=0}^{k} \lambda_\gamma * a_{\alpha,\gamma} * b_{\gamma,\beta} \end{aligned} \tag{5}$$

Algorithm 4: Database Processing – Client

```
/* SVD-based low-rank decomposition on the $M \times N$ database */
Function low-rank decomposition($db, M, N, r, L$):
    // $r$ – number of ranks, $L$ – Length of ciphertext
    // database is tiled into $L \times L$ squares: $db = \{db_{i,j}\}$
    $U = \{U_{i,j}\}, V = \{V_{i,j}\}, \hat{\lambda} = \{\hat{\lambda}_{i,j}\}$;    // initialization
    for $i = 0; i < M/L; ++i$ do
        for $j = 0; j < N/L; ++j$ do
            $A, \Lambda, B$ = SVD($db_{i,j}, r$);    // apply SVD on $db_{i,j}$
            $\hat{\lambda}_{i,j} = |\lambda_0|$ ;    // maximum eigenvalue-norm in $\Lambda$
            $A'' = A\sqrt{\Lambda/|\lambda_0|}$;  $B'' = \sqrt{\Lambda/|\lambda_0|}B$;
            $A'' = A''[:, :r]$;   $B'' = B''[:r, :]$ ; // set rank to r
            for $k = 0; k < r; ++k$ do
                $U_{i,j}^k$ = BatchEncrypt($A''[:, k]$);
                $V_{i,j}^k$ = BatchEncrypt($B''[k, :]$);
            end
        end
    end
return $U, V, \hat{\lambda}$
```

Bound intermediate results by Diving $|\lambda_0|$: We use a_α and b_β to represent the α^{th} row and β^{th} column in matrix A and B, respectively. We further use $a_\alpha[:k]$ and $b_\beta[:k]$ to represent the first k elements in vector a_α and b_β. Since A and B are unitary matrices, we have

$$\begin{aligned} |a_\alpha[:k]| &= \sqrt{\sum_{\gamma=0}^{k} a_{\alpha,\gamma}^2} \leq \sqrt{\sum_{\gamma=0}^{L} a_{\alpha,\gamma}^2} = |a_\alpha| = 1, \\ |b_\beta[:k]| &= \sqrt{\sum_{\gamma=0}^{k} b_{\gamma,\beta}^2} \leq \sqrt{\sum_{\gamma=0}^{L} b_{\gamma,\beta}^2} = |b_\beta| = 1, \end{aligned} \tag{6}$$

when $k \leq L$. Since singular values in Λ are arranged in descending order, we have $|\lambda_0| = max_i|\lambda_i|$. For the intermediate result $x_{\alpha,\beta}[:k]$, the boundary of its norm can be formulated as

$$\begin{aligned} |x_{\alpha,\beta}[:k]| &= |\sum_{\gamma=0}^{k} \lambda_\gamma * a_{\alpha,\gamma} * b_{\gamma,\beta}| \\ &< |\lambda_0| * \sum_{\gamma=0}^{k} a_{\alpha,\gamma} * b_{\gamma,\beta} \\ &= |\lambda_0| * |a_\alpha[:k]| * |b_\beta[:k]| * cos\theta \\ &\leq |\lambda_0| \end{aligned} \tag{7}$$

Here, θ is the angle between vectors $a_\alpha[:k]$ and $b_\beta[:k]$. We divide the original matrix X with $|\lambda_0|$, and we formulate $\frac{X}{|\lambda_0|} = A''B''$, where $A'' = A\sqrt{\frac{\Lambda}{|\lambda_0|}}$ and $B'' = \sqrt{\frac{\Lambda}{|\lambda_0|}}B$. Now, the norm of all intermediate results during the matrix multiplication between A'' and B'' are limited to [0,1]. Finally, we retrieve matrix $\frac{X}{|\lambda_0|}$ from the server. The benefit of this design is that we can use the same secure parameters for arbitrary $L \times L$ square matrix.

5.3 Database Pre-Processing

Our SPG system requires the client to process the low-rank decomposition offline. After the database is decomposed tile by tile, the client needs to send the encrypted database to the server. After the device has retrived data from the server through SqueezePIR, the client also need to send the desired eigenvalue to the device to finish the answer recovering. Specifically, each $db_{i,j}$ in the database is divided by its $\hat{\lambda}_{i,j}$ (we use $\hat{\lambda}_{i,j}$ to denote $|\lambda_0|$ for $db_{i,j}$), the original answer will be recovered by $\hat{\lambda}_{t/L,j} * ans_j$ on the device side. Here, t is the ID of the desired record in the database and L is the degree of ciphertext. We summarize our database processing in Algorithm 4.

6 EVALUATION

In this section, we first evaluate our SqueezePIR with different database settings. Then, we analyze the performance of SqueezePIR and the accuracy of GNN models on real graph databases.

6.1 Experiment Setup

Homomorphic Encryption Setup: We adopt two homomorphic encryption schemes: BFV [11, 31] and CKKS [20]. BFV encrypts integers and performs homomorphic computation on integers. For the security parameters in BFV, we follow the settings in FastPIR [3]. Specifically, the coefficient bits for plaintext and ciphertext are 16 bits and 109 bits, respectively. We use BFV during the edge list retrieving and the baseline of FastPIR.

CKKS is another encryption scheme that can encrypt complex floating-point numbers. Since the graph node features are in floating-point format, we adopt CKKS for the node feature retrieving. For CKKS, we set the polynomial degree for the ciphertext to 8192, and the valid slots L per ciphertext equals half of the polynomial degree. Thus, there are 4096 slots per ciphertext. The number of bits for coefficients in the ciphertext is {60, 49, 49, 60}. The scaling factor is 2^{49}. The degree size is the most critical factor that affects SqueezePIR's performance. We analyze the impact of the degree size in Table 3. For a smaller degree, a lower rank is required for an equal amount of storage and computation in plaintext. The computation time increases with a larger degree size, however, the noise for PIR is large when we set the degree to 4096. Thus, we pick 8192 as the degree size in our evaluation.

Software Implementation: We implement our SqueezePIR and SPG system in C++, with homomorphic encryption algorithms from Microsoft SEAL library [57]. The CPU platform is Intel Xeon Platinum 8200 CPU @ 2.7GHz. The main memory size is 1 TB.

Table 3: SqueezePIR & SVD performance with different ciphertext degree settings on Citation2 dataset

rank	ciph degree	MSE nosie (SqueezePIR)	time (SqueezePIR)	MSE noise (SVD)	time (SVD)
64	4096	4.91×10^{-3}	6.88s	5.58×10^{-8}	104.57s
128	8192	9.97×10^{-18}	9.20s	1.47×10^{-8}	134.13s
256	16384	1.32×10^{-18}	21.36s	3.80×10^{-9}	359.38s

6.2 Analysis of SqueezePIR

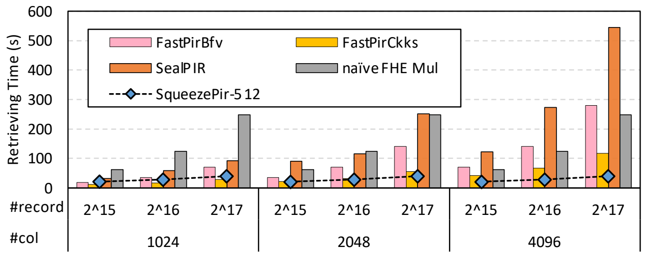


Figure 7: Retrieving time comparison between different PIR protocols: FastPIR [3] under BFV and CKKS, SealPIR [4], SqueezePIR with rank 512, and naive FHE matrix multiplication [34, 35].

Comparison between SqueezePIR and Other PIR Protocols: We first analyze SqueezePIR with self-defined parameter settings. Figure 7 shows the comparison between SqueezePIR and other PIR protocols. Here, we also adopt a naive solution to handle all homomorphic multiplication on the decomposed dataset. Usually, the FastPIR-based solution and our SqueezePIR are outperforms SealPIR and the naive method. Specifically, our SqueezePIR provides 5.54× and 4.53× speedup against SealPIR and the naive method on average. By comparing our SqueezePIR with the FastPIR-based solution, we made the following observations: Firstly, for FastPIR, the performance under BFV encryption is worse when compared to CKKS encryption. Since we need to encrypt the original database, all homomorphic operations are performed between ciphertexts, which may not be friendly for the BFV encryption scheme. Secondly, the performance of SqueezePIR does not change as the number of columns in the database increasing. The reason is that when the number of columns of the database is smaller than the number of slots of the ciphertext, the retrieving time for SqueezePIR is only influenced by the rank of the decomposed database if the number of records is fixed. Thirdly, when $\#col \leq 2\times\#rank$, the performance of SqueezePIR is worse than FastPIR under CKKS encryption. The reason is that in this circumstance, the size of the database after decomposition is equal to or even larger than the original database. Thus, when the number of columns in the original database is small, we will split the database along the column direction and concatenate them along the row direction, i.e. reshape an $M \times N$ database to a $m_1 \times m_2 N$ database, where $m_2 N \approx \#slot$.

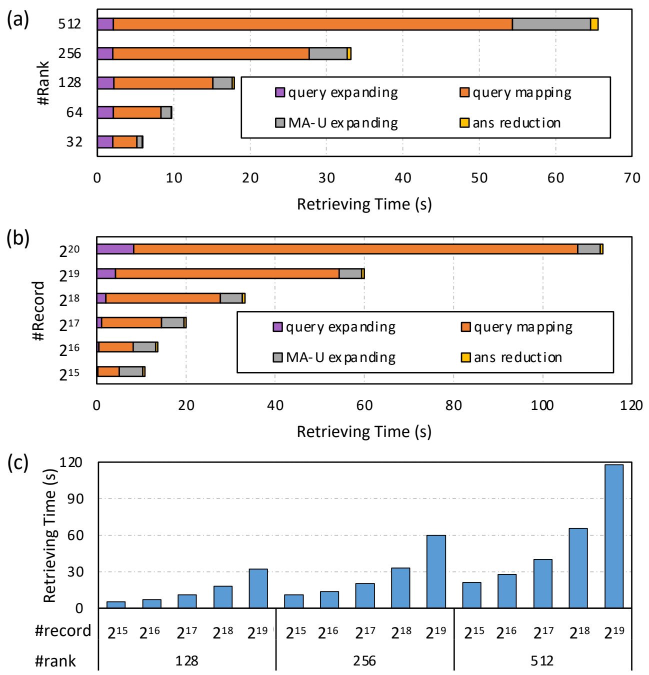


Figure 8: Retrieving time analysis of SqueezePIR: (a) retrieving time breakdown under various #rank with fixed #record (2^{18}) and (b) various #record with fixed #rank (256); (c) overall retrieving time comparison with different #record and #rank.

Breakdown of SqueezePIR: Next, we analyze the time consumption for each step in SqueezePIR. We first fix the number of records to 2^{18} and seek how the rank size affects the performance. The results are shown in Figure 8(a). From the results, we can easily find that the query expanding time is fixed since the query expanding is only influenced by the number of records in the database. All other steps in SqueezePIR grow linearly as the rank size increases. Also, we can observe that the query mapping (step ②) takes the most portion of retrieving time.

Then, we analyze the impact of record size by fixing the rank size to 256. The results are shown in Figure 8(b). We can observe that the time consumption for *MA-U* expanding (step ③) and answer reduction (step ④) are fixed since these two steps are only affected by the rank size. Other steps grow linearly as the record size increases. Like the results in Figure 8(a), the query mapping (step ②) takes the majority of retrieving time.

Figure 8(c) shows the retrieving time on the server with various record and rank sizes for SqueezePIR. Previously we find that the query mapping occupies most of retrieving time, and its execution time is proportional to the rank and record sizes. Thus, the overall retrieving time for SqueezePIR is increasing linearly as the rank and record size increase.

Table 4: Settings for various graph databases.

		Reddit [36]	Products [21]	Collab [37]	Citation2 [62]	Syn1	Syn2
original settings	#node	232965	2449024	23568	2927963	-	-
	#feature	602	100	128	128	-	-
	#edge/node (avg)	619	26	6	11	-	-
edge database	#record	49152	24576	4096	12288	-	-
	#column	4096	4096	4096	4096	-	-
node feature database	#record	40960	61440	8192	94208	131072	262144
	#column	3612	4000	4096	4096	4096	4096

Table 5: Performance of SqueezePIR for one node's edges and features retrieving on real-word graph databases. F and S represent FastPIR and SqueezePIR respectively. The retrieving time is in seconds.

	selected rank	GNN model	original acc	low-rank acc	edge retrieving time	node feature retrieving time (F)	node feature retrieving time (S)	speedup for SqueezePIR
Reddit	128	GraphSaint [71]	96.69%	94.96%	106.16	25043.19	3578.25	6.83
Products	32	SAGN [58]	84.45%	82.90%	53.61	1657.37	53.27	16.01
Collab	32	PLNLP [64]	63.05%	61.94%	10.50	89.07	6.41	5.89
Citation2	32	SEAL [72]	86.76%	86.65%	28.38	1059.39	29.84	18.68

6.3 SqueezePIR on Secure Graph System

Graph Neural Network Setup: In this work, we adopt four real-world graph databases and two synthetic databases to evaluate the performance of SqueezePIR. The graph database settings are shown in Table 4. Since the average edge list and feature size for each node are always smaller than the number of slots (4096) per ciphertext, we reshape the database and make the number of columns close to 4096 for more efficient processing. In this work, we adopt commonly used one-hop sampling for GNN inference. Thus, one edge list and the corresponding neighbor nodes' features need to be retrieved during PIR.

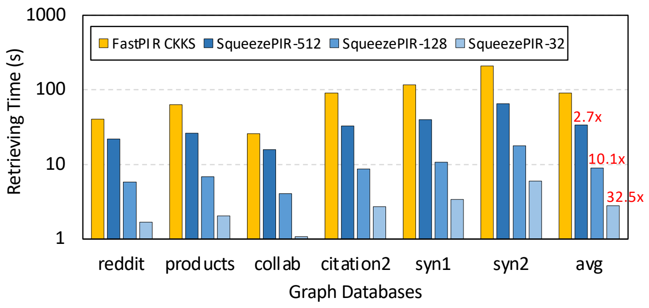


Figure 9: Retrieving time comparison between FastPIR and SqueezePIR for one node feature on different databases.

Node Feature Retrieving with SqueezePIR: We apply SqueezePIR on the databases provided in Table 4 and compare the performance with the FastPIR under CKKS encryption. The results are shown in Figure 9. Here, we set the ranks for SqueezePIR to 512, 128, and 32. From the results, we can find that the SqueezePIR achieves considerable speedup when compared to the FastPIR. Specifically, the performance improvement is more obvious for the databases with a larger number of records. The reason is that with more records, the query mapping occupies more portion of retrieving time as Figure 8(b), and SqeezePIR enjoys more speedup. On average, compared to FastPIR encrypted with CKKS, our SqueezePIR achieves 2.7×, 10.1×, and 32.5× speedup when the ranks are 512, 128, and 32, respectively.

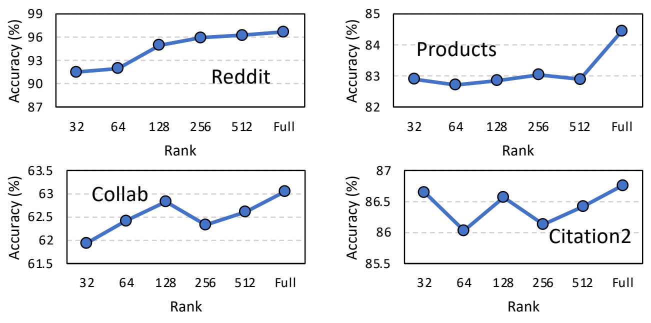


Figure 10: Accuracy of GNNs on various graph databases after low-rank decomposition.

GNN Accuracy under Low-rank Decomposition: The accuracy of GNNs after low-rank decomposition is shown in Figure 10. In most cases, GNNs can tolerate a small rank. We also find that the accuracy does not continuously decrease as the rank becomes smaller. The main reason is that the node features of a graph may have some redundant information. Low-rank decomposition helps to remove the redundancy that can help to improve the accuracy of the model. This phenomenon also appears in GNN pruning [18, 45]. Usually, the accuracy drop of pruning algorithms on GNN is larger than 2% [19, 54]. Thus, assuming we can tolerate 2% accuracy loss, we set the rank for Reddit to 128 and 32 for other databases. The overall performance comparison between SqueezePIR and FastPIR (CKKS) on real-world databases is shown in Table 5. During our evaluation, each node will first retrieve its edge list through FastPIR (BFV). Then, the node retrieves its neighbor nodes' feature vectors through FastPIR (CKKS) or SqueezePIR. Here, we adopt average edges as the number of neighbors for each node. From the result, our SqueezePIR achieves 5.89× to 18.68× speedup when compared to FastPIR (CKKS). On average, the SqueezePIR achieves 11.85×

speedup, which makes the node feature retrieving time close to the edge retrieving time.

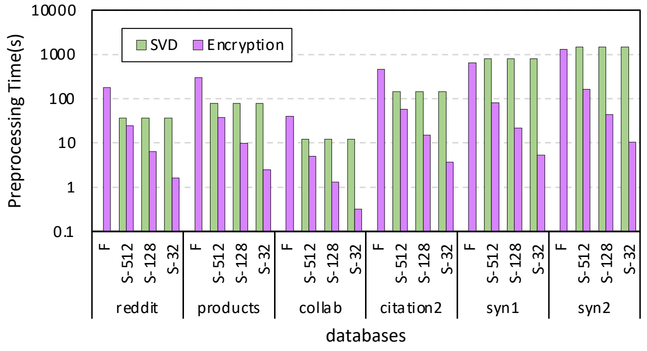


Figure 11: Database pre-processing time for FastPIR and SqueezePIR. (F and S represent FastPIR and SqueezePIR)

Database Encryption and Decomposition for SqueezePIR: We analyze the database pre-processing time in Figure 11. Compared to FastPIR, an additional low-rank decomposition should be applied to the original database before the encryption in SqueezeIPR. Since the SVD is performed for each $L \times L$ block, the pre-processing time of SVD for a larger graph database grows linearly. Compared to the results in Figure 9, the database pre-processing time is acceptable. For example, for database 'Syn2', the database pre-processing time is 25.2× compared to one node feature retrieving time when we set the rank to 512 under SqueezePIR. The reason that we can afford this overhead for database pre-processing is we only need to pre-process the entire database once, however, each node will retrieve tens to hundreds of neighbor node features for each inference.

Table 6: Execution time breakdown in the real deployment, where PIR protocols are executed under multi-thread mode.

	Reddit	Product	Collab	Citation2
edge retrieving	15.72	9.43	2.27	4.96
node feature retrieving	103.21	0.51	0.51	1.49
GNN inference	1.13×10^{-4}	1.79×10^{-5}	6.67×10^{-5}	4.64×10^{-5}

Discussion on Real-world Deployment: In real-world deployment, PIR protocols can be easily accelerated with parallelism execution. Specifically, FastPIR can parallel along the column direction, and SqueezePIR can parallel along the rank and batch direction. We use OpenMP dynamic parallelism [14] to accelerate PIR protocols. From the results in 6, the data retrieving still occupies most of the execution time, which needs to speed up. Also, we find that SqueezePIR shows a higher speedup since execution PIR in batch can enjoy more data reuse on the database.

7 RELATED WORK

Private Deep Learning and Graph Applications: For deep learning applications, the dataset can contain sensitive information and the parameters in the trained model are often confidential. Thus, privacy-preserving deep learning becomes an important topic. Prior work has leveraged various approaches to protect the deep learning process, including fully homomorphic encryption [6, 25], federated learning [43, 69], and much more.

For graph-formatted data, there are also various private systems to process graph algorithms. In order to hide personal information (important features) during the processing on the server, structural anonymization or differential privacy [8, 13, 16, 52] try to build a substitute graph by modifying the graph structure and node features to hide information. These approaches can prevent personal data leakage theoretically without degrading performance. However, these techniques are still vulnerable if the attacker has background knowledge [47, 55]. Also, the attacker can still train a GNN model based on the substitute graph. Some approaches focus on some specific graph applications such as shortest distance[32, 51, 63], nearest search[61], and graph neural networks[1, 74]. Compared to previous work, our SPG framework can encrypt the node feature and edge lists, which can guarantee the server learns nothing about the feature and structure information of a graph. Meanwhile, our framework can potentially let the server performs GNN workloads with homomorphic encryption [25, 40]. In this case, the GNN needs to be encrypted with FHE schemes, and thus the computation on the server will not leak any information.

Software Optimization for PIR: Extensive work has been done to improve the performance of PIR. Some work alleviates the overhead in PIR through new cryptographic techniques [49], while some work optimizes the query or answer with reduced size [3, 4]. There are also studies leveraging data pipelining and parallelism to accelerate answer generation [7]. Our SqueezePIR is the first to apply the compression method and directly reduce the data required for the PIR process.

Hardware Acceleration for PIR: Researchers also discussed the opportunity of accelerating PIR with emerging hardware. For example, some work leverages the extreme parallelism and computation capacity provided by GPUs [50] to reduce the execution time. There is also limited work designing customized accelerator architectures [44]. Our SqueezePIR does not put constraints on the computation platforms. Therefore, these acceleration schemes can also be applied to further boost the performance of SqueezePIR.

8 CONCLUSION

In this work, we explore how to build a GNN system to avoid the information of a graph being exposed to an untrusted third-party server. We first design an SPG framework which is built on top of PIR to prevent the server from learning the graph structure through the retrieving tracks. Then, we design SqueezePIR that can accelerate data retrieving by approximating the database with low-rank decomposition. Our evaluation shows an average of 11.85× speedup with less than 2% accuracy loss when comparing our SqueezePIR with the state-of-the-art FastPIR protocol.

9 ACKNOWLEDGEMENT

This work is supported by the National Science Foundation under Grant No. 2126327. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 308–318.

[2] Nahla Aburawi, Alexei Lisitsa, and Frans Coenen. 2018. Querying encrypted graph databases. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy*. SCITEPRESS-Science and Technology Publications.

[3] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 962–979.

[5] Sebastian Angel and Srinath Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 551–569.

[6] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. 2017. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security* 13, 5 (2017), 1333–1345.

[7] Karim Banawan and Sennur Ulukus. 2018. Multi-message private information retrieval: Capacity results and near-optimal schemes. *IEEE Transactions on Information Theory* 64, 10 (2018), 6842–6862.

[8] Aaron Beach, Mike Gartrell, and Richard Han. 2010. Social-k: Real-time k-anonymity guarantees for social network applications. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE, 600–606.

[9] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and J-F Raymond. 2002. Breaking the O (n/sup 1/(2k-1)/) barrier for information-theoretic private information retrieval. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*. IEEE, 261–270.

[10] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial attacks on node embeddings via graph poisoning. In *International Conference on Machine Learning*. PMLR, 695–704.

[11] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*. Springer, 868–886.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[13] Alina Campan, Yasmeen Alufaisan, Traian Marius Truta, and T Richardson. 2015. Preserving Communities in Anonymized Social Networks. *Trans. Data Priv.* 8, 1 (2015), 55–87.

[14] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[15] Yan-Cheng Chang. 2004. Single database private information retrieval with logarithmic communication. In *Australasian Conference on Information Security and Privacy*. Springer, 50–61.

[16] Zhao Chang, Lei Zou, and Feifei Li. 2016. Privacy preserving subgraph matching on large graphs in cloud. In *Proceedings of the 2016 International Conference on Management of Data*. 199–213.

[17] Melissa Chase and Seny Kamara. 2010. Structured encryption and controlled disclosure. In *International conference on the theory and application of cryptology and information security*. Springer, 577–594.

[18] Cen Chen, Kenli Li, Xiaofeng Zou, and Yangfan Li. 2021. DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1201–1206.

[19] Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang. 2021. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*. PMLR, 1695–1706.

[20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[21] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[22] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 41–50.

[23] Andrea Fronzetti Colladon and Elisa Remondi. 2017. Using social network analysis to prevent money laundering. *Expert Systems with Applications* 67 (2017), 49–58.

[24] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial attack on graph structured data. In *International conference on machine learning*. PMLR, 1115–1124.

[25] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 142–156.

[26] Daniel Demmler, Amir Herzberg, and Thomas Schneider. 2014. Raid-pir: Practical multi-server pir. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*. 45–56.

[27] Casey Devet, Ian Goldberg, and Nadia Heninger. 2012. Optimally robust private information retrieval. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 269–283.

[28] Changyu Dong and Liqun Chen. 2014. A fast single server private information retrieval protocol with low communication cost. In *European symposium on research in computer security*. Springer, 380–399.

[29] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. 2020. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 315–324.

[30] Rafał Dreżewski, Jan Sepielak, and Wojciech Filipkowski. 2015. The application of social network analysis algorithms in a system supporting money laundering detection. *Information Sciences* 295 (2015), 18–32.

[31] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[32] Esha Ghosh, Seny Kamara, and Roberto Tamassia. 2021. Efficient Graph Encryption Scheme for Shortest Path Queries. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 516–525.

[33] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and private media consumption with Popcorn. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 91–107.

[34] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Annual Cryptology Conference*. Springer, 554–571.

[35] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Annual International Cryptology Conference*. Springer, 93–120.

[36] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[37] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[38] Anish Khazane, Jonathan Rider, Max Serpe, Antonia Gogoglou, Keegan Hines, C Bayan Bruss, and Richard Serpe. 2019. Deeptrax: Embedding graphs of financial transactions. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 126–133.

[39] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[40] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* 10 (2022), 30039–30054.

[41] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. 2020. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739* (2020).

[42] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, et al. 2022. Hyperscale FPGA-as-a-service architecture for large-scale distributed graph neural network. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 946–961.

[43] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine* 37, 3 (2020), 50–60.

[44] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: in-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 102–115.

[45] Chuang Liu, Xueqi Ma, Yinbing Zhan, Liang Ding, Dapeng Tao, Bo Du, Wenbin Hu, and Danilo Mandic. 2022. Comprehensive Graph Gradual Pruning for Sparse Training in Graph Neural Networks. *arXiv preprint arXiv:2207.08629* (2022).

[46] Zhiwei Liu, Yingtong Dou, Philip S Yu, Yutong Deng, and Hao Peng. 2020. Alleviating the inconsistency problem of applying graph neural network to fraud detection. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 1569–1572.

[47] Nidhi Maheshwarkar, Kshitij Pathak, and Vivekanand Chourey. 2011. Privacy issues for k-anonymity model. *International Journal of Engineering Research and Application* 1, 4 (2011), 1857–1861.

[48] C Mawdesley, R Trueman, and WJ Whiten. 2001. Extending the Mathews stability graph for open–stope design. *Mining Technology* 110, 1 (2001), 27–39.

[49] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies* 2016 (2016), 155–174.

[50] Carlos Aguilar Melchor, Benoit Crespin, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. 2008. High-speed private information retrieval computation on gpu. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*. IEEE, 263–272.

[51] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. Grecs: Graph encryption for approximate shortest distance queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 504–517.

[52] Tamara T Mueller, Dmitrii Usynin, Johannes C Paetzold, Daniel Rueckert, and Georgios Kaissis. 2022. SoK: Differential Privacy on Graph-Structured Data. *arXiv preprint arXiv:2203.09205* (2022).

[53] Rafail Ostrovsky and William E Skeith. 2005. Private searching on streaming data. In *Annual International Cryptology Conference*. Springer, 223–240.

[54] Hongwu Peng, Deniz Gurevin, Shaoyi Huang, Tong Geng, Weiwen Jiang, Omer Khan, and Caiwen Ding. 2022. Towards sparsification of graph neural networks. *arXiv preprint arXiv:2209.04766* (2022).

[55] Keerthana Rajendran, Manoj Jayabalan, and Muhammad Ehsan Rana. 2017. A study on k-anonymity, l-diversity, and t-closeness techniques. *IJCSNS* 17, 12 (2017), 172.

[56] John Scott. 1988. Social network analysis. *Sociology* 22, 1 (1988), 109–127.

[57] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[58] Chuxiong Sun, Hongming Gu, and Jie Hu. 2021. Scalable and adaptive graph neural networks with self-label-enhanced training. *arXiv preprint arXiv:2104.09376* (2021).

[59] Lei Tang and Huan Liu. 2010. Graph mining applications to social network analysis. In *Managing and mining graph data*. Springer, 487–513.

[60] Binghui Wang and Neil Zhenqiang Gong. 2019. Attacking graph-based classification via manipulating the graph structure. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2023–2040.

[61] Boyang Wang, Yantian Hou, and Ming Li. 2016. Practical and secure nearest neighbor search on encrypted large-scale data. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[62] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies* 1, 1 (2020), 396–413.

[63] Qian Wang, Kui Ren, Minxin Du, Qi Li, and Aziz Mohaisen. 2017. SecGDB: Graph encryption for exact shortest distance queries with efficient updates. In *International Conference on Financial Cryptography and Data Security*. Springer, 79–97.

[64] Zhitao Wang, Yong Zhou, Litao Hong, Yuanhang Zou, and Hanjing Su. 2021. Pairwise Learning for Neural Link Prediction. *arXiv preprint arXiv:2112.02936* (2021).

[65] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 717–729.

[66] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2020. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)* (2020).

[67] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 346–353.

[68] Pengtao Xie and Eric Xing. 2014. Cryptgraph: Privacy preserving graph analytics on encrypted graph. *arXiv preprint arXiv:1409.5021* (2014).

[69] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. 2019. Federated learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 13, 3 (2019), 1–207.

[70] Rei Yoshida, Yang Cui, Rie Shigetomi, and Hideki Imai. 2008. The practicality of the keyword search using pir. In *2008 International Symposium on Information Theory and Its Applications*. IEEE, 1–6.

[71] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[72] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. 2021. Labeling Trick: A Theory of Using Graph Neural Networks for Multi-Node Representation Learning. *Advances in Neural Information Processing Systems* 34 (2021).

[73] Yuhao Zhang, Peng Qi, and Christopher D Manning. 2018. Graph convolution over pruned dependency trees improves relation extraction. *arXiv preprint arXiv:1809.10185* (2018).

[74] Zaixi Zhang, Qi Liu, Zhenya Huang, Hao Wang, Chengqiang Lu, Chuanren Liu, and Enhong Chen. 2021. Graphmi: Extracting private graph data from graph neural networks. *arXiv preprint arXiv:2106.02820* (2021).

[75] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* 2 (2020), 412–428.

[76] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).